NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

CLASS: A LOGICAL FOUNDATION FOR TYPEFUL PROGRAMMING WITH SHARED STATE

PEDRO MANUEL SABINO ROCHA

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
June, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CLASS: A LOGICAL FOUNDATION FOR TYPEFUL PROGRAMMING WITH SHARED STATE

PEDRO MANUEL SABINO ROCHA

Adviser: Luís Manuel Marques da Costa Caires
Full Professor, FCT-NOVA University of Lisbon

Examination Committee

Chair: Nuno Manuel Robalo Correia
Full Professor, FCT-NOVA University of Lisbon

Rapporteurs: Lars Birkedal
Full Professor, Aarhus University

Nobuko Yoshida
Full Professor, University of Oxford

Sam Lindley
Reader, University of Edinburgh

Vasco Manuel Tudichum de Serpa Vasconcelos
Full Professor, University of Lisbon

Luís Manuel Marques da Costa Caires
Full Professor, FCT-NOVA University Lisbon

Carla Maria Goncalves Ferreira
Associate Professor, FCT-NOVA University of Lisbon

Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, FCT-NOVA University of Lisbon

DOCTORATE IN COMPUTER SCIENCE

NOVA University Lisbon
June, 2022

CLASS: A Logical Foundation for Typeful Programming with Shared State

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.16) [100].

To my loving partner Fábio.

Acknowledgements

First and foremost, without whom this thesis would not be possible, I thank my advisor Luís Caires. Luís encouraged me though this long journey to always strive for the simplest, more elegant and principled solution. He introduced me to this wonderful world of logical foundations of programming languages, linear logic, concurrency, session types and π-calculus, always with excitement. I have enjoyed our several discussions, where I could see his brilliant mind in action, and where I have learned many skills, which I am sure will be valuable for the rest of my professional life.

I also thank the members of my thesis advisory committee, namely Sam Lindley, Bernardo Toninho and Vasco Vasconcelos, for their guidance and insightful discussions. Bernardo also joined us on some preliminary discussions and reviewed several or our drafts. Thanks also to Stephanie Balzer, Nobuko Yoshida and Philip Wadler for providing useful comments. To all the members of the PLASTIC research group for giving me the opportunity of presenting and discussing this work.

Throughout the PhD I had the pleasure of having several colleagues, many of which became friends, and made this journey much more pleasant: Eduardo Geraldo, Álvaro Santos, Mirko Engler, Filipe Meirim, Mário Pereira, Miguel Lourenço, Afonso Carvalho, Paulo Santos, Rita Macedo, Gil Alves, Serena Delli. I met Serena just in the beginning, when attending the subject Topics of Theory of Computation, and since then we are very close friends.

Outside the academic world, many were the friends that support me. I cannot mention them all, but of some them I need to: Sofia Farinha for making her comfortable couch always available when I most needed it, Sofia Araújo for being *always* there with her precious advices, Pedro Tavares for the fun and spark and Lilian Castiglione, with whom I spent, in the last phase, a lovely co-working season.

Thanks to the unconditional support of my loving family: my parents, my brother and my grandmother. To my most faithful four-legged and sweet companion Bess, that arrived just in the middle of the journey, but brought with her an immense energy. At last, to my loving partner Fábio, for his tremendous, relentless support and for always bringing so much joy and love to my life.

Abstract

Software construction depends on imperative state sharing and concurrency, which are naturally present in several application domains and are also exploited to improve the structure and efficiency of computer programs. However, reasoning about concurrency and shared mutable state is hard, error-prone and the source of many programming bugs, such as memory leaks, data corruption, deadlocks and non-termination.

In this thesis, we develop CLASS: a core session-based language with a lightweight substructural type system, that results from a principled extension of the propositions-as-types correspondence with second-order classical linear logic. More concretely, CLASS offers support for session-based communication, mutex-protected first-class reference cells, dynamic state sharing, generic polymorphic algorithms, data abstraction and primitive recursion.

CLASS expresses and types significant realistic programs, that manipulate memory-efficient linked data structures (linked lists, binary search trees) with support for updates in-place, shareable concurrent ADTs (counters, stacks, functional and imperative queues), resource synchronisation methods (fork-joins, barriers, dining philosophers, generic corecursive protocols). All of these examples are guaranteed to be safe, a result that follows by the logical approach.

The linear logical foundations guarantee that well-typed CLASS programs do not go wrong: they never deadlock on communication or reference cell acquisition, do not leak memory and always terminate, even if they share complex data structures protected by synchronisation primitives. Furthermore, since we follow a propositions-as-types approach, we can reason about the behaviour of concurrent stateful processes by algebraic program manipulation.

The feasibility of our approach is witnessed by the implementation of a type checker and interpreter for CLASS, which validates and guides the development of many realistic programs. The implementation is available with an open-source license, together with several examples.

Keywords: Propositions-as-Types, Shared State, Session Types, Linear Logic

Resumo

A construção de software depende de estado partilhado imperativo e concorrência, que estão naturalmente presentes em vários domínios de aplicação e que também são explorados para melhorar o a estrutura e o desempenho dos programas. No entanto, raciocinar sobre concorrência e estado mutável partilhado é difícil e propenso à introdução de erros e muitos *bugs* de programação, tais como fugas de memória, corrupção de dados, programas bloqueados e programas que não terminam a sua execução.

Nesta tese, desenvolvemos CLASS: uma linguagem baseada em sessões, com um sistema de tipos leve e subestrutural, que resulta de uma extensão metodológica da correspondência proposições-como-tipos com a lógica linear clássica de segunda ordem. Mais concretamente, a linguagem CLASS oferece suporte para comunicação baseada em sessões, células de memória protegidas com mutexes de primeira classe, partilha dinâmica de estado, algoritmos polimórficos genéricos, abstração de dados e recursão primitiva.

A linguagem CLASS expressa e tipifica programas realistas significativos, que manipulam estruturas de dados ligadas eficientes (listas ligadas, árvores de pesquisa binária) suportando actualização imperativa local, TDAs partilhados e concorrentes (contadores, pilhas, filas funcionais e imperativas), métodos de sincronização e partilha de recursos (bifurcar-juntar, barreiras, jantar de filósofos, protocolos genéricos corecursivos). Todos estes exemplos são seguros, uma garantia que resulta da nossa abordagem lógica.

Os fundamentos, baseados na lógica linear, garantem que programas em CLASS bem tipificados não incorrem em erros: nunca bloqueiam, quer na comunicação, quer na aquisição de células de memória, nunca causam fugas de memória e terminam sempre, mesmo que compartilhem estruturas de dados complexas protegidas por primitivas de sincronização. Além disso, uma vez que seguimos uma abordagem de proposições-como-tipos, podemos raciocinar sobre o comportamento de processos concorrentes, que usam estado, através de manipulação algébrica.

A viabilidade da nossa abordagem é evidenciada pela implementação de um verificador de tipos e interpretador para a linguagem CLASS, que valida e orienta o desenvolvimento de vários programs realistas. A implementação está disponível com uma licença de acesso livre, juntamente com inúmeros exemplos.

Palavras-chave: Proposições-como-Tipos, Estado Partilhado, Tipos de Sessão, Lógica Linear

Contents

List of Figures

1 Introduction

Several systems rely on concurrency and shared mutable state like, for example, aircraft systems and cloud computing networks, in which many sequential processes are being executed simultaneously and cooperate among themselves to attain a common goal. Besides being naturally present in several applications domains, shared mutable state and concurrency are also exploited to improve memory usage and speed up computations. Therefore, it comes at no surprise that mainstream programming languages offer support for imperative state, state sharing and concurrency.

Despite its ubiquity, reasoning about concurrency and shared mutable state is difficult, error-prone and the source of many programming bugs, particularly in the presence of other features such as aliasing, higher-order state, nondeterminism and locks. In fact, safely programming and reasoning about shared mutable state and concurrency has always been considered a significant challenge, which has lead to the exploration of several programming abstractions, logics and type systems. However, despite these explorations, a canonical solid foundation for shared mutable state and concurrency is still lacking. These solid foundations can be methodologically sought in logic via the celebrated propositions-as-types correspondence.

Proposition-as-types is a bridge connecting logic and computation, which allows knowledge to be transferred between these two fields. Languages built upon propositions-as-types satisfy certain metatheoretical properties, which hold because of the logical correspondence, for example: deadlock-freedom, confluence, normalisation, and the ability to reason about program behaviour through simple equational laws. Overall, these properties can save a lot of programming bugs, ease the task of building complex large-scale programs and are even used to improve program efficiency in a calculational way.

Propositions-as-types is quite widespread in the field of computer science: it lays the foundation of core functional calculi such as the simply-typed lambda calculus and system F, and it made its way to to concurrent programming languages via a correspondence from session types to linear logic. Unfortunately, the correspondence with linear logic limits the expressiveness of the computational model as it rules out programs involving

shared mutable state. In fact, in such a setting, the overall computation remains essentially functional. Some recent approaches try to address this problem but either diverge from a pure propositions-as-types foundation or fail to deliver the expected metatheoretical properties.

In this thesis, we develop CLASS, a core session-typed language with shared affine state. CLASS builds on top of the propositions-as-types correspondence between session types and second-order classical linear logic, extended with inductive types. The typing rules for the imperative fragment are inspired by those for the exponentials of differential linear logic (DiLL).

In CLASS, well-typed programs do not go wrong. They never deadlock on communication or reference cell acquisition and do not leak memory. Furthermore, they always terminate, which is a challenging result in the presence of first-class reference cells, higher-order polymorphic functional code and even recursion. These properties are established by lifting the basic acyclicity of communication topologies in linear logic to shared state, without the need to resort to ad-hoc devices such as partial orders. The proposition-as-types CLASS foundations then allows us to reason about the behaviour of concurrent stateful processes by doing algebraic program manipulations.

Any type discipline imposes a tradeoff between expressiveness and safety, language CLASS offers a relevant and interesting balance. The reference cells of CLASS are first-class objects which can be communicated around and shared dynamically by an arbitrary number of concurrent threads. Furthermore, in CLASS we can express a significant set of realistic examples, ranging from booleans and naturals to stateful memory-efficient linked data structures, such as linked lists and binary search trees, up to mutable shareable concurrent ADTs such as counters, stacks and queues. We can even express solutions to famous synchronisation problems such as the dining philosophers, which are guaranteed to be safe (deadlock-free) purely by the logical correspondence.

The feasibility of our propositions-as-types approach is witnessed by the implementation of a type checker and interpreter in Java for CLASS, which pragmatically validates and guides the development of many complex programs. The metatheoretical properties of CLASS provide strong guarantees for the implementation, which hold even when running complex examples. For instance, when running tests on our concurrent shareable ADTs, where everything is implemented using pure session-based processes, from the high-level ADTs menus to the basic inductive datatypes such as lists, the implementation spawns thousands of short-lived threads that synchronise perfectly thanks to the linear logic foundations.

The rest of the chapter is organised as follows. Section 1.1 presents the propositions-as-types correspondence between logic and computer science. Then, Section 1.2 focuses on the particular correspondence between session types and linear logic. Section 1.3 presents the challenges of bringing shared mutable state into the picture and Section 1.4 gives a taste of our proposed solution by giving a high-level overview of CLASS. Finally, Section 1.5 presents our contributions and lays out the thesis outline.

1.1 Propositions-as-Types

Propositions-as-types is a correspondence between logic and computer science. This correspondence is established by setting a back-and-forth map between two structures: (a) a deductive proof system for some logic and (b) a typed programming language. This map establishes a kind of dictionary in which propositions of a logical system are interpreted as types of a typed programming language, proofs as programs and proof simplification as program evaluation.

To illustrate, we will consider the seminal instance of propositions-as-types, that connects intuitionistic logic with the simply-typed λ-calculus. It was initially discovered by Curry [41, 42] and then worked by Howard [76], hence proposition-as-types goes also by the name of the *Curry-Howard correspondence*. We consider the implicational fragment of intuitionistic logic, where propositions include propositional variables and can be combined with the implication $\rightarrow$ connective. On the other hand, types in the simply-typed λ-calculus include type variables and can be combined with the functional arrow $\rightarrow$. On purpose, we use the same notation to denote implication and functional arrow, because they are connected via proposition-as-types: a proof of the proposition $A \rightarrow B$ is witnessed by a function that converts a proof of A into a proof of B. This interpretation of propositions goes by the name of *the Brouwer-Heyting-Kolmogorov interpretation* and it is considered the standard explanation of intuitionistic logic [8].

Judgments of intuitionistic logic are of the form $\Gamma \vdash A$ and can be read as follows: from a set of assumptions Γ we derive conclusion A. The rules of the deductive proof system for this logic are

$$\frac{}{\Gamma, A \vdash A}\ [\text{ax}] \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ [\rightarrow\text{I}] \qquad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ [\rightarrow\text{E}]$$

The identity axiom rule [ax] states that every assumption is a conclusion. Rule [$\rightarrow$I] introduces the implication connective $\rightarrow$ and it states that we can derive $A \rightarrow B$, whenever we can derive the conclusion B assuming A. Finally, rule [$\rightarrow$E] is the elimination rule for $\rightarrow$ and corresponds to *modus ponens*: we can derive B, whenever we derive both $A \rightarrow B$ and A.

On the other hand, processes of the simply-typed λ-calculus include process variables $x, y, z, \ldots$ and can be combined by defining λ-abstractions $\lambda x : A.\ M$ - which correspond to function definitions - and can also be combined by function application MN - where function M is applied to input N. The defining rules of the type system are as follows

$$\frac{}{\Gamma, x : A \vdash x : A}\ [\text{var}] \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\ M : A \rightarrow B}\ [\text{abs}] \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ [\text{app}]$$

If we forget the term annotations of the typing rules we obtain precisely the rules for intuitionistic logic, namely [var] corresponds to the identity axiom [ax], [abs] to the introduction rule [$\rightarrow$I] and [app] to *modus ponens*, rule [$\rightarrow$E].

One of the immediate practical consequences of the correspondence is that then we obtain a notation for proofs, we can write them as if they were programs. For example, the proof of proposition $A \to A$, on the left side, is denoted by the identity function, on the right

$$\dfrac{\dfrac{}{\Gamma, A \vdash A}\,[\text{ax}]}{\Gamma \vdash A \to A}\,[\to\text{I}] \qquad \dfrac{\dfrac{}{\Gamma, x : A \vdash x : A}\,[\text{var}]}{\Gamma \vdash \lambda x : A.\, x : A \to A}\,[\text{abs}]$$

Another example: the proof of proposition $A \to (B \to A)$, on the left side, is interpreted as the K combinator, which defines a constant function $B \to A$, that always returns k regardless of the argument x, displayed on the right

$$\dfrac{\dfrac{\dfrac{}{\Gamma, A, B \vdash A}\,[\text{ax}]}{\Gamma A \vdash B \to A}\,[\to\text{I}]}{\Gamma \vdash A \to (B \to A)}\,[\to\text{I}] \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, k : A, x : B \vdash k : A}\,[\text{var}]}{\Gamma, k : A \vdash \lambda x : B.\, k : B \to A}\,[\text{abs}]}{\Gamma \vdash \lambda k : A.\, \lambda x : B.\, k : A \to (B \to A)}\,[\text{abs}]$$

This opens a vast array of applications, because then we can have these proofs/programs mechanically checked by a computer and we can even use computers to help us in the complex, and sometimes daunting task, of finding proofs for our propositions. Indeed, several proof assistants and automated theorem provers were inspired by the correspondence, like for example Agda [19], Coq [40, 16], Isabelle [115] and Lean [111], to name just a few. Interestingly, this has led to the formalisation of a huge body of mathematics and even to the formal verification of complex pieces of software, as exemplified by the CompCert project [96].

Another consequence of propositions-as-types, one that we will explore in this thesis, is to lay principled foundations for programming languages. Programming languages built on top of a consistent logical theory automatically enjoy some desirable properties. For example: no program blocks when executing (deadlock-freedom), the outcome of a program is independent of the order in which instructions are executed (confluence), each program has a normal reduced form that summarises its behaviour (cut normalisation), all programs terminates (strong normalisation) and we can reason about program behaviour in a very simple and algebraic-like way (algebraic reasoning). Overall, these properties can save a lot of programming bugs, ease the task of building complex large-scale programs and are even used to improve program efficiency in a calculational way [103].

Propositions-as-types is a notion widespread in the field of computer science and inspires the development of *real-world* programming languages. The core simply-typed lambda calculus has influenced the development of programming languages such as Lisp [140], Haskell [78] and Scala [117]. After the seminal correspondence by Curry and Howard, several other instances were discovered like, for example, the correspondence between second-order logic and the polymorphic system F [62, 130], this has established the logical underpins of data abstraction [109] and has inspired the design of generics in the Java programming language [20]. Even classical logic was given a computational interpretation [120] by propositions-as-types, where the famous Peirce's law corresponds precisely to the type of continuation operators such as Scheme's call/cc.

These are just some examples, we refer the interested reader to [159] for an engaging expository overview of propositions-as-types and [66, 139] for those wishing to pursue a deeper technical study of the subject.

In this thesis, we shall be particularly interested in an instance of *a* propositions-as-types correspondence, that was established roughly a decade ago and that sets the foundations for session-based concurrent programming. More specifically, it connects session types to linear logic, and it will be presented in the following section.

1.2 Session Types and Linear Logic

In this section we discuss the propositions-as-types correspondence that connects session types and linear logic. Let us start with session types, which are protocols that structure the interaction of communicating processes. They were initially developed by Kohei Honda [73] and further adapted in [74]. In session-typed based systems, processes communicate through sessions. A session has two endpoints and each endpoint has an associated session type that describes the direction, order and the content of the exchanged messages.

The type system then guarantees that each endpoint user will follow the protocol dictated by its associated session type (session fidelity). As a consequence, communication between the two users of a session will run smoothly as in a perfectly synchronised interaction and there will be no mismatch between the type of exchanged data (communication safety).

Session types are a prolific research topic with several variants found in the literature, for example: multiparty [72], exceptional [57] and gradual session types [79]. Naturally, you also find them in the practical world of computing: they have inspired the development of some programming languages such as Links [39] and are also embedded in existing languages such as C [113], Java [77, 114], Haskell [127, 97] and Rust [81, 36], to name just a few. We refer the interested reader to [48, 153] for an introduction to the topic.

On the other hand, linear logic is a resource-sensitive logic developed (or discovered, if you will) by Girard [64]. It is essentially a substructural logic [129] that limits the application of weakening (the ability to discard resources) and contraction (the ability to copy resources) to only certain propositions, whereas in *traditional* classical and intuitionistic logic there is no such control. In other words, in traditional logic you can have a cake and eat it too, whereas in linear logic this is not the case: if you eat the cake then you do not have it and, furthermore, you must eat the cake. There are some other substructural logics, called affine, that relax this latter restriction and allow you to trash the cake without eating it. Interestingly, these kind of affine or ownership type systems [38, 45] are at the base of the popularity-growing programming language Rust [86].

Since its development, linear logic was looked upon as a promising avenue to establish the foundations of concurrent stateful programming languages [156]. A connection to computation was initially hinted by the works of Abramsky [1] and Bellin and Scott [14].

However, we would have to wait roughly two decades to see the foundations consolidated by the works of Caires and Pfenning [23] and Wadler [157], which finally established a propositions-as-types correspondence between session types and linear logic.

This correspondence revitalised the field and lead to a prolific exploration of further orthogonal logical or computational concepts, such is the modularity of a propositions-as-types approach. For example, the correspondence was extended in order to include parametric polymorphism [32, 157], dependent types [148], code mobility [144], multiparty protocols [22, 34], recursion [146, 99] and control effects [28].

By basing the development of session types on linear logic, the type system imposes a couple of conditions that guarantee that well-typed programs do not go wrong. Since processes cannot simply drop their session endpoints (no-weakening) they must fully follow the protocol until the end and, therefore, cannot leave other processes hanging for a message that will never arrive. Furthermore, the linear logic correspondence ensures that communication topologies are acyclic by construction. This acyclicity is then key to guarantee, on one hand, that programs do not block (deadlock-freedom) and, at the same time, that they do not diverge infinitely (termination). Remarkably, these properties are guaranteed at static time by the lightweight session type system.

Unfortunately, the correspondence with linear logic limits the expressiveness of the computational model as it rules out programs involving shared mutable state. Session typed programs based on the correspondence exhibit parallelism, in the sense that many sessions may run simultaneously and independently, but there is no sharing of linear sessions (no contraction). There is a limited form of sharing (applied to the so-called exponential or unrestricted sessions) which allows processes to invoke the same replicated server. Nonetheless, this shared server behaviour is identical for each invocation (cf. uniform receptiveness of shared names [136]), in other words: these replicated servers are stateless. In such a setting, the overall computation remains essentially *functional* [149], which is not surprising given the well known interpretations of linear logic as linear lambda calculi.

The problem is that shared mutable state is an essential ingredient of everyday programming that cannot simply be overlooked. But a successful attempt of integrating shared state into the propositions-as-types correspondence with linear logic [23, 157] faces several challenges. Because, as soon as we bring state into the picture some of the previously mentioned good theoretical properties *offered* by propositions-as-types are easily compromised. This is a topic that we shall explore further in the next section.

1.3 Shared State

A process behaviour is often stateful as it often depends on the history of its previous interactions. Furthermore, this state is often shared by several entities, as in, for example, a shared bank account. Besides being naturally present, shared state is also exploited in programming to speed up some computations or even to code memory-efficient data

structures such as linked lists. As a consequence, either of it being natural or of having a practical value, we may find shared state present in several programming languages. For example, in imperative or object-oriented languages such as C, C++ and Java, but even functional programming languages such as Haskell offer support for shared mutable state.

However, despite its ubiquity and practical value, programming with and reasoning about shared mutable state is quite hard, error-prone and the source of many bugs, particularly in the presence of other language features that are usually associated such as concurrency and aliasing. One immediate consequence of having shared state is that we no longer have referential transparency - the ability to replace equals by equals without changing the value of an expression. This force us to forgo equational program behaviour and the opportunity to simplify programs by doing simple algebraic-like manipulations.

Furthermore, if one wants to avoid data corruption, some form of thread synchronisation must be imposed in order to guarantee mutual exclusion by resorting, for example, to semaphores [50] or synchronisation monitors [68]. Unfortunately, then it becomes quite easy to express deadlock programs which are the source of many bugs [101]. Additionally, the interaction with other language features might have undesired consequences. For example, with first-class reference cells and recursion we can potentially express non-terminating programs such as the Landin's knot, compromising the normalisation properties.

Given its importance, we do not want to give up shared state, nonetheless we would like to mitigate some of its negative consequences. This is, in fact, a prolific research topic which has lead to the exploration of several programming abstractions, logics and type systems that in some way or another try to discipline the usage of shared mutable state like, for example [4, 31, 112, 116, 156, 141, 85].

Other approaches even build on top or are inspired by the propositions-as-types correspondence between session types and linear logic [10, 11, 7, 91, 128]. But, overall, they either diverge from a pure logic approach or fail to deliver the good metatheoretical properties, expected by propositions-as-types. For example, the work [11] relies on ad-hoc partial orders, whereas [7, 91] give up some important properties like confluence or algebraic reasoning. Furthermore, some of these approaches do not interact well with further orthogonal language constructs such as polymorphism, dependent types, control effects and even recursion.

It is the purpose of this thesis to show that the propositions-as-types correspondence between session types and linear logic is nevertheless sufficient to seamlessly accommodate shared mutable state, thereby laying the solid foundations for concurrent stateful programming. We show how this can be done without sacrificing the desirable metatheoretical properties, that should naturally follow from the basic linear logic foundation. And while still retaining a considerable degree of language expressiveness and the ability to accommodate other language constructs. In the next section we give an overview of the approach.

1.4 A Taste of CLASS

We will now give a high-level overview of the session-typed language CLASS, developed in this thesis, presenting some simple examples.

The Basic Language μCLL

The starting point is the pure fragment μCLL, which CLASS extends with stateful imperative constructs. Language μCLL is related, via propositions-as-types [23, 24, 157], to second-order classical linear logic, here extended with mix and inductive/coinductive session types.

In μCLL processes interact through session endpoints x which satisfy a certain protocol, described by a session type A. Session types are structured by an involutive operation of duality $A \mapsto \overline{A}$, where A and $\overline{A}$ are dual types, that corresponds to linear logic negation and guarantees that interactions occur always on matching dual pairs, i.e. when one process sends, the other receives, when offers a menu, the other chooses from that menu, and so on.

In μCLL, the basic fundamental operation to compose processes is expressed by the cut construct

$$\textbf{cut}\ \{P\ \ |x|\ \ Q\}$$

that composes two processes P and Q offering duality-related behaviours on each session endpoint x. In a cut $\textbf{cut}\ \{P\ \ |x|\ \ Q\}$, processes P and Q run concurrently communicating through a single private session x. This restriction - that two concurrent processes interact through a single session - guarantees acyclicity on the process communication topologies, which then implies desirable metatheoretical properties such as the absence of deadlocks.

Another way of composing processes is given by $\textbf{par}\ \{P\ ||\ Q\}$ where P and Q run in parallel without ever interfering with each other. This corresponds to linear logic mix rule. μCLL also has the bidirectional forwarding construct $\textbf{fwd}\ x\ y$, that corresponds to the identity axiom of linear logic and computationally acts as a link between two dual-typed endpoints x and y, forwarding all the interactions with x to y and vice-versa.

The basic propositional fragment of μCLL comprises the multiplicative units $\mathbf{1}/\bot$ that type session termination, the additives $A \,\&\, B/A \oplus B$ type menu offer and selection and the multiplicatives $A \otimes B/A \parr B$ that type session communication. We also have the exponentials $!A/?A$ that satisfy weakening and contraction and type replicated sessions and their invocation by clients.

μCLL also has existential and universal type-quantifiers $\exists X.A\ /\forall X.A$, which are interpreted by processes that respectively send and receive types. The type-quantifiers play a key role to define abstract data types (ADTs) and generic parametric processes. Finally, the inductive/coinductive session types $\mu X.\ A/\nu X.\ A$ allows us to type recursive/corecursive processes.

Language μCLL has already an interesting level of expressiveness, which compares with system F [149, 66]: we can capture some useful patterns of communication-based concurrency like session delegation and information hiding, we can even code basic datatypes such as booleans and naturals. For example, the inductive datatype Nat of the naturals is encoded in μCLL as

$$\mathsf{Nat} \triangleq \mu X.\ \oplus\{|\mathsf{Z} : \mathbf{1}\ |\mathsf{S} : X\}$$

A session $x : \mathsf{Nat}$ chooses either Z (zero), in which case it closes, or S (successor) in which case it recurs as $x : \mathsf{Nat}$. Then, processes $V_0(n), V_1(n), V_2(n) \ldots$ offering a behaviour of type Nat on session n, and corresponding to the naturals $0, 1, 2, \ldots$, are encoded as

$$V_0(n) \triangleq \mathsf{Z}\ n; \mathbf{close}\ n \quad V_1(n) \triangleq \mathsf{S}\ n; \mathsf{Z}\ n; \mathbf{close}\ n \quad V_2(n) \triangleq \mathsf{S}\ n; \mathsf{S}\ n; \mathsf{Z}\ n; \mathbf{close}\ n \quad \ldots$$

That is, each process $V_k(n)$ chooses k times option S on session n, after which it chooses Z and closes.

Whereas inductive types allows us to define recursive data structures, coinductive types allows us to define corecursive process definitions. For example, process $\mathsf{double}(n, m)$ produces the double of natural n on m:

$$\begin{array}{lcl} \mathsf{double}(n,m) & \triangleq & \mathsf{case}\ n\{\ \ \mathsf{Z} : \mathbf{wait}\ n; \mathsf{V0}(m) \\ & & \qquad\quad \mathsf{S} : \mathbf{cut}\ \{\mathsf{double}(n,k)\ \ |k|\ \ \mathsf{S}\ m; \mathsf{S}\ m; \mathbf{fwd}\ m\ k\}\} \end{array}$$

It starts by pattern matching on n: if n chooses Z (zero) we simply wait for n to be closed and continues as the natural $\mathsf{V}_0(m)$ on m. Otherwise: if n chooses S (successor) we spawn a corecursive call $\mathsf{double}(n, k)$ that interacts with its continuation and produces its double in k. The corecursive call is composed via a cut on k with a process that chooses twice S on m and then forwards to k.

Further examples can be expressed such as shared replicated boolean calculators. For example, the following process

```
calculator(s) ≜
!s(c);
case c {
  |dis :   recv c(x);
           par {discard(x) || close c}
  |neg :   recv c(x);
           send c(y.not(x, y));
           close c
  |conj :  recv c(x);
           recv c(y);
           send c(z.and(x, y, z));
           close c
}
```

defines a boolean calculator $\mathsf{calculator}(s)$ that persistently (as indicated by the bang !) offers a menu of three options: disc, to discard an inputted boolean x; neg, to compute the logical negation of an inputted boolean x; and conj, to compute the logical conjunction of two inputted booleans x and y. On each invocation on session s, the calculator spawns a session c to interact with the client, the session is explicitly closed in the end.

However, in μCLL computation is essentially functional: there are no side effects, no races and shared objects are stateless. We will see next how to bring shared state into the picture.

Reference Cells and Cell Usages

CLASS extends μCLL with basic imperative stateful operations, which are comprised by reference cells and cell usages, as displayed below

(1)	**cell** $c(a.P)$	(3)	**release** c
(2)	**empty** c	(4)	**take** $c(a); Q$
		(5)	**put** $c(a.Q_1); Q_2$

The process expressions (1) and (2) define reference cells on a session c. In CLASS a cell can either be full (1) or empty (2), typed by the modalities $\mathbf{S}_f\ A$ (state full) and $\mathbf{S}_e\ A$ (state empty), respectively. When the cell is full it stores a session a, the behaviour of which is implemented by a certain process P.

If in one endpoint of the session c we have reference cells, in the other we must have cell usages (3), (4) and (5), which are typed either by $\mathbf{U}_f\ A$ (usage full) or $\mathbf{U}_e\ A$ (usage empty). In (3) we are simply releasing a full cell usage $c : \mathbf{U}_f\ A$, this corresponds to coweakening in DiLL [53] and must be explicitly indicated in CLASS.

We can also take the session stored in a full reference cell c, as expressed in (4), after which the cell becomes empty. The stored session is then accessed by input parameter a and processed by the continuation Q.

In CLASS, each reference cell has an implicit lock which is acquired with the take operation. The continuation Q of the take must then release the lock by putting a new session in the empty cell, after which the cell becomes full again. This lock-releasing is done with (5), which puts a new session a, the behaviour of which is implemented by process Q_1, and then continues as Q_2. The take operation shifts a usage from full $\mathbf{U}_f\ A$ to empty $\mathbf{U}_e\ A$, whereas put does the converse.

Affine Stored State

In CLASS, besides linear and persistent, sessions can also be affine. Affine sessions are types by the modality $\wedge A$ and represent well-behaved discardable sessions (satisfy weakening), that, when discarded, safely discard all the sessions they hereditarily refer to, as in [6](akin to cascading deletes in relational databases). The use of affinity is pervasive in many substructural type systems [3, 150, 38, 33, 110, 28, 57], either to express general

resourceful programming or to model failures and exceptions, and it has also recently made its way into the practical world of programming with its adoption by Rust.

In CLASS, all the sessions stored by reference cells are affine. This is an essential restriction since reference cells must be released when no longer needed. And in order to release a reference cell, we must be able to safely discard the session that it stores, otherwise leaks or deadlocks may occur, hence affinity.

Furthermore, cell deallocation in CLASS is handled automatically through the interplay of a couple of simple process algebraic-like manipulations. This automatic memory management contrasts with [128], in which resources need to be explicitly discarded by programmers or even with [10], where memory management is left as an open problem.

A Stateful "Hello World"

To illustrate the concepts introduced so far, we will consider a basic programming example in CLASS of a reference cell storing a natural and two atomic imperative operations, one for incrementing and other for getting the natural stored in the cell.

Let us start with the following definition

$$\mathsf{init}(c) \triangleq \mathbf{cell}\ c(n.\mathbf{affine}\ n; \mathsf{V}_{42}(n))$$

This defines a process $\mathsf{init}(c)$ that behaves as a reference cell on session $c : \mathbf{S}_f\ \mathsf{Nat}$. The cell is initially storing the affine natural 42.

We will now define an increment cell usage operation

$$\begin{aligned}\mathsf{inc}(c) \triangleq\ & \mathbf{take}\ c(n);\\ & \mathbf{put}\ c(m.\quad \mathbf{affine}\ m;\\ & \qquad\qquad \mathbf{use}\ n;\\ & \qquad\qquad \mathsf{succ}(n, m));\\ & \mathbf{release}\ c\end{aligned}$$

Process $\mathsf{inc}(c)$ offers the dual behaviour $c : \mathbf{U}_f\ \overline{\mathsf{Nat}}$, and takes the affine natural n stored in the reference cell c. Then, it puts back the successor of n, after which it releases its cell usage.

We also define an operation that gets the natural stored in the reference cell

$$\begin{aligned}\mathsf{get}(c, x) \triangleq\ & \mathbf{take}\ c(n);\\ & \mathbf{par}\{\\ & \quad \mathbf{use}\ n;\\ & \quad \mathbf{send}\ x(n);\\ & \quad \mathbf{close}\ x\\ & \quad ||\\ & \quad \mathbf{put}\ c(m.\mathbf{affine}\ m; V_0(m));\\ & \quad \mathbf{release}\ c\\ & \quad \}\end{aligned}$$

Process $\mathsf{get}(c, x)$ offers the behaviours $c : \mathbf{U}_f\ \overline{\mathsf{Nat}}$ and $x : \mathsf{Nat} \otimes \mathbf{1}$. It takes the natural n stored in the reference cell c and outputs n in the session channel x, after which it closes x. This output-and-close sequence of interactions on x can, for example, represent the action of printing the natural in the console. In parallel, $\mathsf{get}(c, x)$ resets the cell to $\mathbf{0}$ and releases its cell usage.

Finally, we define two simple systems that composes, via cuts on session c, the reference cell and with the operations to increment and get

$$\mathsf{system}_1 \triangleq \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathsf{inc}(c)\} \qquad \mathsf{system}_2(x) \triangleq \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathsf{get}(c, x)\}$$

When evaluating system_1, the cell on c will end up storing the natural 43. On the other, $\mathsf{system}_2(x)$ reduces to a process in which the natural 42 is being transmitted on channel x, this time the cell c ends up storing the natural 0. Both threads $\mathsf{inc}(c)$ and $\mathsf{get}(c, x)$ release their usages c and, in this case, since the cell is not being shared by any other thread, this leads to cell deallocation in both systems.

Shared State

Until now, we have only considered sequential cell usage operations. But one of the advantages of having state is to be able to share it among cooperative threads. Sharing in CLASS is introduced by the operation

$$\mathbf{share}\ c\ \{P \mid\mid Q\}$$

in which threads P and Q are running concurrently and sharing reference cell usage c.

Sharing relates to DiLL cocontraction [53], this can be understood as a disciplined fork operation, in which P and Q may linearly interact at the shared reference c, but not on other shared sessions. This condition is related with the basic linear logic cut (cf. [23, 157], where two threads cannot interact on more than one channel. Cocontraction is essential to ensure deadlock-freedom by purely logical means.

Revisiting our stateful "hello world" we can now define the following system

$$\mathsf{system}_3 \quad \triangleq \quad \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathbf{share}\ c\ \{\mathsf{obs}(c, x) \mid\mid \mathsf{inc}(c) \mid\mid \mathsf{inc}(c)\}\}$$

in which the reference cell $\mathsf{init}(c)$ is composed with three concurrent threads, two of them are incrementing and there is one observing the stored natural.

Well-Behaved Cell Usages

The fine-grained type system of CLASS distinguishes between full and empty state at the type level, with the prefixed state/usage modalities $\mathbf{S}_f\ A/\mathbf{U}_f\ A$ and $\mathbf{S}_e\ A/\mathbf{U}_e\ A$. This has some pleasant consequences, among which allows us to rule out wrong cell usage scenarios statically. Add the basic acyclicity, granted by the linear logic foundations and we have the guarantee that stateful programs in CLASS are deadlock-free, purely by construction.

This fine-grained type level distinction contrasts with some typed systems, like for example typed Concurrent Haskell MVars [83, 102], in which both full and empty state are aggregated under a single type. As a consequence, programs that use MVars in Haskell can cause runtime errors aborting the whole computations if a put is ever attempted in a full MVar, whereas in CLASS these *wrong* cell usages are excluded at compile time.

Actually, this type level distinction between the full and empty states is also present in Rust [86] mutexes (see std::sync::Mutex). However, the linear logic based type system of CLASS exclude misbehaved guarantees at compile-time the absence of deadlocks, whereas in Rust, mutex-based programs can block.

Therefore, in CLASS, the take-put dynamics follows the usual safe pattern of mutex-protected objects in which each lock-acquire (take) must be followed by a lock-release (put). Furthermore, in CLASS, the lock and the data it protects are tightly associated since one must acquire the data with a take operation before any update. This contrasts with the mutex-based style of programming, e.g mutex synchronisation in POSIX threads API, in which one might forget to acquire the lock before updating, leading to data corruption.

Reference Cells are First-Class

In CLASS, reference cells are first-class objects. Therefore, they can be passed around communication channels or even being used to code pointed data structures such as linked lists and binary search trees. To illustrate the first point consider the following system in which we compose a reference cell with two communicating threads

$$\mathbf{cut}\ \{\mathbf{cell}\ c(a.V)\ |c|\ \mathbf{send}\ x(c); P\ \ |x|\ \ \mathbf{recv}\ x(c'); \mathbf{release}\ c' Q\}$$

The composition is done with an n-ary cut, which is left associative. In this example, a cell usage c is sent along channel x to the receiver, that frees it upon reception on the input parameter c'. Notice that the sending thread $\mathbf{send}\ x(c); P$ loses access to the reference cell c, it transfers ownership of c to the receiving thread. This is because reference usages are linear sessions, whose visibility may only be duplicated by the explicit share construct.

To illustrate the second point, we will show how recursion coupled with higher-order state allow us to program interesting linked data structures that support memory-efficient updates in-place. Consider the following mutual recursive pair of type definitions that express linked lists $\mathsf{LL}(A)$, which store affine sessions A (recall that in CLASS every value is represented as a session)

$$\begin{array}{lcl} \mathsf{LL}(A) & \triangleq & \mathbf{S}_f\ \mathsf{Node}(A) \\ \mathsf{Node}(A) & \triangleq & \oplus\{\mathsf{Null} : \mathbf{1}, \mathsf{Next} : \wedge A \otimes \mathsf{LL}(A)\} \end{array}$$

A process offering the recursive type $\mathsf{LL}(A)$ behaves as a cell which stores a node of session $\mathsf{Node}(A)$. A session of type $\mathsf{Node}(A)$ either chooses Null if the list is empty, in which case it closes; or chooses Next, in which case it sends an affine session $\wedge A$ representing the head element and recurs as the tail $\mathsf{LL}(A)$.

With some auxiliary processed, to be defined later, we can express a process

$$\textbf{cut}\ \{\textbf{cell}\ c_2(n_2.\mathsf{next}(a_2, c_1, n_2))\ \ |c_1|\ \ \textbf{cell}\ c_1(n_1.\mathsf{next}(a_1, c_0, n_1))\ |c_0|\ \textbf{cell}\ c_0(n_0.\mathsf{null}(n_0))\}$$

offering $c : \mathsf{LL}(A)$ and which codes a linked list with two elements a_2, a_1.

We also define memory-efficient update in-place operations, like operation

```
1 :  append(l1, n2, l) ≜
2 :   take l1(n);
3 :   use n;
4 :   case n {
5 :     |#Null :  wait n;
6 :               put l1(n'.fwd n2 n');
7 :               fwd l1 l
8 :     |#Next :  recv n(a);
9 :               cut {
10 :                append(n, n2, l')
11 :                 |l'|
12 :                put l1(n'.next(a, l', n'));
13 :                fwd l1 l}
14 :  }
```

that corecursively appends node n_2 to the tail of linked list l_1 and outputs the updated list in l. Process $\mathsf{append}(l_1, n_2, l)$ takes the node n stored in l_1, uses n and then performs case analysis (ln. 2-4). If Null, it waits for n to be closed and puts n_2 in the list l_1, after which l_1 is forwarded to l (ln.5-7). This corresponds to the base case in which list l_1 is empty.

The coinductive step, in which l_1 has at least one element is processed when n chooses Next. Then, it it receives the head element $a : \vee\overline{A}$ on session n (ln. 8), recursively appends n_2 to the back of the linked list $n : \overline{\mathsf{LL}(A)}$ (ln. 10) and puts on l_1 a node whose head element is a and whose tail is the result l' of the recursive call (ln. 12), after which it forwards the updated usage l_1 to l (ln. 13).

These linked data structures are then the building blocks of further complex programs in CLASS. In Example 4.3, they are used to solve famous resource synchronisation problems like the dining philosophers [51], in which the order the order on the locks is ingeniously represented by a passive linked chain. Deadlock-freedom follows by the basic acyclicity of the linear logic, without needing to resort to extra-logical partial orders.

They are also used to program more complex mutable shareable ADTs such as the imperative queue 4.2, which is based on a linked list and two independent pointers that allows for concurrent enqueueing and dequeueing in $O(1)$ time. Remarkably, even in the presence of first-class reference cells and recursion, CLASS type system resiliently precludes circularities in memory (such as the Landin's knot). In fact, as we will show, every well-typed program in CLASS terminates.

Nondeterminism and Algebraic Reasoning

Sharing of stateful reference cells leads to nondeterminism, which naturally emerge from racy concurrent behaviour, specifically from concurrent take operations. In CLASS this nondeterminism is captured by a non-collapsing sum operation

$$P + Q$$

which represents a choice between two alternatives P and Q. Sums are also present in DiLL [53], where cut elimination needs to generate sums of proofs.

In CLASS, they satisfy the expected axioms of nondeterministic sums of process algebras [67], like commutativity, associativity, idempotency and the basic interleaving law that connects shared state with nondeterminism. Interestingly, in CLASS these laws also have an interpretation at the proof level as typed conversions.

Sums play a key role in the metatheoretical model of CLASS as they allows us to explicitly capture all the possible nondeterministic evolutions of a stateful system and, because of that, to have a confluent notion of reduction. Remarkably, with sums we can reason the behaviour of concurrent stateful programs equationally by doing simple algebraic-like manipulations.

For example, consider the process

$$\mathsf{system}_3 \triangleq \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathbf{share}\ c\ \{\mathsf{inc}(c)\ ||\ \mathsf{obs}(c, x)\}\}$$

which composes the previously defined reference cell $\mathsf{init}(c)$, initially storing the natural 42, with the concurrent threads $\mathsf{inc}(c)$ and $\mathsf{obs}(c, x)$ that, respectively, increment and observe. Observation is done by communicating on session x the stored natural in the cell c. The value to be observed on x will depend on the nondeterministic scheduling of the two concurrent atomic actions. This will be either 42, if the observation happens before the increment, or 43, otherwise.

In CLASS, we can rewrite system_3 in the simplified equivalent form as a sum processes

$$\mathbf{cut}\ \{V_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\} \quad + \quad \mathbf{cut}\ \{V_{43}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}$$

which summarises the behaviour of the initial system. Notice that each summand is pure process that does not use imperative constructs, even though the original system_3 internally manipulates shared state. As we shall see, this is not a coincidence, but a particular consequence of a cut normalisation result that we establish for CLASS.

A Simple Fork-Join

We conclude this overview with the implementation of a simple fork-join synchronisation method in which a process waits for a number of concurrent threads to finish their work, inspired in the Rust reference page for std::sync::Mutex. The fork-join FJ representation datatype is encoded as a reference cell of two components

$$\mathsf{FJ} \triangleq \mathbf{S}_f\ (!\mathsf{Nat} \otimes \wedge\bot)$$

which are grouped with a tensor ⊗. The first component, encoded by a persistent natural !Nat, counts the number of threads that have already finished their work. The second, encoded by an affine bottom ∧⊥, represents the session used to signal the waiting thread when all the threads have finished their job.

Code for each working thread is defined on the left

```
1 :  thread(s, max) ≜
2 :   take s(x);
3 :   use x;
4 :   recv x(n);
5 :   ?n;
6 :   if (n == max){
7 :   par {
8 :     use x;
9 :     close x
10 :    ||
11 :    put s(y.affine y;
12 :          send y(m.let! m 0);
13 :          wait y;0);
14 :    release s
15 :    }
16 :  }{
17 :    put s(y.affine y;
18 :          send y(m.let! m n + 1);
19 :          fwd x y);
20 :    release s}
```

```
1 :  main() ≜
2 :  cut{
3 :   affine x;
4 :   wait x;
5 :   P
6 :   |x|
7 :   cell s(y.affine y;
8 :            send y(n.let! n 1);
9 :            fwd y x)
10 :  |s|
11 :  let! max 5
12 :  |max|
13 :  ?max;
14 :  for  spre : FJ  to spost(s, s′, max){
15 :    share spre {thread(spre, max)
16 :                ||
17 :                fwd spre spost}}
18 :  |s′|
19 :  release s′
20 :  }
```

Process thread(s, max) takes the two components x stored in the cell s : FJ, uses x, receives on x the current counter n and compares it with a maximum value max (ln. 2-6). If the maximum value is reached, it signals the waiting thread by using and then closing the session x (ln. 8-9). In parallel, it resets s to 0 and puts a dummy waiting session y (ln. 11-13). If, on the other hand, the maximum value was not reached yet, we simply increment the counter and put back the session x, used to signal the waiting thread (ln. 17-19). Both if and else clauses finish by releasing s (ln. 14 and 20)

On the right, we define main() that composes, with several cuts (recall n-ary cut is left associative), the following processes:

- the waiting thread on x that, after waiting, continues as P (ln. 3-4);
- a reference cell s : FJ, that starts with counter on 1 and stores session x to signal the waiting thread (ln. 7-9);
- the parameter max, set to 5 (ln. 11);

- a for-loop that spawns a number max working threads, starting on s to yield s' (ln. 15-17);
- an operation that releases usage s', after it being processed by the working threads (ln. 9).

For the sake of presentation, we have used some idiomatic expressions like ML-like let constructs, if-conditionals and for-loops, both of which can be implemented using pure sessions in the pure fragment μCLL of CLASS. In particular, the for-loop can be compiled into primitive recursion and iterates max times the loop body, that yields spost from spre. The loop body **share** spre {...} essentially launches a new parallel thread with shared access to the counter, and forwards a state alias for the next iteration, so that an arbitrary number max of shared threads is created.

Notice that in CLASS the cell sharing topologies are dynamic since users can come and go and, furthermore, they can grown potentially unbounded, as in this example, in which an arbitrary number of shared threads is created within a for-loop. This contrasts with some session-typed languages (e.g., [91]) in which the number of participants sharing a session is statically bounded by the type system.

Interestingly, the waiting thread is wrapped as an affine on session x since it can be the case that session x is never used and closed, in which the case x has to be safely discarded.

In this implementation, clients can tamper with the representation of the fork-join datatype but we will see later how to hide the implementation details with existential types [109] and force the clients to manipulate the stateful objects with a provided interface.

The thesis discusses many more interesting examples, a shareable linked list with an update in-place append operation, a concurrent queue, and also discuss how to express and type resource synchronisation methods such as barriers and even a solution to the famous dining philosophers problem. For all these programs, strong guarantees including memory safety, deadlock-freedom, termination, and general absence of "dynamic bugs", which survive even in the presence of our blocking primitives and higher-order state, are automatically and compositionally implied by our type discipline based on propositions-as-types and linear logic, which imposes crisp yet expressive acyclicity conditions on channel communication topology and shared linked data structures. In the next section, we present our contribution and outline the thesis structure.

1.5 Contributions and Outline

This thesis has the following original contributions

Linear Logical Foundations for Shared State. We build on the propositions-as-types approach to session-based concurrency [23, 24, 157] by defining CLASS: an extension of classical linear logic with shared affine state. The system fully complies with the propositions-as-types approach (proofs as programs, formulas as types, evaluation

as proof simplification). The typing rules for the imperative fragment are inspired by those for the exponentials and sum connectives of differential linear logic (DiLL) [54, 53]. In particular, we originally interpret the logical principle of cocontraction from DiLL as state sharing, which is a key contribution of this thesis, and relate nondeterministic sums with concurrency, via the interleaving law of process algebras, which in our framework naturally appears as a typed conversion.

Language Expressiveness and Orthogonality. We showcase the expressiveness of language CLASS by coding several examples, ranging from basic datatypes such as booleans and naturals to stateful memory-efficient linked data structures, such as linked lists and binary search trees, up to mutable shareable concurrent ADTs such as counters, stacks and queues. We show that it is even possible to code resource synchronisation methods in CLASS (fork-joins, barriers, dining philosophers), where deadlock-freedom follows just by the linear logic based type system. Remarkably, everything is coded by relying on pure linear logical sessions.

These examples also show how orthogonal type features combine well with the imperative stateful framework. Namely, standard existential type quantifiers [35, 109] harmoniously combine with the basic stateful framework of CLASS in order to define stateful ADTs, while the presence of basic inductive/coinductive session types allows us to define linked data structures.

Metatheory. Building on top of a logic edifice pays off. We show that CLASS enjoys several metatheoretical properties: deadlock-freedom, confluence, cut normalisation and termination. Furthermore, because of our propositions-as-types approach we can reason equationally about the behaviour of concurrent stateful programs. These results follow from the correspondence with linear logic and are fully detailed in this thesis. They adapt previous methods so as to accommodate shared mutable state.

Type Checker and Interpreter. We show the feasibility of our propositions-as-types approach by implementing a type checker and interpreter in Java for CLASS. The implementation is used to validate many examples such as inductive datatypes (naturals, lists) using system F style encodings, linked data structures (linked lists, binary search trees), and shareable concurrent mutable ADTs (bank accounts, stacks, functional and imperative queues). We also developed a test suite.

An article was published in Proc. ACM Program. Lang. 5, ICFP 2021 [134], in which we developed a concurrent session-typed language πSSL with first-class reference cells and locks, based on a propositions-as-types correspondence with classical linear logic. The publication is accompanied of a type checker and interpreter implementation, approved as an artifact [131] and there is also a technical report [132] with complete definitions and detailed proofs.

Language CLASS, presented in this thesis, is a natural outgrowth of πSSL, but results from a more fundamental and expressive approach. In [133], the reader may find an extended draft presenting CLASS together with a type checker and interpreter implementation with a creative commons license. The implementation comes with detailed use instructions (*README.md*), the source code (folder *src/*) and an extensive (~ 6k loc) collection of examples (folder *examples/*). We refer to these examples throughout the thesis.

The thesis is split into two parts. In the first part we present the session-typed concurrent language CLASS, showcase its expressiveness and present its practical implementation. More specifically:

Chapter 2. We present the session-typed process calculus μCLL which is related, via propositions-as-types correspondence [23, 24, 157], to second-order classical linear logic, here extended with mix and inductive/coinductive types. We also give a tutorial-like presentation of μCLL going though each of the process constructs, associated typing rules and operational semantics.

Chapter 3. We present CLASS, which extends μCLL with first-class stateful reference cells, sharing and nondeterminism. We defines its type and process syntax, the type system and the reduction-based operational semantics. We give a tutorial-like presentation of CLASS, interspersed with several examples that illustrate the main concepts.

Chapter 4. we exhibit the expressive power of CLASS language and type system, by coding several realistic examples: general corecursive protocols with some resource invariant, a linked list, a solution to the dining philosophers problem and a concurrent imperative queue.

Then, in the second, part we present the language metatheoretical results:

Chapter 6. We show that our stateful language CLASS satisfies the key safety properties of type preservation (Theorem 1) and progress (Theorem 2), which in the context of session-based concurrency entail session-fidelity and deadlock-freedom. Progress follows naturally from the acyclicity inherent to linear logic.

Chapter 7. We establish that our system CLASS, internalising nondeterminism with sum processes, enjoys confluence (Theorem 3). The proof relies on the Tait and Martin-Löf technique and shows that proof reductions and conversions represent proof identities or behavioural equivalences. This also reveals a new connection between logic interpretations of programming languages, concurrency and nondeterminism. We prove an interesting auxiliary result that shows that each process can be written in an equivalent form by interleaving all the concurrent take operations and by distributing sums.

Chapter 8. We establish a cut normalisation result for open processes in a sublanguage of CLASS without second-order quantifiers and without inductive/coinductive session types (Theorem 4) by adapting Pfenning's structural cut elimination technique for classical linear logic [122, 123].Then, it derives some corollaries: the Subformula Property Corollary 1 and a Cut Elimination for Pure Sequents Corollary 2. The latter implies that any process that potentially uses shared state internally, but that offers a pure typing interface, normalises to a cut-free process that does not use imperative constructs at all (Corollary 2).

Chapter 9. We prove strong normalisation (Theorem 5) for well-typed closed CLASS programs, a challenging result in this setting, which strengthens the propositions-as-types interpretation of our language. Our proof scales linear logical relations [121, 63, 9] to accommodate shared mutable state and relies on new techniques to handle stateful computation and interference.

Finally, Chapter 10 offer some concluding remarks and present further research directions. Each chapter, with the exception of Chapter 4 and Chapter 10, is concluded with a section in which we present further discussion and related work. Detailed proofs are presented in appendices A-E.

Part I

The Concurrent Programming Language CLASS

2

The Basic Language μCLL

2.1 Introduction

In this chapter we present the pure fragment of CLASS, dubbed μCLL. Sublanguage μCLL is related, via a propositions-as-types correspondence [23, 24, 157], to second-order classical linear logic, here extended with mix and inductive/coinductive types.

As typical of session-typed based programming languages [73, 74], in μCLL processes communicate trough sessions $x, y, z, \ldots$. A session has two endpoints, each with an associated protocol described by a session type A, where we write $x : A$ to indicate that the session endpoint x obeys type A. Session types describe the direction, the order and the content of the messages exchanged. For example

$$x : \mathsf{Nat} \multimap \mathsf{Bool} \otimes \mathbf{1}$$

characterises an endpoint of a session x that inputs a natural number, then outputs a boolean and finally closes, after which no interaction can possibly occur. Furthermore, session types are dynamic entities since they evolve as computation takes place. In the referred example, after the natural being received, the protocol of the session changes to

$$x : \mathsf{Bool} \otimes \mathbf{1}$$

Session types are structured by an involutive operation of duality $A \mapsto \overline{A}$, where A and $\overline{A}$ are dual types, that corresponds to linear logic negation and captures symmetry in interaction: when one process sends, the other receives, when offers a menu, the other chooses from that menu, and so on.

In μCLL, the basic fundamental operation to compose processes is expressed by the cut construct

$$\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$$

that composes two processes P and Q offering duality-related behaviours on each session endpoint x. In a cut $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$, processes P and Q run concurrently communicating through a single private session x. This interactive composition corresponds to the cut rule of linear logic. Cut restricts interactions to a single private session, this is essential

Table 2.1: Types and actions of μCLL, together with their interpretation.

Type	Corresponding Action(s)	Interpretation
$\mathbf{1}$	**close** x	Close x.
$\bot$	**wait** $x; Q$	Wait on x, continue as Q.
$A \,\&\, B$	**case** x $\{\vert\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}$	Case on x: left and continue as P_1; or right and continue P_2.
$A \oplus B$	$x.\mathbf{inl}; Q_1$	Choose right on x, continue as Q_1.
	$x.\mathbf{inr}; Q_2$	Choose left on x, continue as Q_2.
$A \otimes B$	**send** $x(y.P_1); P_2$	Send y on x, continue as P_2.
$A \parr B$	**recv** $x(y); Q$	Receive y on x, continue as Q.
$!A$	$!x(y); P$	Replicated session on x with parameter y.
$?A$	$?x; Q$	Make x unrestricted, continue as P.
	call $x(y); Q$	Call x with input y, continue as P.
$\exists X.A$	**sendty** x $B; P$	Send on x type B, continue as P.
$\forall X.A$	**recvty** $x(X); Q$	Receive on x type X, continue as Q.
μXA	**unfold**$_\mu$ $x; P$	Unfold x, continue as P.
νXA	**unfold**$_\nu$ $x; Q$	Unfold x, continue as Q.
	corec $X(z, \vec{w}); Q$ $[x, \vec{y}]$	Corecursive definition with body Q.

to guarantees acyclicity on the process communication topologies, which then implies desirable metatheoretical properties such as the absence of deadlocks.

Another way of composing processes is given by **par** $\{P \parallel Q\}$ where P and Q run in parallel without ever interfering with each other. This corresponds to linear logic mix rule. μCLL also has the bidirectional forwarding construct **fwd** x y that computationally acts as a link between two dual-typed endpoints x and y, forwarding all the interactions with x to y and vice-versa. Forwarding corresponds to the identity axiom of linear logic.

Table 2.1 lists the types A of μCLL, together with the corresponding actions on a session endpoint $x : A$ and their interpretation. The multiplicative units $\mathbf{1}/\bot$ type session termination, the additives $A \,\&\, B/\overline{A} \oplus \overline{B}$ type menu offer and selection, the multiplicatives $A \otimes B/\overline{A} \parr \overline{B}$ type session communication. Then, the exponentials $!A/?\overline{A}$ type replicated sessions and their invocation by clients. The existential and universal type-quantifiers $\exists X.A$ $/\forall X.\overline{A}$ are interpreted by processes that respectively send and receive types and play a key role for defining abstract data types (ADTs). The inductive/coinductive session types $\mu XA/\nu X\{\overline{X}/X\}\overline{A}$ type recursive/corecursive process.

As typical of process calculi [15, 107, 75], μCLL has a reduction-based operational semantics, which is defined by specifying a structural congruence relation $\equiv$ that captures the static laws, essentially rearranging processes, and a dynamic reduction relation $\longrightarrow$ that captures the evolution of a process as interaction takes place. For example, $\equiv$ rule [**0** M]

$$\mathbf{par}\ \{P \parallel \mathbf{0}\} \equiv P$$

expresses that inaction **0** is a unit of the binary mix construct, whereas rule [CM]

$$\mathbf{cut}\ \{P\ \ \vert x : A\vert\ \ \mathbf{par}\ \{Q \parallel R\}\} \equiv \mathbf{par}\ \{\mathbf{cut}\ \{P\ \ \vert x : A\vert\ \ Q\} \parallel R\}$$

linearly distributes a cut over a mix. For example, the reduction $\rightarrow$ rule [$\mathbf{1}\bot$]

$$\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\} \rightarrow P$$

models the interaction between a close and a wait: after the interaction the cut evolves to the continuation P of the waiting process. Rules of $\equiv$ and $\rightarrow$ are connected to the commuting and principal cut conversions of classical linear logic [23, 157].

As we shall see, μCLL is able to express general polymorphic higher-order concurrent and functional programs, and compare in expressiveness with System F [149, 66]. We will see, in particular, how to code basic datatypes such as booleans and naturals and shared boolean calculators. Since everything is implemented by using pure linear logic based sessions, by construction processes are guaranteed to be deadlock-free and always terminate. However, in μCLL computation is essentially *functional*: there are no side effects, no races and shared objects are stateless. In the next chapter we will explore how to lift these restrictions to capture more interesting computational scenarios involving shared state, without messing up with the linear logic foundations.

This chapter is organised as follows. Section 2.2 defines the process calculus μCLL and its operational semantics. Then, in Section 2.3, we present the type system going though each of the process constructs, associated typing and reduction rules. The introduced concepts are illustrated with several examples. Finally, Section 2.4 concludes with further discussion and related work.

2.2 Process Calculus and Operational Semantics

In this section, we define the process syntax and the reduction-based operational semantics of μCLL.

Types and Duality

Since in μCLL processes depend on types, we start by presenting types and defining type duality.

Definition 1 (Types)**.** *The types of* μCLL *are defined by*

$$\begin{array}{rlll} A, B ::= & X \text{ (type variable)} & | & \overline{X} \text{ (dual of type variable)} \\ | & \mathbf{1} \text{ (one)} & | & \bot \text{ (bottom)} \\ | & A \otimes B \text{ (tensor)} & | & A \parr B \text{ (par)} \\ | & A \oplus B \text{ (plus)} & | & A \,\&\, B \text{ (with)} \\ | & !A \text{ (bang)} & | & ?A \text{ (why not)} \\ | & \exists X.A \text{ (exists)} & | & \forall X.A \text{ (for all)} \\ | & \mu X.\ A \text{ (mu)} & | & \nu X.\ A \text{ (nu)} \end{array}$$

Types are composed from type variables, units ($\mathbf{1}$, $\bot$), multiplicatives ($\otimes$, $\parr$), additives ($\oplus$, $\&$), exponentials (!, ?), second-order type quantifies ($\exists$, $\forall$) and inductive types (μ, ν). The expressions $\exists X.A$, $\forall X.A$, $\mu X.\ A$, $\nu X.\ A$ all bind the type variable X in A. The expression $\{A/X\}B$ denotes capture-avoiding substitution of type variable X by A in B.

We consider that the binary type connectives associate to the right, therefore the type $A \otimes B \parr C$ should be parsed as $A \otimes (B \parr C)$. Furthermore, we consider that the unary operators !, ?, $\exists X$, $\forall X$, μX and νX have higher precedence that the binary connectives. Therefore, the type $!A \otimes B$ should be parsed as $(!A) \otimes B$.

Types are structured by an involutive operation $A \mapsto \overline{A}$ of duality that corresponds to linear logic negation and captures symmetry in interaction.

Definition 2 (Duality on Types $\overline{A}$)**.** *Duality $\overline{A}$ is the involution on types defined by*

$$\begin{array}{llllllll} \overline{\mathbf{1}} & \triangleq & \bot & \overline{A \otimes B} & \triangleq & \overline{A} \parr \overline{B} & \overline{A \oplus B} & \triangleq & \overline{A} \ \& \ \overline{B} \\ \overline{!A} & \triangleq & ?\overline{B} & \overline{\exists X.A} & \triangleq & \forall X.\overline{A} & \overline{\mu X.\ A} & \triangleq & \nu X.\ \{\overline{X}/X\}(\overline{A}) \end{array}$$

For convenience, we define the lollipop type constructor by $A \multimap B \triangleq \overline{A} \parr B$, using duality and par ($\parr$), which types session input.

Type-level duality is explicitly defined for half of the type connectives ($\mathbf{1}$, $\otimes$, $\oplus$, ! and $\exists$, μ). Since it is an involution we can obtain the explicit definition for the other half ($\bot$, $\parr$, $\&$, ?, $\forall$, ν). For example, for the connective tensor $\otimes$, Definition 2 postulates

$$\overline{A \otimes B} \triangleq \overline{A} \parr \overline{B}$$

By instantiating the metavariables A and B with $\overline{C}$ and $\overline{D}$, respectively, yields

$$\overline{\overline{C} \otimes \overline{B}} = \overline{\overline{C}} \parr \overline{\overline{B}}$$

By applying duality to both sides of the equation and by using the involution property $\overline{\overline{X}} = X$ to simplify the resulting expression, we obtain the explicit expression that gives duality for the connective par $\parr$:

$$\overline{C} \otimes \overline{B} = \overline{C \parr B}$$

Duality preserves type substitution $\overline{\{A/X\}B} = \{A/X\}\overline{B}$. As we shall see, duality enforces a pleasant symmetry on process interaction, captured at the proof level by a series of principal cut reductions.

Processes

We will now define processes of μCLL.

Definition 3 (Processes)**.** *The syntax of process terms for μCLL is defined in Fig. 2.1.*

$$
\begin{array}{lcl}
x, y, z, \ldots & \in & \mathcal{N} \text{ (session names)} \\
P, Q & ::= & \mathbf{0} \text{ (inaction)} \\
& & \mid \mathbf{fwd}\ x\ y \text{ (forwarder)} \\
& & \mid \mathcal{A} \text{ (action)} \\
& & \mid X(x, \vec{y}) \text{ (variable)} \\
& & \mid \mathbf{par}\ \{P \parallel Q\} \text{ (mix)} \\
& & \mid \mathbf{cut}\ \{P \ \ |x : A|\ \ Q\} \text{ (linear cut)} \\
& & \mid \mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \text{ (unrestricted cut)} \\
\mathcal{A}, \mathcal{B} & ::= & \mathbf{close}\ x \text{ (close)} \\
& & \mid \mathbf{wait}\ x; P \text{ (wait)} \\
& & \mid x.\mathbf{inl}; P \text{ (choose left)} \\
& & \mid x.\mathbf{inr}; P \text{ (choose right)} \\
& & \mid \mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\} \text{ (offer)} \\
& & \mid \mathbf{send}\ x(y.P); Q \text{ (send)} \\
& & \mid \mathbf{recv}\ x(y); P \text{ (receive)} \\
& & \mid !x(y); P \text{ (replication)} \\
& & \mid ?x; P \text{ (unrestrict)} \\
& & \mid \mathbf{call}\ x(y); P \text{ (call)} \\
& & \mid \mathbf{sendty}\ x\ A; P \text{ (send type)} \\
& & \mid \mathbf{recvty}\ x(X); P \text{ (receive type)} \\
& & \mid \mathbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}] \text{ (corecursion} \\
& & \mid \mathbf{unfold}_{\mu}\ x; P \text{ (unfold } \mu) \\
& & \mid \mathbf{unfold}_{\nu}\ x; P \text{ (unfold } \nu)
\end{array}
$$

Figure 2.1: Processes P of μCLL (extended by Fig. 3.1).

Processes depend on session names $x, y, z, \ldots$ and on variables $X(x, \vec{y})$. The static part of the syntax comprises inaction, mix, cut and cut!; the dynamic part includes actions $\mathcal{A}, \mathcal{B}$, and forwarder. An action is typically a process $\alpha; P$, where α is an action-prefix and P is the continuation. In these cases, the subject $s(\mathcal{A})$ of an action $\mathcal{A}$ is the leftmost name occurrence of $\mathcal{A}$. For example, the subject of the action $\mathbf{send}\ x(y.P); Q$ is x. The subject of $\mathbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}]$ is x.

For convenience, we equip μCLL with n-ary cut which is right-associative, encoded using binary cuts as:

$$\mathbf{cut}\ \{P_1\ |x_1 : A_1|\ \ldots\ P_{n-1}\ |x_{n-1} : A_{n-1}|\ P_n\} \triangleq \mathbf{cut}\ \{P_1\ \ |x_1 : A_1|\ \ \ldots \mathbf{cut}\ \{P_{n-1}\ \ |x_{n-1} : A_{n-1}|\ \ P_n\}\}$$

Similarly, we also equip μCLL with n-ary mix

$$\textbf{par}\ \{P_1 \mid\mid \ \ldots\ P_{n-1} \mid\mid P_n\}$$

The expression $\textbf{cut}\ \{P \mid x : A \mid Q\}$ binds the name x on processes P and Q. $\textbf{cut!}\ \{y.P \mid x : A \mid Q\}$ binds y in P and x in Q. Actions $\textbf{send}\ x(y.P); Q$, $\textbf{recv}\ x(y); P$, $!x(y); P$, $\textbf{call}\ x(y); P$ bind y on P. Action $\textbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}]$ binds names $z, \vec{w}$ in P. All other name occurrences are free. The set of free names of P is denoted by $\mathsf{fn}(P)$; if $\mathsf{fn}(P) = \emptyset$, we say P is closed.

The expressions $\textbf{recvty}\ x(X); P$ binds the type variable X on process P. All the other type variable occurrences are free. Capture-avoiding substitution and α-conversion are defined as usual. We denote by $\{x/y\}P$ the process obtained by replacing the name y by x on P. Similarly, we denote by $\{A/X\}P$ the process term obtained by replacing type variable X by type expression A in process term P. If $\vec{x}$ and $\vec{y}$ are arrays of names with the same length n we let $\{\vec{x}/\vec{y}\}P$ denote the substitution $\{\vec{x}[0]/\vec{y}[0]\}(\ldots\{\vec{x}[n]/\vec{y}[n]\}P)$.

Operational Semantics

We will now present the reduction-based operational of μCLL , which is defined by specifying a structural congruence relation $\equiv$ that captures the static laws, essentially rearranging processes, and a a dynamic reduction relation $\longrightarrow$ that captures the evolution of a process as interaction takes place.

Before defining structural congruence, we need to introduce process contexts. A process context C is a process expression containing a single hole $\Box$ [137]. For example, the following

$$\Box \qquad \textbf{cut}\ \{\Box \mid x : A \mid P\} \qquad \textbf{wait}\ y; \textbf{par}\ \{\textbf{cut}\ \{Q \mid x : A \mid \Box\} \mid\mid P\}$$

are both process contexts. We write $C[P]$ for the process obtained by replacing the hole in C by P (notice that in $C[P]$ the context C may bind free names of process P). Similarly, given two process contexts C_1, C_2, we write$C_1[C_2]$ for the context obtained by replacing the hole in C_1 by C_2. We define context composition by $C_1 \circ C_2 C_1[C_2]$. A process P' is a subprocess of P if $P = C[P']$, for some process context C. We say that a relation $\mathcal{R}$ is a process congruence iff whenever $P\mathcal{R}Q$, then $C[P]\mathcal{R}C[Q]$.

Definition 4 (Structural Congruence $P \equiv Q$)**.** *Structural congruence $\equiv$ is the least relation on processes that includes α-conversion and the rules in Fig. 2.2.*

The basic rules of $\equiv$ essentially reflect the expected static laws, along the lines of the structural congruences / conversions in [23, 157]. Relation $\equiv$ is an equivalence relation (rules [refl], [symm] and [trans]) which satisfies the congruence rule [cong]. The binary operators cut, forwarder and mix are commutative (rules [C], [fwd], [M]). Furthermore, mix is associative (rule [MM]) and has the inaction process $\mathbf{0}$ as identity (rule [0M]). We can linearly distribute the linear and the unrestricted cuts, where the distributions are

Congruence Rules

$P \equiv P$ [refl]

$P \equiv Q \supset Q \equiv P$ [symm]

$P \equiv Q$ and $Q \equiv R \supset P \equiv R$ [trans]

$P \equiv Q \supset C[P] \equiv C[Q]$ [cong]

Commutativity Rules

$\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \equiv \mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\}$ [C]

$\mathbf{fwd}\ x\ y \equiv \mathbf{fwd}\ y\ x$ [fwd]

Mix Rules

$\mathbf{par}\ \{P \parallel \mathbf{0}\} \equiv P$ [0M]

$\mathbf{par}\ \{P \parallel Q\} \equiv \mathbf{par}\ \{Q \parallel P\}$ [M]

$\mathbf{par}\ \{P \parallel \mathbf{par}\ \{Q \parallel R\}\} \equiv \mathbf{par}\ \{\mathbf{par}\ \{P \parallel Q\} \parallel R\}$ [MM]

Linearly Distributive Conversions

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{par}\ \{Q \parallel R\}\} \equiv \mathbf{par}\ \{\mathbf{cut}\ \{P\ \ |x|\ \ Q\} \parallel R\},\ x \in \mathrm{fn}Q$ [CM]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{cut}\ \{Q\ \ |y|\ \ R\}\} \equiv \mathbf{cut}\ \{\mathbf{cut}\ \{P\ \ |x|\ \ Q\}\ \ |y|\ \ R\},\ x, y \in \mathrm{fn}Q$ [CC]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{cut!}\ \{y.Q\ |z|\ R\}\} \equiv \mathbf{cut!}\ \{y.Q\ |z|\ \mathbf{cut}\ \{P\ \ |x|\ \ R\}\},\ z \notin \mathrm{fn}P$ [CC!]

$\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{par}\ \{Q \parallel R\}\} \equiv \mathbf{par}\ \{\mathbf{cut!}\ \{y.P\ |x|\ Q\} \parallel R\}, x \notin \mathrm{fn}R$ [C!M]

$\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{cut!}\ \{w.Q\ |z|\ R\}\}$
$\equiv \mathbf{cut!}\ \{w.Q\ |z|\ \mathbf{cut!}\ \{y.P\ |x|\ R\}\}, x \notin \mathrm{fn}Q, z \notin \mathrm{fn}P$ [C!C!]

Unrestricted Cut Distributive Conversions

$\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{par}\ \{Q \parallel R\}\} \equiv \mathbf{par}\ \{\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \parallel \mathbf{cut!}\ \{y.P\ |x : A|\ R\}\}$ [D-C!M]

$\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{cut}\ \{Q\ \ |z|\ \ R\}\}$
$\equiv \mathbf{cut}\ \{\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}\ \ |z|\ \ \mathbf{cut!}\ \{y.P\ |x : A|\ R\}\}$ [D-C!C]

$\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{cut!}\ \{w.Q\ |z|\ R\}\}$
$\equiv \mathbf{cut!}\ \{w.\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}\ |z|\ \mathbf{cut!}\ \{y.P\ |x : A|\ R\}\}$ [D-C!C!]

Figure 2.2: Structural congruence $P \equiv Q$ rules of μCLL (extended by Fig. 3.2).

guided by the provisos (rules [CM], [CC], [CC!], [C!M], [C!C!]). Finally, we can distribute the unrestricted cut over the static constructs mix, cut and unrestricted cut as expressed by rules [D-C!M], [D-C!C] and [D-C!C!].

Before defining reduction, we introduce static contexts, which are defined by

$$C ::= \Box \mid \mathbf{par}\ \{C \parallel P\} \mid \mathbf{par}\ \{P \parallel C\} \mid \mathbf{cut}\ \{C\ \ |x|\ \ P\} \mid \mathbf{cut}\ \{P\ \ |x|\ \ C\} \mid \mathbf{cut!}\ \{y.P\ |x|\ C\}$$

A static context is therefore a context where the hole is neither guarded by any action nor lies in the server body P of a cut! $\mathbf{cut!}\ \{y.P\ |x|\ Q\}$.

$$\mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |y|\ \ P\} \longrightarrow \{x/y\}P \qquad [\text{fwd}]$$

$$\mathbf{cut}\ \{\mathbf{close}\ x\ \ |x|\ \ \mathbf{wait}\ x; P\} \longrightarrow P \qquad [\mathbf{1}\bot]$$

$$\mathbf{cut}\ \{\mathbf{send}\ x(y.P); Q\ \ |x|\ \ \mathbf{recv}\ x(z); R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x|\ \ \mathbf{cut}\ \{P\ \ |y|\ \ \{y/z\}R\}\} \qquad [\otimes\parr]$$

$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x|\ \ x.\mathbf{inl}; R\} \longrightarrow \mathbf{cut}\ \{P\ \ |x|\ \ R\} \qquad [\&\oplus_l]$$

$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x|\ \ x.\mathbf{inr}; R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x|\ \ R\} \qquad [\&\oplus_r]$$

$$\mathbf{cut}\ \{!x(y); P\ \ |x|\ \ ?x; Q\} \longrightarrow \mathbf{cut!}\ \{y.P\ |x|\ Q\} \qquad [!?]$$

$$\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{call}\ x(z); Q\} \longrightarrow \mathbf{cut}\ \{\{z/y\}P\ \ |z|\ \ \mathbf{cut!}\ \{y.P\ |x|\ Q\}\} \qquad [\text{call}]$$

$$\mathbf{cut}\ \{\mathbf{sendty}\ x\ A; P\ \ |x|\ \ \mathbf{recvty}\ x(X); Q\} \longrightarrow \mathbf{cut}\ \{P\ \ |x|\ \ \{A/X\}Q\} \qquad [\exists\forall]$$

$$\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x; P\ \ |x|\ \ \mathbf{unfold}_\nu\ x; Q\} \longrightarrow \mathbf{cut}\ \{P\ \ |x|\ \ Q\} \qquad [\mu\nu]$$

$$\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x; P\ \ |x|\ \ \mathbf{corec}\ Y(z, \vec{w}); Q\ [x, \vec{y}]\} \longrightarrow \mathbf{cut}\ \{P\ \ |x|\ \ \{x/z\}\{\vec{y}/\vec{w}\}\{\mathbf{corec}\ Y(z, \vec{w}); Q/Y\}Q\} \qquad [\text{corec}]$$

$$P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q\ \supset\ P \longrightarrow Q \qquad [\equiv]$$

$$P \longrightarrow Q\ \supset\ C[P] \longrightarrow C[Q] \qquad [\text{cong}]$$

Figure 2.3: Reduction $P \longrightarrow Q$ rules of μCLL (extended by Fig. 3.3).

We need also to define substitution of a process variable by a corecursive process, which will be used when modelling the one-step unfold of a corecursive process definition. The base cases are defined by

$$\begin{aligned} \{\mathbf{corec}\ X(z, \vec{w}); P/X\}X(x, \vec{y}) &\triangleq \mathbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}] \\ \{\mathbf{corec}\ X(z, \vec{w}); P/X\}Y(x, \vec{y}) &\triangleq Y(x, \vec{y}),\ Y \neq X \end{aligned}$$

and the substitution is propagated without surprises to the remaining cases.

Definition 5 (Reduction $P \longrightarrow Q$)**.** *Reduction $\longrightarrow$ is the least relation on processes that includes the rules in Fig. 2.3. N.B.: In [cong], C is an arbitrary static context.*

Reduction includes a set of principal cut conversions, it is closed by structural congruence ([≡]) and defined on the nose, so in rule [cong] we consider that C is a static context. Operationally, the forwarding behaviour is implemented by name substitution [30] ([fwd]). All the other conversions apply to a cut between two dual actions. For example, rule [$\otimes\parr$] applies to a cut on session $x : A \otimes B$ between send and receive and reduces to a process expression with two cuts. The inner cut on $y : A$ connects the continuation $\{y/z\}R$ of the receiver with the provider P of the sent channel, whereas the outer cut on $x : B$ connects $\{y/z\}R$ with the continuation Q of the send process.The principal cut conversions are type-annotated for convenience, but types play no role in reduction. We let $\overset{+}{\longrightarrow}$ stand for the transitive closure of $\longrightarrow$, and $\overset{*}{\longrightarrow}$ be the reflexive-transitive closure of $\longrightarrow$. We postpone detailed comments on the operational model for the next section.

In the following guided tour, we will often appeal to notions such as *communication channels* and later to *memory cells*, for the sake of conveying intuition about the operational model. However, we would like to stress that, in our approach, process reduction is explained solely by pure algebraic manipulations on process terms that contain variables and variable binding operators, as usual in computational interpretations of logic, therefore *communication channels* and *memory cells* are nothing but standard variables in proof terms. In our practical interpreter implementation (Chapter 5), we naturally resort to actual channels and memory locations, but that is a different realm.

2.3 Type System

In this section we define the type system of μCLL. Then, we go through each of the typing rules in detail by presenting the associated structural congruence rules (Definition 4) and all the principal cut conversions of reduction $\rightarrow$ (Definition 5). The introduced concepts will be illustrated with several examples.

We start by defining typing contexts and typing judgments. A typing context is a finite partial assignment from names to types, which we denote by

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\Delta} \; ; \; \underbrace{y_1 : B_1, \ldots, y_m : B_m}_{\Gamma}$$

Typing contexts are separated (with a semi-colon) into two parts: a linear part denoted by Δ and an unrestricted (or exponential) part, which absorbs weakening and contraction, and is denoted by Γ. The empty context is written $\emptyset$. We write Δ, Δ' (two comma-separated contexts) for the disjoint union of Δ and Δ'. Given a set S of names we write $\Delta \restriction S$ to denote the partial map obtained by restricting Δ to S.

The set of free type variables of a typing context is the union of the free type variables of the types in the image of the typing context. We denote by $\{A/X\}(\Delta;\Gamma)$ the typing context obtained by replacing the free type variable X in every type in the image of $\Delta;\Gamma$ by A. Similarly, we extend name substitutions to typing contexts accordingly, written $\{\vec{x}/\vec{y}\}(\Delta;\Gamma)$, which affects the domain of $\Delta;\Gamma$ by replacing each name $\vec{y}[i]$ by $\vec{x}[i]$.

Typing judgments are of the form $P \vdash_\eta \Delta;\Gamma$ where P is a process, $\Delta;\Gamma$ is a typing context and η is a finite partial map

$$\eta = X_1(\vec{x_1}) \mapsto \Delta_1;\Gamma, \ldots, X_n(\vec{x_n}) \mapsto \Delta_n;\Gamma_n$$

where recursion variables are assigned to typing contexts, this essentially allows to type corecursive process definitions in μCLL. Later, in 2.3, we will see in detail the role of η.

If Γ is empty we write just $P \vdash_\eta \Delta$ instead of $P \vdash_\eta \Delta;\emptyset$. Sometimes we omit the subscript η, writing $P \vdash \Delta;\Gamma$.

We define $\{y/x\}(\Delta;\Gamma)$ by cases:

$$\begin{array}{lcl} \{y/x\}(\Delta' x : A;\Gamma) & = & \Delta', y : A;\Gamma \\ \{y/x\}(\Delta;\Gamma', x : A) & = & \Delta;\Gamma, y : A \\ \{y/x\}(\Delta;\Gamma) & = & \Delta;\Gamma,\ x \notin \mathrm{dom}(\Delta) \cup \mathrm{dom}(\Gamma) \end{array}$$

We denote by $\{A/X\}(\Delta;\Gamma)$ the typing context obtained by replacing the free type variable X by A in every type in the image of $\Delta;\Gamma$. Similarly, we extend simultaneous substitutions to typing contexts accordingly, written $\{\vec{x}/\vec{y}\}(\Delta;\Gamma)$.

Definition 6 (Type System)**.** *The typing rules of* μCLL *are listed in Fig. 2.4.*

A process P is well-typed if $P \vdash_\eta \Delta;\Gamma$ for some typing contexts Δ and Γ and map η.

In the following subsections we go through each of the typing rules in detail by presenting the associated structural congruence rules (Definition 4) and all the principal cut conversions of reduction $\longrightarrow$ (Definition 5).

Inaction and Mix

The inaction process $\mathbf{0}$ operationally does nothing, i.e. there is no process P for which

$$\mathbf{0} \longrightarrow P$$

and types with an empty linear context.

$$\frac{}{\mathbf{0} \vdash \emptyset;\Gamma}\ [\mathrm{T}\mathbf{0}]$$

The process construct $\mathbf{par}\ \{P \parallel Q\}$ composes P and Q, which run in parallel without ever interfering with each other. It is typed by rule [Tmix]

$$\frac{P \vdash \Delta_1;\Gamma \quad Q \vdash \Delta_2;\Gamma}{\mathbf{par}\ \{P \parallel Q\} \ \vdash \Delta_1, \Delta_2;\Gamma}\ [\mathrm{Tmix}]$$

which corresponds to linear logic mix. Processes P and Q type with disjoint linear typing contexts Δ_1 and Δ_2, hence there is possible linear interaction between P and Q.

There are some structural congruence $\equiv$ rules involving inaction and mix (Fig. 2.2):

$$\begin{array}{lcll} \mathbf{par}\ \{P \parallel \mathbf{0}\} & \equiv & P & [\mathbf{0}\mathrm{M}] \\ \mathbf{par}\ \{P \parallel Q\} & \equiv & \mathbf{par}\ \{Q \parallel P\} & [\mathrm{M}] \\ \mathbf{par}\ \{P \parallel (\mathbf{par}\ \{Q \parallel R\})\} & \equiv & \mathbf{par}\ \{(\mathbf{par}\ \{P \parallel Q\}) \parallel R\} & [\mathrm{MM}] \end{array}$$

Inaction is a unit for independent parallel composition as expressed by $\equiv$-rule [$\mathbf{0}$M]. Additionally, since mix is commutative (rule [M]) and associative (rule [MM]), we conclude that the set of processes modulo $\equiv$ forms a commutative monoid where the multiplication is given by $\parallel$ and whose unit is $\mathbf{0}$.

A final remark concerning the unrestricted context Γ: rule [T$\mathbf{0}$] types inaction with an arbitrary unrestricted typing context Γ in which the unrestricted names are not used. As

$$\frac{}{\mathbf{0} \vdash_\eta \emptyset; \Gamma}\ [\mathrm{T}\mathbf{0}] \qquad \frac{P \vdash_\eta \Delta'; \Gamma \quad Q \vdash_\eta \Delta; \Gamma}{\mathbf{par}\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta; \Gamma}\ [\mathrm{Tmix}]$$

$$\frac{}{\mathbf{fwd}\ x\ y \vdash_\eta x : \overline{A}, y : A; \Gamma}\ [\mathrm{Tfwd}] \qquad \frac{P \vdash_\eta \Delta', x : A; \Gamma \quad Q \vdash_\eta \Delta, x : \overline{A}; \Gamma}{\mathbf{cut}\ \{P \ |x : A|\ Q\} \vdash_\eta \Delta', \Delta; \Gamma}\ [\mathrm{Tcut}]$$

$$\frac{}{\mathbf{close}\ x \vdash_\eta x : \mathbf{1}; \Gamma}\ [\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash_\eta \Delta; \Gamma}{\mathbf{wait}\ x; Q \vdash_\eta \Delta, x : \bot; \Gamma}\ [\mathrm{T}\bot]$$

$$\frac{P_1 \vdash_\eta \Delta, x : A; \Gamma \quad P_2 \vdash_\eta \Delta, x : B; \Gamma}{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\} \vdash_\eta \Delta, x : A \,\&\, B; \Gamma}\ [\mathrm{T}\&]$$

$$\frac{Q_1 \vdash_\eta \Delta', x : A; \Gamma}{x.\mathbf{inl}; Q_1 \vdash_\eta \Delta', x : A \oplus B; \Gamma}\ [\mathrm{T}\oplus_l] \qquad \frac{Q_2 \vdash_\eta \Delta', x : B; \Gamma}{x.\mathbf{inr}; Q_2 \vdash_\eta \Delta', x : A \oplus B; \Gamma}\ [\mathrm{T}\oplus_r]$$

$$\frac{P_1 \vdash_\eta \Delta_1, y : A; \Gamma \quad P_2 \vdash_\eta \Delta_2, x : B; \Gamma}{\mathbf{send}\ x(y.P_1); P_2 \vdash_\eta \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes] \qquad \frac{Q \vdash_\eta \Delta, z : A, x : B; \Gamma}{\mathbf{recv}\ x(z); Q \vdash_\eta \Delta, x : A \parr B; \Gamma}\ [\mathrm{T}\parr]$$

$$\frac{P \vdash_\eta y : A; \Gamma}{!x(y); P \vdash_\eta x :!A; \Gamma}\ [\mathrm{T}!] \qquad \frac{Q \vdash_\eta \Delta; \Gamma, x : A}{?x; Q \vdash_\eta \Delta, x :?A; \Gamma}\ [\mathrm{T}?]$$

$$\frac{P \vdash_\eta y : A; \Gamma \quad Q \vdash_\eta \Delta; \Gamma, x : \overline{A}}{\mathbf{cut!}\ \{y.P \ |x : A|\ Q\} \vdash_\eta \Delta; \Gamma}\ [\mathrm{Tcut!}] \qquad \frac{Q \vdash_\eta \Delta, z : A; \Gamma, x : A}{\mathbf{call}\ x(z); Q \vdash_\eta \Delta; \Gamma, x : A}\ [\mathrm{Tcall}]$$

$$\frac{P \vdash_\eta \Delta, x : \{B/X\}A; \Gamma}{\mathbf{sendty}\ x\ B; P \vdash_\eta \Delta, x : \exists X.A; \Gamma}\ [\mathrm{T}\exists] \qquad \frac{Q \vdash_\eta \Delta, x : A; \Gamma}{\mathbf{recvty}\ x(X); Q \vdash_\eta \Delta, x : \forall X.A; \Gamma}\ [\mathrm{T}\forall]$$

$$\frac{P \vdash_{\eta'} \Delta, z : A; \Gamma \quad \eta' = \eta, X(z, \vec{w}) \mapsto \Delta, z : Y; \Gamma}{\mathbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}] \vdash_\eta \{\vec{y}/\vec{w}\}\Delta, x : \nu Y.\ A; \{\vec{y}/\vec{w}\}\Gamma}\ [\mathrm{Tcorec}]$$

$$\frac{\eta = \eta', X(x, \vec{y}) \mapsto \Delta, x : Y; \Gamma}{X(z, \vec{w}) \vdash_\eta \{\vec{w}/\vec{y}\}(\Delta, z : Y; \Gamma)}\ [\mathrm{Tvar}] \qquad \frac{P \vdash_\eta \Delta, x : \{\nu X.\ A/X\}A; \Gamma}{\mathbf{unfold}_\nu\ x; P \vdash_\eta \Delta, x : \nu X.\ A; \Gamma}\ [\mathrm{T}\nu]$$

$$\frac{P \vdash_\eta \Delta, x : \{\mu X.\ A/X\}A; \Gamma}{\mathbf{unfold}_\mu\ x; P \vdash_\eta \Delta, x : \mu X.\ A; \Gamma}\ [\mathrm{T}\mu]$$

Figure 2.4: Typing rules $P \vdash_\eta \Delta; \Gamma$ for μCLL (extended by Fig. 3.4).

we shall see, this applies to all the axiom rules, hence in μCLL we have a contained form of weakening that applies to the leaves of typing derivations. Furthermore, the unrestricted context also absorbs contraction: in the non-axiom rules involving more than one premise, as it is the case, for example, with [Tmix], the unrestricted typing context is copied from the conclusion to the premises.

Cut, Duality and Forwarding

Whereas mix is independent parallel composition, the cut construct allows to build dependent interactive composition. In a cut $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}$, processes P and Q run concurrently, interacting on a single private linear session x. This restriction - that two concurrent processes interact through a single session - guarantees acyclicity on the process communication topologies, which then implies desirable metatheoretical properties such as the absence of deadlocks.

The cut construct is typed by [Tcut]

$$\frac{P \vdash \Delta', x : A; \Gamma \quad Q \vdash \Delta, x : \overline{A}; \Gamma}{\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \vdash \Delta', \Delta; \Gamma}\ [\text{Tcut}]$$

which corresponds precisely to the cut rule of linear logic. Process P provides a behaviour of type A along x, whereas Q offers on x a dual behaviour of type $\overline{A}$. Cut is annotated with the type of its left argument, but we sometimes omit this annotation.

The cut construct is commutative as modelled by $\equiv$ rule [C]:

$$\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \equiv \mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\}$$

On going from left to right we swap the arguments and replace the cut annotation A by its dual $\overline{A}$. Cut linearly distributes over all the static constructs. For example, the linearly distribution of cut over mix is modelled by $\equiv$ rule [CM]

$$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{par}\ \{Q\ ||\ R\}\} \equiv \mathbf{par}\ \{\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\ ||\ R\},\ x \in \mathsf{fn}(Q)$$

The distribution is guided by the proviso $x \in \mathsf{fn}(Q)$.

In μCLL we also have forwarding. The forwarding process $\mathbf{fwd}\ x\ y$ acts as a link between the two channel endpoints x and y, forwarding all the interactions with x to y and vice-versa. It is typed by

$$\frac{}{\mathbf{fwd}\ x\ y \vdash x : \overline{A}, y : A; \Gamma}\ [\text{Tfwd}]$$

and corresponds to the identity axiom of linear logic. are interpreted by processes $\mathbf{fwd}\ x\ y$ and $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}$, respectively.

Operationally, the forwarding behaviour is modelled by name substitution [30]

$$\mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |y : A|\ \ P\} \quad \rightarrow \quad \{x/y\}P \qquad [\text{fwd}]$$

Here, on the left-hand side of the reduction rule, a forwarding between x and y is composed via a cut on y with a process P. On the right-hand side there is process P with name y substituted by x. Algebraically, forwarding behaves like an identity for cut composition. The following typing rule for name substitution is admissible

$$\frac{P \vdash \Delta, y : B; \Gamma}{\{x/y\}P \vdash \Delta, x : B; \Gamma}$$

provided y is not a free name of P.

The symmetric behaviour of the forwarding construct is patent on structural congruence law [fwd]

$$\textbf{fwd}\ x\ y\ \equiv \textbf{fwd}\ y\ x$$

Close and Wait

Session termination is expressed by the processes **close** x, that closes a session x, and **wait** $x; Q$, that symmetrically waits for x to be closed and then continues as Q. Close and wait are typed by dual types **1** (one) and $\bot$ (bottom)

$$\frac{}{\textbf{close}\ x \vdash x : \mathbf{1}; \Gamma}\ [\mathrm{T}\mathbf{1}] \qquad \frac{Q \vdash \Delta; \Gamma}{\textbf{wait}\ x; Q\ \vdash \Delta, x : \bot; \Gamma}\ [\mathrm{T}\bot]$$

The interaction between close and wait is modelled by the reduction rule

$$\textbf{cut}\ \{\textbf{close}\ x\ \ |x : \mathbf{1}|\ \ \textbf{wait}\ x; Q\} \quad \rightarrow \quad Q \quad [\mathbf{1}\bot]$$

where, after the interaction, the cut evolves to the continuation process Q and the session on x disappears. The reduction rule corresponds to the following conversion at the level of proofs / typing conversions

$$\frac{\dfrac{}{\textbf{close}\ x \vdash x : \mathbf{1}; \Gamma}\ [\mathrm{T}\mathbf{1}] \qquad \dfrac{Q \vdash \Delta; \Gamma}{\textbf{wait}\ x; Q\ \vdash \Delta, x : \bot; \Gamma}\ [\mathrm{T}\bot]}{\textbf{cut}\ \{\textbf{close}\ x\ \ |x|\ \ \textbf{wait}\ x; Q\} \vdash \Delta; \Gamma}\ [\mathrm{Tcut]} \quad \rightarrow \quad Q \vdash \Delta; \Gamma$$

Send and Receive

Session communication is expressed by processes **send** $x(y.P_1); P_2$ (send) and **recv** $x(z); Q$ (receive). Process **send** $x(y.P_1); P_2$ sends a fresh session channel y on session x and continues as P_2. Symmetrically, process **recv** $x(z); Q$ receives a name z on session x and continues as Q. Name y is bound in P_1 and z is bound in Q.

Send and receive are typed by the dual types $A \otimes B$ (tensor) and $\overline{A} \parr \overline{B}$ (par)

$$\frac{P_1 \vdash \Delta_1, y : A; \Gamma \quad P_2 \vdash \Delta_2, x : B; \Gamma}{\textbf{send}\ x(y.P_1); P_2\ \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [\mathrm{T}\otimes] \qquad \frac{Q \vdash \Delta, z : \overline{A}, x : \overline{B}; \Gamma}{\textbf{recv}\ x(z); Q \vdash \Delta, x : \overline{A} \parr \overline{B}; \Gamma}\ [\mathrm{T}\parr]$$

The send process is composed of two independent parts: a term P_1 which implements the session on fresh channel $y : A$ to be sent, and a term P_2 which provides the continuation session behaviour on $x : B$.

The associated reduction between send and receive is expressed by

$$\textbf{cut}\ \{\textbf{send}\ x(y.P_1); P_2\ \ |x : A \otimes B|\ \ \textbf{recv}\ x(z); Q\} \rightarrow \textbf{cut}\ \{P_2\ \ |x : B|\ \ \textbf{cut}\ \{P_1\ \ |y : A|\ \ \{y/z\}Q\}\}\ [\otimes\parr]$$

Notice that a cut on a session $A \otimes B$ gives origin to two *lower rank* cuts on sessions A and B. The inner cut on y connects the continuation Q of the receiver with the provider P_1 of the sent channel, whereas the outer cut on x connects Q to the continuation P_2 of the sending process. Observe that the type associated with session x evolves from $A \otimes B$ to B upon communication.

As in [23, 157], only fresh (bound) names are sent in communication, following the *internal* mobility discipline of Boreale [17], as opposed to *external* mobility in which free names can be transmitted. However, we can encode free output of a (free) linear name with the following definition, justified by Proposition 1.

Definition 7 (Free Output of a Linear Name)**.** *Define*

$$\mathbf{send}\ x(y); P \triangleq \mathbf{send}\ x(w.\ \mathbf{fwd}\ w\ y); P$$

Proposition 1. *The following typing*

$$\frac{P \vdash \Delta, x : B; \Gamma}{\mathbf{send}\ x(y); P \vdash \Delta, y : \overline{A}, x : A \otimes B; \Gamma}\ [T\otimes_f]$$

and reduction rule

$$\mathbf{cut}\ \{\mathbf{send}\ x(y); P\ \ |x : A \otimes B|\ \ \mathbf{recv}\ x(z); Q\} \xrightarrow{+} \mathbf{cut}\ \{P\ \ |x : B|\ \ \{y/z\}Q\} \qquad [\otimes\parr_f]$$

are derivable.

Proof. Rule [T$\otimes_f$] is derivable by

$$\frac{\dfrac{}{\mathbf{fwd}\ w\ y \vdash w : A, y : \overline{A}; \Gamma}\ [\text{Tfwd}] \qquad P \vdash \Delta, x : B; \Gamma}{\mathbf{send}\ x(w.\mathbf{fwd}\ y\ w); P \vdash \Delta, y : \overline{A}, x : A \otimes B; \Gamma}\ [\text{T}\otimes]$$

Rule [$\otimes\parr_f$] is derivable by

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{send}\ x(y); P\ \ |x : A \otimes B|\ \ \mathbf{recv}\ x(z); Q\} \\
&= \mathbf{cut}\ \{\mathbf{send}\ x(w.\ \mathbf{fwd}\ w\ y); P\ \ |x : A \otimes B|\ \ \mathbf{recv}\ x(z); Q\} && \text{(Def. 7)} \\
&\to \mathbf{cut}\ \{P\ \ |x : B|\ \ \mathbf{cut}\ \{\mathbf{fwd}\ w\ y\ \ |y : A|\ \ \{y/z\}Q\}\} && (\to \text{ rule } [\otimes\parr]) \\
&\to \mathbf{cut}\ \{P\ \ |x : B|\ \ \{w/y\}(\{y/z\}Q)\} && (\to \text{ rule [fwd]}) \\
&= \mathbf{cut}\ \{P\ \ |x : B|\ \ \{w/z\}Q\} && (\{w/y\}(\{y/z\}Q) = \{w/z\}Q)
\end{aligned}$$

□

The following example illustrates the symmetric nature of the tensor construct $\otimes$.

Example 1 (Symmetry of $\otimes$)**.** *There is a canonical proof term that witnesses commutativity of the tensor construct*

$$\mathsf{symm}(x, y) \triangleq \mathbf{recv}\ x(z); \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{fwd}\ z\ y$$

$\mathsf{symm}(x, y)$ *converts a typed channel* $x : A \otimes B$ *to* $y : B \otimes A$*, therefore testifying the symmetric nature of* $\otimes$*. We have*

$$\mathsf{symm}(x, y) \vdash x : \overline{A \otimes B}, y : B \otimes A; \Gamma$$

the derivation of which is a follows

$$\dfrac{\dfrac{\dfrac{}{\mathbf{fwd}\ w\ x \vdash w : B, x : \overline{B}; \Gamma}\ [Tfwd] \quad \dfrac{}{\mathbf{fwd}\ z\ y \vdash z : \overline{A}, y : A}\ [Tfwd]}{\mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{fwd}\ z\ y \vdash z : \overline{A}, x : \overline{B}, y : B \otimes A; \Gamma}\ [T\otimes]}{\mathbf{recv}\ x(z); \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{fwd}\ z\ y \vdash x : \overline{A} \parr \overline{B}, y : B \otimes A; \Gamma}\ [T\parr]$$

When we compose symm *with a process that outputs* A *and continues as* B *on channel* x

$$\dfrac{P_1(u) \vdash \Delta_1, u : A; \Gamma \quad P_2(x) \vdash \Delta_2, x : B; \Gamma}{\mathbf{send}\ x(y.P_1); P_2 \vdash \Delta_1, \Delta_2, x : A \otimes B; \Gamma}\ [T\otimes]$$

we obtain a process that outputs B *and continues as* A

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{send}\ x(u.P_1(u)); P_2(x)\ \ |x|\ \ \mathsf{symm}(x, y)\} \\
&= \mathbf{cut}\ \{\mathbf{send}\ x(u.P_1(u)); P_2(x)\ \ |x|\ \ \mathbf{recv}\ x(z); \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{fwd}\ z\ y\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)} \\
&\to \mathbf{cut}\ \{P_2(x)\ \ |x|\ \ \mathbf{cut}\ \{P_1(u)\ \ |u|\ \ \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{fwd}\ u\ y\}\} \qquad (\to [\otimes\parr]) \\
&\approx \mathbf{cut}\ \{P_2(x)\ \ |x|\ \ \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); \mathbf{cut}\ \{\mathbf{fwd}\ u\ y\ \ |u|\ \ P_1(u)\}\} \\
&\to \mathbf{cut}\ \{P_2(x)\ \ |x|\ \ \mathbf{send}\ y(w.\mathbf{fwd}\ w\ x); P_1(y)\} \qquad (\to [\text{fwd}]) \\
&\approx \mathbf{send}\ y(w.\mathbf{cut}\ \{\mathbf{fwd}\ w\ x\ \ |x|\ \ P_2\}); P_1(y) \\
&\to \mathbf{send}\ y(w.P_2(w)); P_1(y) \qquad (\to [\text{fwd}])
\end{aligned}$$

In the derivation we used two $\approx$ *laws that commute the send action with cut and which are not part of structural congruence. We will present the relation* $\approx$*, which extends structural congruence* $\equiv$ *with further commuting conversions, appropriately when studying cut normalisation (see Chapter 8). This example was adapted from [158] and [23].*

Offer and Choice

Menu offer and choice is expressed by processes $\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}$ and $x.\mathbf{inl}; Q_1$, $x.\mathbf{inr}; Q_2$, respectively. Process $\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}$ offers on session x a menu of two options: inl (left), after which it continues as P_1 or inr (right), after which it continues as P_2. Symmetrically, process $x.\mathbf{inl}; Q_1$ (resp., $x.\mathbf{inr}; Q_2$) chooses inl (resp., inr) on session x and continues as Q_1 (resp., Q_2).

Offer and choice are typed by $A \,\&\, B$ (with) and $\overline{A} \oplus \overline{B}$ (plus), respectively

$$\dfrac{P_1 \vdash \Delta, x : A; \Gamma \quad P_2 \vdash \Delta, x : B; \Gamma}{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\} \vdash \Delta, x : A \,\&\, B; \Gamma}\ [\mathrm{T}\&]$$

$$\dfrac{Q_1 \vdash \Delta', x : \overline{A}; \Gamma}{x.\mathbf{inl}; Q_1 \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma}\ [\mathrm{T}\oplus_l] \qquad \dfrac{Q_2 \vdash \Delta', x : \overline{B}; \Gamma}{x.\mathbf{inr}; Q_2 \vdash \Delta', x : \overline{A} \oplus \overline{B}; \Gamma}\ [\mathrm{T}\oplus_r]$$

The interaction between offer and choice is expressed by the following two associated reduction rules

$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inl}; Q_1\} \ \longrightarrow\ \mathbf{cut}\ \{P_1\ \ |x : A|\ \ Q_1\} \quad [\&\oplus_l]$$
$$\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inr}; Q_2\} \ \longrightarrow\ \mathbf{cut}\ \{P_2\ \ |x : B|\ \ Q_2\} \quad [\&\oplus_r]$$

We conclude this subsection with some examples. First we code the optional datatype $\mathsf{Maybe}\ A$, then we illustrate how generic labelled-choice can be implemented with binary left-right choice. Finally, we conclude by coding the booleans and define some operations using the session-based constructs that we have already introduced.

Example 2 (Maybe)**.** *The type*

$$\mathsf{Maybe}\ A \triangleq \mathbf{1} \oplus A$$

is the protocol of sessions that either choose left and close or choose right and continue as a session of type A.

We can think of sessions of type $\mathsf{Maybe}\ A$ *as sessions that* maybe *offer a behaviour of type A, though they have the possibility of offering no behaviour at all. As we shall see, this type arises naturally, for example it allows us to express the returning type of a non-blocking dequeue operation, where we may return just an element if the queue is nonempty or we return nothing otherwise.*

Processes

$$\mathsf{nothing}(x) \vdash x : \mathsf{Maybe}\ A \quad \mathsf{just}(a, x) \vdash a : \overline{A}, x : \mathsf{Maybe}\ A$$

are defined by

$$\begin{aligned} \mathsf{nothing}(x) &\triangleq x.\mathbf{inl}; \mathbf{close}\ x \\ \mathsf{just}(a, x) &\triangleq x.\mathbf{inr}; \mathbf{fwd}\ a\ x \end{aligned}$$

Example 3 (Labelled Choice)**.** *In this example we illustrate how binary left-right offer and choice is sufficient to implement offer and choice from an arbitrary set of labels. For the purpose, we consider menu offer and choice from a set* $\{l_1, l_2, l_3\}$ *of three labelled possibilities.*

We have the following definitions

$$\begin{aligned}
&\&\{l_1 : A_1, l_2 : A_2, l_3 : A_3\} && \triangleq && (A_1 \,\&\, A_2) \,\&\, A_3 && (1)\\
&\oplus\{l_1 : \overline{A}_1, l : \overline{A}_2, l_3 : \overline{A}_3\} && \triangleq && (\overline{A}_1 \oplus \overline{A}_2) \oplus \overline{A}_3 && (2)\\
&\mathsf{case}\ x\ \{\ |\ l_1 : P_1 \mid l_2 : P_2 \mid l_3 : P_3\} && \triangleq && \mathbf{case}\ x\ \{|\mathsf{inl} : (\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}) \mid \mathsf{inr} : P_3\} && (3)\\
&x.l_1; Q_1 && \triangleq && x.\mathbf{inl}; x.\mathbf{inl}; Q_1 && (4)\\
&x.l_2; Q_2 && \triangleq && x.\mathbf{inl}; x.\mathbf{inr}; Q_2 && (5)\\
&x.l_3; Q_3 && \triangleq && x.\mathbf{inr}; Q_3 && (6)
\end{aligned}$$

The following typing rules are then derivable

$$\frac{P_1 \vdash \Delta, x : A_1; \Gamma \quad P_2 \vdash \Delta, x : A_2; \Gamma \quad P_3 \vdash \Delta, x : A_3; \Gamma}{\mathsf{case}\ x\ \{\ |\ l_1 : P_1 \mid l_2 : P_2 \mid l_3 : P_3\} \vdash \Delta, x : \&\{l_1 : A_1, l_2 : A_2, l_3 : A_3\}}$$

$$\frac{Q_i \vdash \Delta', x : \overline{A}_i; \Gamma}{x.l_i; Q_i \vdash \Delta', x : \oplus\{l_1 : \overline{A}_1, l_2 : \overline{A}_2, l_3 : \overline{A}_3\}}$$

for all $1 \leq i \leq 3$.

The offering process can interact with three possible choices, which we model by the principal cut conversions

$$\mathbf{cut}\ \{\mathsf{case}\ x\ \{\ |\ l_1 : P_1\ |\ l_2 : P_2\ |\ l_3 : P_3\}\ \ |x|\ \ x.l_i; Q_i\} \to \mathbf{cut}\ \{P_i\ \ |x|\ \ Q_i\},\ 1 \leq i \leq 3$$

We illustrate how to derive one of the reduction rules:

$$\begin{aligned}
&\mathbf{cut}\ \{\mathsf{case}\ x\ \{\ |\ l_1 : P_1\ |\ l_2 : P_2\ |\ l_3 : P_3\}\ \ |x|\ \ x.l_2; Q_2\} \\
&= \mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : (\mathbf{case}\ x\ \{|\mathsf{inl} : P_1\ |\ \mathsf{inr} : P_2\})\ |\ \mathsf{inr} : P_3\}\ \ |x|\ \ x.\mathbf{inl}; x.\mathbf{inr}; Q_2\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)} \\
&\to \mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1\ |\ \mathsf{inr} : P_2\}\ \ |x|\ \ x.\mathbf{inr}; Q_2\} \qquad\qquad (\to [\&\oplus_l]) \\
&\to \mathbf{cut}\ \{P_2\ \ |x|\ \ Q_2\} \qquad\qquad (\to [\&\oplus_r])
\end{aligned}$$

Hereafter, we will assume the existence of labelled choices with an arbitrary number of options.

Example 4 (Booleans)**.** *This example illustrates how basic programming data types can be encoded as pure session types in* μCLL*. In this example, we encode booleans but, as we shall see later, more complex data structures such as naturals (Examples 6) can be encoded.*

We encode the linear booleans as sessions of type

$$\mathsf{Bool} \triangleq \oplus\{\ |\ \mathsf{F} : \mathbf{1}\ |\ \mathsf{T} : \mathbf{1}\}$$

A boolean session chooses either F *(false) or* T *(true) and then closes. The basic boolean constants* true *and* false *are then defined by*

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{false}(b) & \vdash & b : \mathsf{Bool} & \mathsf{true}(b) & \vdash & b : \mathsf{Bool} \\
\mathsf{false}(b) & \triangleq & \mathbf{F}\ b; \mathbf{close}\ b & \mathsf{true}(b) & \triangleq & \mathbf{T}\ b; \mathbf{close}\ b
\end{array}$$

Process $\mathsf{false}(b)$ *chooses* F *on session* b*, whereas process* $\mathsf{true}(b)$ *chooses* T*. Then, both processes* $\mathsf{false}(b)$ *and* $\mathsf{true}(b)$ *close the session* b*. We now define some processes that operate on linear booleans. Process* $\mathsf{discard}(b) \vdash b : \overline{\mathsf{Bool}}$*, defined by*

$$\mathsf{discard}(b) \triangleq \mathbf{case}\ b\ \{\ |\ \mathsf{F} : \mathbf{wait}\ b; \mathbf{0}\ \ |\ \mathsf{T} : \mathbf{wait}\ b; \mathbf{0}\}$$

performs case analysis on session b and in each branch of the case it waits for b to be closed, after which continues as the inaction process, thereby discarding the linear boolean on b. Some linear types (such as Bool*) can be discarded and copied, in fact these principles are derivable for some types of linear logic [56]. The following pair of reductions are easily derivable*

$$\mathbf{cut}\ \{\mathsf{false}(b)\ \ |b : \mathsf{Bool}|\ \ \mathsf{discard}(b)\} \xrightarrow{+} \mathbf{0}\ (1) \qquad \mathbf{cut}\ \{\mathsf{true}(b)\ \ |b : \mathsf{Bool}|\ \ \mathsf{discard}(b)\} \xrightarrow{+} \mathbf{0}\ (2)$$

Process $\mathsf{not}(b, b') \vdash b : \overline{\mathsf{Bool}}, b' : \mathsf{Bool}$ *interacts with a boolean b and offers its negation on session* b'

$$\mathsf{not}(b, b') \triangleq \mathbf{case}\ b\ \{\ |\ \mathsf{F} : \mathbf{wait}\ b; \mathsf{true}(b')\ |\ \mathsf{T} : \mathbf{wait}\ b; \mathsf{false}(b')\}$$

We have

$$\mathbf{cut}\ \{\mathsf{false}(b)\ \ |b : \mathsf{Bool}|\ \ \mathsf{not}(b, b')\} \xrightarrow{+} \mathsf{true}(b') \quad (3)$$
$$\mathbf{cut}\ \{\mathsf{true}(b)\ \ |b : \mathsf{Bool}|\ \ \mathsf{not}(b, b')\} \xrightarrow{+} \mathsf{false}(b') \quad (4)$$

Finally, process $\mathsf{and}(b_1, b_2, b)$ *interacts with booleans* b_1 *and* b_2 *and produces their logical conjunction on channel* b

$$\mathsf{and}(b_1, b_2, b) \triangleq \mathbf{case}\ b_1\ \{\ |\ \mathsf{F} : \mathbf{wait}\ b_1; \mathbf{par}\ \{\mathsf{discard}(b_2)\ ||\ \mathsf{false}(b)\}\ |\ \mathsf{T} : \mathbf{wait}\ b_1; \mathbf{fwd}\ b\ b_2\}$$

Process $\mathsf{and}(b_1, b_2, b)$ *starts by performing case analysis on* b_1*. If* b_1 *chooses* F*, which means that the boolean on* b_1 *is false, then it waits for* b_1 *to be closed and then, in parallel, discards* b_2 *and defines the boolean false on* b *(the output is false regardless of the input* b_2*). On the other hand, if* b_1 *chooses* T*, which corresponds to the encoding of the boolean true, then it waits for* b_1 *to be closed and then forwards the output* b *to the input* b_2*. As expected, we have the following*

$$\mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{false}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \xrightarrow{+} \mathsf{false}(b) \quad (5)$$
$$\mathbf{cut}\ \{\mathsf{true}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{false}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \xrightarrow{+} \mathsf{false}(b) \quad (6)$$
$$\mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{true}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \xrightarrow{+} \mathsf{false}(b) \quad (7)$$
$$\mathbf{cut}\ \{\mathsf{true}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{true}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \xrightarrow{+} \mathsf{true}(b) \quad (8)$$

We derive (5)

$$\begin{aligned}
&\mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{false}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \\
&\to \mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathbf{close}\ b_1\ \ |b_1|\ \ \mathbf{wait}\ b_1; \mathbf{par}\ \{\mathsf{discard}(b_2)\ ||\ \mathsf{false}\ b\}\}\} \\
&\qquad\qquad \text{(expanding def. of } \mathsf{false}(b_1) \text{ and } \mathsf{and}(b_1, b_2, b) \text{ and labelled choice)} \\
&\to \mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathbf{par}\ \{\mathsf{discard}(b_2)\ ||\ \mathsf{false}(b)\}\} && (\to \text{ rule } [\mathbf{1}\bot]) \\
&\equiv \mathbf{par}\ \{\mathbf{cut}\ \{\mathsf{false}(b_2)\ \ |b_2|\ \ \mathsf{discard}(b_2)\}\ ||\ \mathsf{false}(b)\} && (\equiv \text{ rule [C-CM]}) \\
&\xrightarrow{+} \mathbf{par}\ \{\mathbf{0}\ ||\ \mathsf{false}(b)\} && \text{(from (1))} \\
&\equiv \mathsf{false}(b) && (\equiv \text{ rule } [\mathbf{0}\mathrm{M}])
\end{aligned}$$

and (8)

$$\begin{aligned}
&\mathbf{cut}\ \{\mathsf{true}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathsf{true}(b_1)\ \ |b_1|\ \ \mathsf{and}(b_1, b_2, b)\}\} \\
&\to \mathbf{cut}\ \{\mathsf{true}(b_2)\ \ |b_2|\ \ \mathbf{cut}\ \{\mathbf{close}\ b_1\ \ |b_1|\ \ \mathbf{wait}\ b_1; \mathbf{fwd}\ b\ b_2\}\} \\
&\qquad\qquad \text{(expanding def. of } \mathsf{true}(b_1) \text{ and } \mathsf{and}(b_1, b_2, b) \text{ and labelled choices)} \\
&\to \mathbf{cut}\ \{\mathsf{true}(b_2)\ \ |b_2|\ \ \mathbf{fwd}\ b\ b_2\} && (\to \text{ rule } [\mathbf{1}\bot]) \\
&\equiv \mathbf{cut}\ \{\mathbf{fwd}\ b\ b_2\ \ |b_2|\ \ \mathsf{true}(b_2)\} && (\equiv \text{ rule [C-C]}) \\
&\to \{b/b_2\}\mathsf{true}(b_2) = \mathsf{true}(b) && (\to \text{ rule [fwd]})
\end{aligned}$$

The derivations of (6) and (7) can be established along similar lines.

Code for this example and some tests is available in examples/pure/booleans.clls.

Persistent Sessions

Until now, all the introduced process constructs operate on linear sessions. The input-replication process $!x(y);P$ defines a persistent session on x with argument y and body P. A persistent session x can be invoked with $\mathbf{call}\ x(z);Q$, which calls x with input z, and continues as Q.

The replication process $!x(y);P$ is typed by $!A$ (bang), whereas $\mathbf{call}\ x(z);Q$ is implicitly typed by $?A$ (why not), as expressed by rules [T!] and [Tcall]

$$\frac{P \vdash_\eta y : A;\Gamma}{!x(y);P \ \vdash_\eta x :!A;\Gamma}\ [\text{T!}] \qquad \frac{Q \vdash_\eta \Delta;\Gamma, x : \overline{A}}{?x;Q \ \vdash_\eta \Delta, x :?\overline{A};\Gamma}\ [\text{T?}]$$

$$\frac{P \vdash_\eta y : A;\Gamma \quad Q \vdash \Delta;\Gamma, x : \overline{A}}{\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \ \vdash_\eta \Delta;\Gamma}\ [\text{Tcut!}] \qquad \frac{Q \vdash_\eta \Delta, z : \overline{A};\Gamma, x : \overline{A}}{\mathbf{call}\ x(z);Q \ \vdash_\eta \Delta;\Gamma, x : \overline{A}}\ [\text{Tcall}]$$

We also display rule [T?] which types $?x;Q$ and the unrestricted cut [Tcut!]. Process $?x;Q$ moves the linear session $x :?\overline{A}$ to the unrestricted context, typed as $x : \overline{A}$, where it can now be shared by many clients and called as many times as necessary.

Whereas [Tcut] acts on the linear context, [Tcut!] acts on the unrestricted context. The term $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}$ composes P with a pool of clients represented by Q, and which can call session x an unbounded (possibly zero) number of times.

The interaction between [T!] and [T?] is expressed by rule [!?]

$$\mathbf{cut}\ \{!x(y);P\ \ |x :!A|\ \ ?x;Q\} \quad \rightarrow \quad \mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \qquad [!?]$$

where a linear cut on $x :!A$ reduces to an unrestricted cut on $x : A$. This cut-reduction is standard in dyadic presentations of linear logic [122].

Then, persistent session invocation by clients is modelled by rule [call]

$$\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{call}\ x(z);Q\} \quad \rightarrow \quad \mathbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ (\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\} \qquad [\text{call}]$$

A linear replica of the body P is instantiated on the fresh name z as $\{z/y\}P$. Notice that session x is still available to the continuation Q for (possibly) further calls.

The unrestricted cut distributes over all the static constructs, for example the distribution over mix is given by $\equiv$ rule

$$\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{par}\ \{Q\ ||\ R\})\} \equiv \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ R\}, x \notin \text{fn}(R)\ [\text{C!M}]$$

Computationally, when interpreted from left to right, it corresponds to the replication of $y.P$ among two clients Q and R. On the right-hand side each client interacts with its own replica in parallel.

The following defines some useful process encoding which are used throughout the text, first the unrestricted copycat forwarding construct and then the free output of an unrestricted name.

Definition 8 (Unrestricted Copycat $\mathbf{fwd}^!\ x\ y$)**.** *The unrestricted copycat process on names x and y is defined by*

$$\mathbf{fwd}^!\ x\ y \triangleq\ !x(x0); \mathbf{call}\ y(y0); \mathbf{fwd}\ x0\ y0$$

Definition 9 (Free Output of an Unrestricted Name)**.** *Free output on unrestricted names y is defined by* $\mathbf{send}\ x(y); P \triangleq \mathbf{send}\ x(w.\mathbf{fwd}^!\ w\ y); P$

Proposition 2. *The typing rules*

$$\frac{}{\mathbf{fwd}^!\ x\ y \vdash x :!A; \Gamma, y : \overline{A}}\ [TfwdE] \qquad \frac{P \vdash \Delta, x : B; \Gamma, y : \overline{A}}{\mathbf{send}\ x(y); P \vdash \Delta, x :!A \otimes B; \Gamma, y : \overline{A}}\ [T\otimes_{f!}]$$

are derivable.

Proof. Immediate from Definitions 8 and 9. □

The unrestricted copycat process $\mathbf{fwd}^!\ x\ y \vdash x :!A; \Gamma, y : \overline{A}$ defines a replicated session on session x that, on each invocation, redirects the call request to y. Process $\mathbf{send}\ x(y); P$ outputs the free unrestricted session $y : \overline{A}$ on session x and then continues as $x : B$, being implemented as a process that communicates on x a fresh session w implemented by the unrestricted copycat process.

We use the same process expression for the free output of a linear name (see Def. 7) and the free output of an unrestricted name. Nevertheless, it is always possible to disambiguate which process definition the expression stands for by analysing the context with which the expression types: the free output of a linear name $\mathbf{send}\ x(y)$; types a session y in the linear part of the context, whereas the free output of an unrestricted name $\mathbf{send}\ x(y)$; types a session y in the unrestricted part of the context.

Before moving on to present polymorphic and inductive type sessions, we show how all the basic process and type constructs presented so far can be combined to define a simple boolean calculator.

Example 5 (Boolean Calculator)**.** *The boolean calculator operates on the datatype* Bool *of boolean sessions, previously defined in Example 4.*

Fig. 2.5 lists code for the calculator calculator(s) *that, on request, either discards a boolean, computes the logical negation of an inputted boolean or the logical conjunction of two inputted booleans. Process* calculator(s) *offers on s a protocol of type* !Menu *where*

$$\begin{array}{rcl} \mathsf{Menu} \triangleq \&\{ & \mathsf{dis} & : \ \mathsf{Bool} \multimap \mathbf{1} \\ & \mathsf{neg} & : \ \mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathbf{1} \\ & \mathsf{conj} & : \ \mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathbf{1}\} \end{array}$$

On each call on channel s, calculator(s) *spawns a linear session on a fresh c (of type* Menu*). Then, on c, it offers a menu with three choices:* dis, neg *or* conj*. In case the choice is* dis*, it receives on c a boolean x and discards x. If* neg*, it receives on c boolean x, computes the logical negation of x on y and sends y on session c. Finally, if the chosen option is* conj*, the boolean calculator receives*

```
calculator(s) ≜
!s(c);
case c {
  |dis :  recv c(x);
          par {discard(x) || close c}
  |neg : recv c(x);
         send c(y.not(x, y));
         close c
  |conj :recv c(x);
         recv c(y);
         send c(z.and(x, y, z));
         close c
}
```

```
client1(s) ≜
  call s(c0);
  neg c0;
  send c0(x.false(x));
  recv c0(y);
  wait c0;
  call s(c1);
  dis  c1;
  send c1(y);
  wait c1;
  client'1(s)
```

```
client2(s) ≜
  call s(c);
  conj c;
  send c(x.true(x));
  send c(y.false(y));
  recv c(z);
  wait c;
  client'2(z, s)
```

Figure 2.5: Code for the boolean calculator and two clients.

on c two booleans x and y, computes their logical conjunction on z and sends back the result z on session c. Both branches finalise the session c by closing it, after which no further interaction takes place.

Fig. 2.5 presents code for two alternative clients of the boolean calculator both offering the same typing context:

$$\mathsf{client}_1(s) \vdash \emptyset; s : \overline{\mathsf{Menu}} \quad \mathsf{client}_2(s) \vdash \emptyset; s : \overline{\mathsf{Menu}}$$

Client client_1 *calls the replicated calculator twice: first to compute the logical negation of the boolean false and then to discard the result, after which it continues as* $\mathsf{client}'_1(s) \vdash s : \overline{\mathsf{Menu}}$. *Alternatively, client* client_2 *chooses option* conj *to compute the logical conjunction of booleans true and false and gets the result on z, continuing as* $\mathsf{client}'_2(z, s) \vdash z : \overline{\mathsf{Bool}}; s : \overline{\mathsf{Menu}}$. *Both client continuations can possibly make further invocations on the replicated calculator, since session s persists on the unrestricted context.*

Finally, we define a closed process $\mathsf{system} \vdash \emptyset; \emptyset$ *that composes the two clients, running in parallel, with the boolean calculator*

$$\mathsf{system} \triangleq \mathbf{cut}\ \{\mathsf{calculator}(s)\ \ |s : !\mathsf{Menu}|\ \ ?s; \mathbf{par}\ \{\mathsf{client}_1(s)\ ||\ \mathsf{client}_2(s)\}\}$$

The process construct $?x; P$ moves a session $x : ?A$ from the linear to the unrestricted context (to be typed as $x : A$) and proceeds as P. Independently of the order in which the calculator requests are processed, both clients will always get the same results. The calculator behaviour is identical for each invocation (cf. uniform receptiveness of shared names [136]). In the next chapter, we will present our language extension CLASS *that allows clients to share stateful objects, in which non-deterministic computations naturally emerge.*

A variant of this example is coded in examples/pure/boolean-server.clls.

Existential and Universal Abstraction

CLASS has support for existential ($\exists X.A$) and universal ($\forall X.A$) types, which allows us to define abstract datatypes [109] and generic polymorphic algorithms.

In session-based process calculi, abstraction is implemented by a process $\mathbf{sendty}\ x\ B; P$ that sends a representation type B along x, and then continues as Q. Symmetrically, process $\mathbf{recvty}\ x(X); Q$ receives on x a representation X and continues as Q.

Type send and receive are typed by $\exists X.A$ (exists) and $\forall X.\overline{A}$ (for all) respectively

$$\frac{P \vdash \Delta, x : \{B/X\}A; \Gamma}{\mathbf{sendty}\ x\ B; P\ \vdash \Delta, x : \exists X.A; \Gamma}\ [\mathrm{T}\exists] \qquad \frac{Q \vdash \Delta, x : \overline{A}; \Gamma}{\mathbf{recvty}\ x(X); Q \vdash \Delta, x : \forall X.\overline{A}; \Gamma}\ [\mathrm{T}\forall]$$

the rules correspond to second-order existential and universal quantification. In rule [T∀], we consider that X does not occur free in Δ, Γ.

The associated reduction between type send and receive is

$$\mathbf{cut}\ \{\mathbf{sendty}\ x\ B; P\ \ |x : \exists X.\ A|\ \ \mathbf{recvty}\ x(X); Q\} \quad \rightarrow \quad \mathbf{cut}\ \{P\ \ |x : \{B/X\}A|\ \ \{B/X\}Q\} \quad [\exists\forall]$$

The presence of existential quantifiers allow us to hide the representation datatype (cf. [109]) of our concurrent shareable stateful ADTs, as we will in the next chapter. On the other hand, with universal quantifiers we can express generic parametric processed and even inductive datatypes such as naturals and lists (see [161, 149]), as we illustrate in the following example, by encoding the Church numerals.

Example 6 (Inductive types)**.** *We illustrate the usage of type quantifiers to encode inductive types, by implementing the naturals with polymorphic sessions (cf. [66])*

$$\begin{aligned}
\mathsf{Nat} &\triangleq \forall X.X \multimap !(X \multimap X) \multimap X \\
\mathsf{zero}(n) &\triangleq \mathbf{recvty}\ n(X); \mathbf{recv}\ n(z); \mathbf{recv}\ n(s); ?s; \mathbf{fwd}\ z\ n \\
\mathsf{succ}(n, m) &\triangleq \mathbf{recvty}\ m(X); \mathbf{recv}\ m(z); \mathbf{recv}\ m(s); ?s; \\
&\qquad \mathbf{sendty}\ n\ X; \mathbf{send}\ n(z); \mathbf{send}\ n(s); \mathbf{call}\ s(c); \mathbf{send}\ c(n); \mathbf{fwd}\ c\ m
\end{aligned}$$

Terms of type Nat *receive a type variable* X*, a value* $z : X$ *and a persistent session* $s :!(X \multimap X)$ *and call the* s *a finite (possibly zero) number of times on* z *to return a value* $n : X$*. Notice that* $\mathsf{zero}(n) \vdash n : \mathsf{Nat}$ *simply forwards* z *on* n *without calling* s*. On the other hand,* $\mathsf{succ}(n, m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ *calls* s *one more time on the output produced by the calls of* n*. The encoding of the naturals allows for the definition of recursive operations. For example, the predicate* $\mathsf{zero?}(n, b) \vdash n : \overline{\mathsf{Nat}}, b : \mathsf{Bool}$ *that consumes a natural* n *and produces the boolean true if* n *is zero and false otherwise is defined by*

$$\mathsf{zero?}(n, b) \triangleq \mathbf{sendty}\ n\ \mathsf{Bool}; \mathbf{send}\ n(z.\ \mathsf{true}\ z); \mathbf{send}\ n(s.\ \mathsf{kfalse}(s)); \mathbf{fwd}\ n\ b$$

where $\mathsf{kfalse}(s) \vdash s :!(\mathsf{Bool} \multimap \mathsf{Bool}); \Gamma$ *is a process that persistently continues as the boolean false regardless of the input*

$$\mathsf{kfalse}(s) \triangleq\ !s(s0); \mathbf{recv}\ s0(b); \mathbf{case}\ b\ \{|\mathsf{inl} : \mathbf{wait}\ b; \mathsf{false}(s0)\ |\ \mathsf{inr} : \mathbf{wait}\ b; \mathsf{false}(s0)\}$$

Check pure/naturals-systemF.clls *for an implementation of this example, where we also define more functions such as* $\mathsf{equal}(n : \overline{\mathsf{Nat}}, m : \overline{\mathsf{Nat}}, b : \mathsf{Bool})$ *that checks for the equality of two naturals. Check also* pure/lists-systemF.clls *for an implementation of lists and*

pure/recursion-for-free.clls *for how to encode recursive types using polymorphism, based on [161].*

Inductive/Coinductive Session Types

In the previous subsection we saw that polymorphism allows us to encode a form of recursion *for free*, in the style of system-F encodings. However, this encoding cannot be extended to accommodate imperative stateful recursive structures, such as linked lists, with support for memory-efficient updates in-place. This is because the recursive iterators obtained with polymorphism destroy the recursive structure over which the iteration is performed.

This has motivated the introduction of inductive $\mu X.\, A$ and coinductive $\nu X.\, A$ session types in CLASS. Coinductive sessions are introduced by rule [Tcorec]

$$\frac{P \vdash_{\eta'} \Delta, z : A; \Gamma \quad \eta' = \eta, X(z, \vec{w}) \mapsto \Delta, z : Y; \Gamma}{\textbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}] \vdash_{\eta} \{\vec{y}/\vec{w}\}\Delta, x : \nu Y.\, A; \{\vec{y}/\vec{w}\}\Gamma}\ [\text{Tcorec}]$$

It types a corecursive process $\textbf{corec}\ X(z, \vec{w}); P\ [x, \vec{y}]$ with parameters $z, \vec{w}$ bound in body P, which is instantiated with arguments $x, \vec{y}$. By convention, the coinductive behaviour is always offered in the first argument z of the corecursive process.

To type the body $P \vdash_{\eta'} \Delta, z : A; \Gamma$ of a corecursive process we extend the map η with a binding for a process variable X mapped to the typing context $\Delta, z : Y; \Gamma$, so that when typing the body P of the corecursion we can appeal to X, which intuitively stands for P itself, and recover its typing invariant.

The type variable Y is free only in $z : A$, when typing the body P. This guarantees that the corecursive call is done in a session that hereditarily descends from the corecursive argument z, thereby ensuring that corecursion is well-founded, which is then essential to have strong normalisation.

Typing rule [Tvar]

$$\frac{\eta = \eta', X(x, \vec{y}) \mapsto \Delta, x : Y; \Gamma}{X(z, \vec{w}) \vdash_{\eta} \{\vec{w}/\vec{y}\}(\Delta, z : Y; \Gamma)}\ [\text{Tvar}]$$

allows us to type a recursive call $X(x, \vec{y})$ by looking up in η for the corresponding binding and by renaming the parameters of the binding with the arguments of the call.

On the other hand, type $\nu X.\, A$ allows us to type recursive process definitions, as we illustrate in Example 7 by encoding the natural datatype. Both inductive and coinductive types can be unfolded, as typed by rules

$$\frac{P \vdash_{\eta} \Delta, x : \{\nu X.\, A/X\}A; \Gamma}{\textbf{unfold}_{\nu}\ x; P \vdash_{\eta} \Delta, x : \nu X.\, A; \Gamma}\ [\text{T}\nu] \qquad \frac{P \vdash_{\eta} \Delta, x : \{\mu X.\, A/X\}A; \Gamma}{\textbf{unfold}_{\mu}\ x; P \vdash_{\eta} \Delta, x : \mu X.\, A; \Gamma}\ [\text{T}\mu]$$

Process $\textbf{unfold}_{\mu}\ x; P$ (resp., $\textbf{unfold}_{\nu}\ x; P$) unfolds the inductive type $x : \mu X A$ (resp., coinductive type $\nu X.\, A$) and continues as P.

The cut between the unfold of an inductive type and a corecursive process definition reduces to a process in which the corecursive definition is unfolded by one step, as modelled by

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{unfold}_{\mu}\ x; P\ \ |x : \mu X.\ A|\ \ \mathbf{corec}\ Y(z, \vec{w}); Q\ [x, \vec{y}]\}\\ &\quad\rightarrow \mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ \{x/z\}\{\vec{y}/\vec{w}\}\{\mathbf{corec}\ Y(z, \vec{w}); Q/Y\}Q\}\ [\text{corec}]\end{aligned}$$

A cut between the unfold of an inductive type and the unfold of a coinductive type simply reduces to a cut of the continuations as modelled by

$$\mathbf{cut}\ \{\mathbf{unfold}_{\mu}\ x; P\ \ |x : \mu X.\ A|\ \ \mathbf{unfold}_{\nu}\ x; Q\} \rightarrow \mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ Q\}\ [\mu\nu]$$

To simplify the presentation of examples, we omit unfolding actions (typed by $[\mathrm{T}\mu]$ and $[\mathrm{T}\nu]$) and write corecursive processes in the form $Q(x, \vec{y}) \triangleq \ldots Q(-) \ldots$ instead of $\mathbf{corec}\ X(z, \vec{w}); \ldots X(-) \ldots\ [x, \vec{y}]$, while of course respecting the typing rules.

We conclude this subsection with two examples: in 7 we encode naturals and define some operations using primitive inductive/coinductive session types; in 8 we show general recursion loops.

Example 7 (Naturals)**.** *Previously, in Example 6 we defined the naturals in the style of System F encodings by resorting to polymorphic session types. In this example we give a direct encoding by relying on our primitive inductive/coinductive session types.*

Let

$$\mathsf{Nat} \triangleq \mu X.\ \oplus\{|\mathsf{Z} : \mathbf{1}\ |\mathsf{S} : X\}$$

A session $x : \mathsf{Nat}$ *chooses either* Z *(zero), in which case it closes, or* S *(successor) in which case it recurs as* $x : \mathsf{Nat}$*.*

Processes $V_0(n), V_1(n), V_2(n) \ldots \vdash n : \mathsf{Nat}$ *corresponding to the naturals* $0, 1, 2, \ldots$ *can be encoded as*

$$\begin{aligned}V_0(n) &\triangleq \mathsf{Z}\ n; \mathbf{close}\ n\\ V_1(n) &\triangleq \mathsf{S}\ n; \mathsf{Z}\ n; \mathbf{close}\ n\\ V_2(n) &\triangleq \mathsf{S}\ n; \mathsf{S}\ n; \mathsf{Z}\ n; \mathbf{close}\ n\\ &\ldots\end{aligned}$$

The idea can be generalised to define process $\mathsf{succ}(n, m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ *which interacts with a natural on* n *and produces its successor on* m *by*

$$\mathsf{succ}(n, m) \triangleq \mathsf{S}\ m; \mathbf{fwd}\ n\ m$$

We will now illustrate corecursion by defining a process that doubles a natural number, but first we defined the following auxiliary process $\mathsf{add2}(n, m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ *that produces on* m *the result of adding* 2 *to* n

$$\mathsf{add2}(n, m) \triangleq \mathsf{S}\ m; \mathsf{S}\ m; \mathbf{fwd}\ n\ m$$

We can rewrite process $\mathsf{add2}(n, m)$ *with explicit unfolds as*

$$\mathsf{add2}(n, m) = \mathbf{unfold}_{\mu}\ m; \mathsf{S}\ m; \mathbf{unfold}_{\mu}\ m; \mathsf{S}\ m; \mathbf{fwd}\ n\ m$$

The type derivation for $\mathsf{add2}(n, m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ *is then as follows*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{fwd}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}}\,[Tfwd]}{\mathsf{S}\ m; \mathbf{fwd}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \oplus\{|\mathsf{Z} : \mathbf{1}\ |\mathsf{S} : \mathsf{Nat}\}}\,[T\oplus]}{\mathbf{unfold}_\mu\ m; \mathsf{S}\ m; \mathbf{fwd}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}}\,[T\mu]}{\mathsf{S}\ m; \mathbf{unfold}_\mu\ m; \mathsf{S}\ m; \mathbf{fwd}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \oplus\{|\mathsf{Z} : \mathbf{1}\ |\mathsf{S} : \mathsf{Nat}\}}\,[T\oplus]}{\mathbf{unfold}_\mu\ m; \mathsf{S}\ m; \mathbf{unfold}_\mu\ m; \mathsf{S}\ m; \mathbf{fwd}\ n\ m \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}}\,[T\mu]$$

In the type derivation, the recursive type Nat *is unfolded twice, so that we can exhibit the type choice and select the branch* S*, corresponding to successor.*

With corecursion we can define process $\mathsf{double}(n, m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$ *produces the double of* n *on* m*:*

$$\begin{array}{rcl} \mathsf{double}(n, m) & \triangleq & \mathsf{case}\ n\{\ \ |\mathsf{Z} :\ \mathbf{wait}\ n; \mathsf{V}_0(m) \\ & & \qquad\quad |\mathsf{S} :\ \mathbf{cut}\ \{\mathsf{double}(n, k)\ \ |k|\ \ \mathsf{add2}(k, m)\}\} \end{array}$$

$\mathsf{double}(n, m)$ *starts by pattern matching on* n*: if* n *chooses* Z *(zero) we simply wait for* n *to be closed and continues as the natural* $\mathsf{V}_0(m)$ *on* m*. Otherwise: if* n *chooses* S *(successor) we spawn a corecursive call* $\mathsf{double}(n, k)$ *that interacts with its continuation and produces its double in* k*. The corecursive call is composed via a cut on* k *with process* $\mathsf{add2}(k, m)$*.*

We will now show how the corecursive definition $\mathsf{double}(n, m)$ *maps to the core language syntax with the explicit annotation of corecursion, by rewriting* $\mathsf{double}(n, m)$ *as*

$$\mathsf{double}(n, m) \quad \triangleq \quad \mathbf{corec}\ Y(x, y); C(x, y)\ [n, m]$$

where the auxiliary case process $C(x, y)$ *is defined by*

$$\begin{array}{rcl} C(x, y) & \triangleq & \mathsf{case}\ x\{\ \ \mathsf{Z} :\ \mathbf{wait}\ x; \mathsf{V}_0(y) \\ & & \qquad\quad \mathsf{S} :\ \mathbf{cut}\ \{Y(x, z)\ \ |z|\ \ \mathsf{add2}(y, z)\}\} \end{array}$$

We will now show the type derivation for $\mathsf{double}(n, m)$*. The corecursive construct extends map* η *with a binding for the variable* Y

$$\dfrac{\dfrac{\vdots}{C(x, y) \vdash_{\eta'} x : \bot \,\&\, X, y : \mathsf{Nat} \quad \eta' = \eta, Y(x, y) \mapsto \Delta, x : X, y : \mathsf{Nat}}\,[T\&]}{\mathbf{corec}\ Y(x, y); P(x, y)\ [n, m] \vdash_\eta n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}}\,[Tcorec]$$

We complete the type derivation by displaying the derivation for each branch of the case, namely Z *(zero)*

$$\dfrac{\mathsf{V}_0(y) \vdash_{\eta'}\ y : \mathsf{Nat}}{\mathbf{wait}\ x; \mathsf{V}_0(y) \vdash_{\eta'} x : \bot, y : \mathsf{Nat}}\,[T\bot]$$

and S *(successor)*

$$\dfrac{\dfrac{\eta' = \eta, Y(x,y) \mapsto \Delta, x : X, y : \mathsf{Nat}}{Y(x,z) \vdash_{\eta'} x : X, z : \mathsf{Nat}}\ [Tvar] \qquad \mathsf{add2}(y,z) \vdash_{\eta'} z : \overline{\mathsf{Nat}}, y : \mathsf{Nat}}{\mathbf{cut}\ \{Y(x,z)\ \ |z|\ \ \mathsf{add2}(y,z)\} \vdash_{\eta'} x : X, y : \mathsf{Nat}}\ [Tcut]$$

where $\eta' = \eta, Y(x,y) \mapsto \Delta, x : X, y : \mathsf{Nat}$.

This example is coded in pure/naturals.clls, *where we also define further corecursive process definitions such as the arithmetic operations of addition and multiplication, as well as some tests.*

Example 8 (Repeat)**.** *In this example we show how to code general recursion loops. We code process*

$$\mathsf{repeat}(n,i,o,b) \vdash n : \overline{\mathsf{Nat}}, i : \overline{A}, o : A; \Gamma, b : \overline{(A \multimap A)}$$

that repeats n *times the replicated session* b*, starting with* i *and producing the output on* o*. The definition is as follows*

$$\begin{array}{rcll} \mathsf{repeat}(n,i,o,b) & \triangleq & \mathbf{case}\ n\{ & \mathsf{Z} : \mathbf{wait}\ n; \mathbf{fwd}\ i\ o \\ & & & \mathsf{S} : \mathbf{call}\ b(x); \mathbf{send}\ x(i); \mathsf{repeat}(n,x,o,b)\} \end{array}$$

Process $\mathsf{repeat}(n,i,o,b)$ *performs case analysis on* n*. If* Z *(zero), it waits for* n *to be closed and then simply forwards the initial input* i *to the output* o*. On the other hand, if* S*, it calls the replicated session* b *on* x*, sends the initial input* i *on* x *and then continues as tail corecursive call* $\mathsf{repeat}(n,x,o,b)$*.*

By using repeat *and process* double*, that doubles a natural number (from previous Example 7), we implement*

$$\mathsf{pow}_2(n,m) \vdash n : \overline{\mathsf{Nat}}, m : \mathsf{Nat}$$

that computes the exponent of base 2 *of* n *and outputs on* m*, defined by*

$$\mathsf{pow}_2(n,m) \triangleq \mathbf{cut}\ \{\mathsf{V}_1(i)\ \ |i|\ \ \mathbf{cut}\ \{?b; \mathsf{repeat}(n,i,m,b)\ \ |b|\ \ !b(k); \mathbf{recv}\ k(j); \mathsf{double}(j,k)\}\}$$

Process $\mathsf{pow}_2(n,m)$ *composes* $\mathsf{repeat}(n,i,m,b)$ *with two processes: natural number* 1 *on session* i*, that represents the initial value of the iteration; and the replicated body* b *that, on each call, doubles the inputted natural.*

Process repeat *allows us to code the for-loop used in the introduction to implement the fork-join 1.4. Given any* $P \vdash \mathsf{spre} : \overline{A}, \mathsf{spost} : A; \Gamma, \mathsf{max} : \overline{\mathsf{Nat}}$*, we define*

$$\begin{array}{l} \mathbf{for}\ \ \mathsf{spre} : A\ \ \mathbf{to}\ \mathsf{spost}(s,s',\mathsf{max})\{P\} \\ \triangleq \mathbf{cut}\ \{!b(\mathsf{spost}); \mathbf{recv}\ \mathsf{spost}(\mathsf{spre}); P\ \ |b|\ \ ?b; \mathbf{call}\ \mathsf{max}(n); \mathsf{repeat}(n,s,s',b)\} \end{array}$$

2.4 Further Discussion and Related Work

This chapter presented the session-typed language μCLL, based on the propositions-as-types interpretation of second-order classical linear logic [23, 24, 157], extended with mix and inductive / coinductive types.

The notion of propositions-as-types goes back to the functional interpretation of Intuitionistic logic due to Brouwer, Heyting and Kolmogorov, but was only brought under the spotlight after the famous notes of Curry and Howard [76]. It has been since then considered both an intriguing and prolific concept, with many instances and consequences (see [160]).

Linear logic was developed by Girard [61] and it was advertised as the logic for concurrency. Some initial steps by Abramsky [1] and Bellin and Scott [14] were taken to connect linear logic to Milner's π-calculus [108]. However, a propositions-as-types correspondence between linear logic and session types [73] was only established a decade ago by Caires and Pfenning [23] and Wadler [157], on which we based the development of μCLL.

In [23] the authors propose a type system, named πDILL, for a synchronous π-calculus that corresponds, by erasing process annotations, exactly to a sequent-calculus formulation of dual intuitionistic linear logic. They prove the existence of a bidirectional simulation between (typed) π-calculus computations and proof conversions on typing derivations, thereby establishing a strong form of subject reduction.

Following [23], the work [157] presents a session-typed process calculus, named CP, that corresponds to classical linear logic and formalises the connection with session types by translating a linear functional language with session types, inspired by [58], to CP. Independently, in [24], the author define a variant of the interpretation [23] for classical linear logic.

The underlying calculus of both πDILL and CP are variants of Milner's π-calculus. The π-calculus has, among others, process constructors νx for name restriction and $\mid$ for parallel composition, which do not find a direct correspondence in linear logic. In fact, those constructors are plugged together in πDILL to obtain the process $\nu x\ (P \mid Q)$ that interprets the cut rule and corresponds to interactive composition. In contrast, in our language μCLL, interactive composition has an indivisible process construct $\|_x$ of its own. Interestingly, this indivisible kind of interactive composition (parallel composition plus hiding) is the basic morphism composition of the work on interactive categories [2] and it was even considered by Milner as a candidate primitive operation of the π-calculus [105]. Those and other discrepancies between the π-calculus and linear logic proof syntax pose some problems for defining an observable semantics, which were recently investigated in [90].

The type system of μCLL is dyadic. Dyadic formulations of logic can be found in the unified sequents of Girard [65], where there is *a zone with a classical maintenance, and a zone with a linear maintenance*. And they can also be found in the work on focusing

proofs of Andreoli [5]. Whereas in monadic presentations of CLL [63], weakening and contraction are explicit rules of the typing system, in dyadic formulation the same rules are admissible. As we shall see when studying proof normalisation, and as explored by Pfenning [122], isolating the non-linear reasoning in a separated context eases the task of building constructive proofs of cut-elimination. As far as process interpretations of linear logic are concerned, dyadic presentations can be found in [23] for the intuitionistic and in [29] for the classical variants. Wadler [157] adopts a monadic presentation.

Least and greatest fixed points were studied in the context of linear logic by Baelde [9], which inspired the development of recursion in session-based propositions-as-types interpretations [147, 99]. Our treatment of recursion in μCLL is inspired by [147, 143], where recursion/corecursion is added on top of the Curry-Howard interpretation of intuitionistic linear logic [23]. Also based on [23], the work [47] develops a session-typed process interpretation with fixed points for a subsingleton fragment of intuitionistic linear logic, in which the antecedent of each sequent consists of at most one formula.

3

CLASS: Classic Linear Logic with Affine Shared State

3.1 Introduction

In this chapter, we conclude the presentation of CLASS by extending the pure fragment μCLL with affine sessions, first-class higher-order reference cells, shared state and nondeterminism. This allows rich imperative stateful programs to be coded and typed, such as linked data structures, resource synchronisation methods and shared mutable concurrent ADTs, to name a few. Some examples were already presented in the introductory overview, in this chapter we revisit some of them and then, in the next chapter, we will code several more.

CLASS fine-grained lightweight type system, based on linear logic, imposes crisp conditions, guaranteeing that all cell operations are well-behaved and that the state sharing topologies are acyclic and also implies strong guarantees like memory safety, deadlock-freedom and termination. Furthermore, because we follow a propositions-as-types approach, in CLASS we can reason about the behaviour of concurrent programs that manipulate shared state, by doing simple algebraic-like manipulations on process expressions.

CLASS has a rich substructural type system: besides linear and unrestricted sessions, it also offers support for affine sessions. The process construct

$$\textbf{affine}\ a; P$$

types with the affine modality $\wedge A$ and defines an affine session on a, it then continues as P. Affine sessions can either be used or discarded, with the operations

$$\textbf{use}\ a; P \qquad \textbf{discard}\ a$$

that type with the dual coaffine modality $\vee A$. Besides being useful for expressing general resourceful programs, affine sessions play a key role in CLASS, since the sessions stored by reference cells are affine. This is an essential condition to guarantee memory-safe deallocation, which is handled automatically in CLASS.

In CLASS, reference cells can either be full or empty, as expressed by process constructs

$$\textbf{cell}\ c(a.P) \quad \textbf{empty}\ c$$

The process on the left-hand side defines a full reference cell on session c, which stores an affine session a, the behaviour on a being implemented by P. On the other hand, the right-hand side defines an empty cell on session c. Both cells type with a state modality, but CLASS fine-grained type system distinguishes between the two full and empty states, expressed by the $\mathbf{S}_f\ A$ (state full) and $\mathbf{S}_e\ A$ (state empty) modalities, respectively.

There are three possible operations on CLASS reference cells

$$\textbf{release}\ c \quad \textbf{take}\ c(a); Q \quad \textbf{put}\ c(a.Q_1); Q_2$$

Processes **release** c and **take** $c(a); Q$ can be applied to full reference cells, they type with the modality $\mathbf{U}_f\ A$ (full usage), the former releases the cell usage whereas the latter takes the session a stored in the reference cell c, then continues as Q. The take operation acquires the implicit lock associated with the reference cell c, the type system then guarantees that this lock is released, so as to ensure the absence of deadlocks. This lock-releasing is done with the put operation **put** $c(a.Q_1); Q_2$, which applies to empty reference cells, it types with the modality $\mathbf{U}_e\ A$ (empty usage). It puts back a new session a, the behaviour of which is implemented by Q_1, and continues as Q_2.

The reference cells and their sequential cell usage operations allows us to express rich efficient data structures in CLASS, such as mutable pairs, linked lists and binary search trees, without violating important safety properties such as termination. We will show some of these examples later, interestingly they illustrate how primitive recursion fits well with higher-order state in CLASS.

CLASS also offers support for state sharing, with the construct

$$\textbf{share}\ c\ \{P \parallel Q\}$$

and which allows us to dynamically share a cell usage c by an arbitrary number of cooperative concurrent threads. With sharing at our disposal, we can code expressive fine-grained concurrent programs, such as shared concurrent mutable ADTs (counters, stacks, functional and imperative queues), resource synchronisation methods (fork-joins, barriers and dining philosophers) and generic corecursive protocols.

The type system of CLASS lifts the basic acyclicity of linear logic session communication topologies, expressed by the cut rule, to cell sharing topologies, in this case expressed logically by cocontraction. This guarantees that CLASS well-typed programs dot not block, neither on channel communication, nor on cell-lock acquisition, are memory-safe and terminate. A guarantee that holds even for complex structures involving recursion, polymorphism and higher-order state, with dynamic unbounded sharing. Furthermore, these properties follow from the basic linear type system, without needing to resort to extra-logical devices such as partial orders.

Sharing of stateful reference cells leads to nondeterminism, which naturally emerge from racy concurrent behaviour, specifically from concurrent take operations. In CLASS this nondeterminism is captured by a non-collapsing sum operation

$$P + Q$$

which represents a choice between two alternatives P and Q.

Sums play a key role in the metatheoretical model of CLASS as they allows us to explicitly capture all the possible nondeterministic evolutions of a stateful system and, because of that, to have a confluent notion of reduction, as required from a propositions-as-types approach. As will see, with sums we can reason about the behaviour of concurrent stateful programs equationally by doing simple algebraic-like manipulations, which paves the way for program optimisations.

This chapter is organised as follows. Section 3.2 presents the process calculus CLASS and its operational semantics. Then, in Section 3.3, we present the type system going though each of the process constructs, associated typing and reduction rules. The introduced concepts are illustrated with several examples. Finally, Section 3.4 concludes with further discussion and related work.

3.2 Process Calculus and Operational Semantics

In this section, we define the process syntax and the reduction-based operational semantics of CLASS, extending the corresponding elements of the pure fragment μCLL (Chapter 2, Section 2.2).

Types and Duality

Since processes in CLASS depend on types, we start with types and type duality.

Definition 10 (Types)**.** *We extend the grammar of types of* μCLL *(Definition 1) with*

$$\begin{array}{lll} A, B ::= & \ldots & \\ & \mid \wedge A \text{ (affine)} & \mid \vee A \text{ (coaffine)} \\ & \mid \mathbf{S}_f\ A \text{ (full state)} & \mid \mathbf{U}_f\ A \text{ (full usage)} \\ & \mid \mathbf{S}_e\ A \text{ (empty state)} & \mid \mathbf{U}_e\ A \text{ (empty usage)} \end{array}$$

The types of CLASS extend the propositions of classical linear logic with type modalities for affine sessions ($\wedge A$, $\vee A$) and for imperative state and state usage ($\mathbf{S}_f\ A$, $\mathbf{U}_f\ A$, $\mathbf{S}_e\ A$,$\mathbf{U}_e\ A$). We consider that affine/coaffine modalities and the state/usage modalities have higher precedence than the binary connectives.

The following definition extends duality to the newly introduced modalities.

$$
\begin{array}{rcl}
P, Q & ::= & \ldots \mid \\
 & & \textbf{share}\ c\ \{P \parallel Q\}\ \textit{(share)} \\
 & & P + Q\ \textit{(sum)} \\
\mathcal{A}, \mathcal{B} & ::= & \ldots \mid \\
 & & \mid \textbf{affine}_{\vec{c},\vec{a}}\ a; P\ \textit{(affine)} \\
 & & \mid \textbf{discard}\ a\ \textit{(discard)} \\
 & & \mid \textbf{use}\ a; P\ \textit{(use)} \\
 & & \mid \textbf{cell}\ c(a.P)\ \textit{(cell)} \\
 & & \mid \textbf{release}\ c\ \textit{(release)} \\
 & & \mid \textbf{take}\ c(a); P\ \textit{(take)} \\
 & & \mid \textbf{empty}\ c\ \textit{(empty cell)} \\
 & & \mid \textbf{put}\ c(a.P); Q\ \textit{(put)}
\end{array}
$$

Figure 3.1: Processes P of CLASS (extends Fig. 2.1).

Definition 11 (Duality on Types $\overline{A}$)**.** *We extend the definition of duality of* μCLL *(Definition 2) with*

$$
\begin{array}{lcl}
\overline{\wedge A} & \triangleq & \vee \overline{A} \\
\overline{S_f\ A} & \triangleq & U_f\ \overline{A} \\
\overline{S_e\ A} & \triangleq & U_e\ \overline{A}
\end{array}
$$

Processes

We will now present the processes of CLASS.

Definition 12 (Processes)**.** *The syntax of process terms for* CLASS *extends the syntax of* μCLL *(Definition 3) with the rules of Fig. 3.1.*

The imperative fragment of CLASS contains affine sessions $\textbf{affine}_{\vec{c},\vec{a}}\ a; P$ and operations to discard $\textbf{discard}\ a$ and use $\textbf{use}\ a; P$ a coaffine session. CLASS also defines full reference cells $\textbf{cell}\ c(a.P)$ which can be either released with $\textbf{release}\ c$ or acquired with a take operation $\textbf{take}\ c(a); P$, as well as empty reference cells $\textbf{empty}\ c$ which support a put operation $\textbf{put}\ c(a.P); Q$.

Besides the static constructs cut and mix, CLASS also has an operation $\textbf{share}\ c\ \{P \parallel Q\}$ to share state and nondeterministic sums $P + Q$. For convenience, we equip CLASS with n-ary right-associative share constructs

$$\textbf{share}\ \{P_1 \parallel \ldots\ P_{n-1} \parallel P_n\}$$

In the process expression $\textbf{affine}_{\vec{c},\vec{a}}\ a; P$ we let $\vec{c}$ and $\vec{a}$ be possible empty arrays of session names, which explicitly capture the dependencies of an affine session on other

Commutativity Rules

$\mathbf{share}\ x\ \{P \parallel Q\} \equiv \mathbf{share}\ x\ \{Q \parallel P\}$ [Sh]

Sum Rules

$\mathbf{0} + \mathbf{0} \equiv \mathbf{0}$ [0Sm] $\quad P + Q \equiv Q + P$ [Sm] $\quad P + (Q + R) \equiv (P + Q) + R$ [SmSm]

Linearly Distributive Conversions

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{share}\ y\ \{Q \parallel R\}\} \equiv \mathbf{share}\ y\ \{\mathbf{cut}\ \{P\ \ |x|\ \ Q\} \parallel R\},\ x, y \in \mathrm{fn}(Q)$ [CSh]

$\mathbf{share}\ x\ \{P \parallel \mathbf{cut!}\ \{y.Q\ |z|\ R\}\} \equiv \mathbf{cut!}\ \{y.Q\ |z|\ \mathbf{share}\ x\ \{P \parallel R\}\},\ z \notin \mathrm{fn}(P)$ [ShC!]

$\mathbf{share}\ x\ \{P \parallel (\mathbf{par}\ \{Q \parallel R\})\} \equiv \mathbf{par}\ \{\mathbf{share}\ x\ \{P \parallel Q\} \parallel R\},\ x \in \mathrm{fn}(Q)$ [ShM]

$\mathbf{share}\ x\ \{P \parallel \mathbf{share}\ y\ \{Q \parallel R\}\} \equiv \mathbf{share}\ y\ \{\mathbf{share}\ x\ \{P \parallel Q\} \parallel R\},\ x, y \in \mathrm{fn}(Q)$ [ShSh]

Sum Distributive Conversions

$\mathbf{par}\ \{P \parallel (Q + R)\} \equiv (\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\})$ [MSm]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ (Q + R)\} \equiv (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}) + (\mathbf{cut}\ \{P\ \ |x : A|\ \ R\})$ [CSm]

$\mathbf{cut!}\ \{y.P\ |x : A|\ (Q + R)\} \equiv (\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}) + (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})$ [C!Sm]

$\mathbf{share}\ x\ \{P \parallel (Q + R)\} \equiv (\mathbf{share}\ x\ \{P \parallel Q\}) + (\mathbf{share}\ x\ \{P \parallel R\})$ [ShSm]

Share Conversions

$\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel \mathbf{take}\ x(y_2); P_2\}$
$\equiv \mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \parallel \mathbf{take}\ x(y_2); P_2\} + \mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel P_2\},$
$y_1 \in \mathrm{fn}(P_1),\ y_2 \in \mathrm{fn}(P_2)$ [TSh]

$\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel P\} \equiv P$ [RSh]

$\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \parallel R\} \equiv \mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \parallel R\}$ [PSh]

Figure 3.2: Structural congruence $P \equiv Q$ Rules of CLASS (extends 2.2).

coaffine sessions $\vec{a}$ and full cell usages $\vec{c}$. Sometimes we omit them and simply write $\mathbf{affine}\ a; P$.

Actions $\mathbf{cell}\ c(a.P)$, $\mathbf{take}\ c(a); P$ and $\mathbf{put}\ c(a.P); Q$ of Fig. 3.1 bind a on P. All other name occurrences are free. In particular, the sharing construct $\mathbf{share}\ c\ \{P \parallel Q\}$ *does not* bind c, it occurs free in the process expression.

Operational Semantics

We will now present the reduction-based operational semantics of CLASS by extending the operational model for the pure fragment. We start with the definition of structural congruence.

Definition 13 (Structural Congruence $P \equiv Q$)**.** *Structural congruence* $\equiv$ *for* CLASS *extends the definition of structural congruence for* μCLL *(Def. 4) with the rules listed in Figure 3.2. In rule [FSh], we consider that whenever* $P \vdash_\eta \Delta; \Gamma$*, then* $\Delta = \Delta', x : A$ *where* $A = \mathbf{U}_f\ B$ *or* $A = \mathbf{U}_e\ B$*.*

$$\textbf{cut}\ \{\textbf{affine}_{\vec{c},\vec{a}}\ a; P\ \ |a|\ \ \textbf{discard}\ a\} \longrightarrow \textbf{par}\ \{\textbf{release}\ \vec{c}\ \|\ \textbf{discard}\ \vec{a}\} \quad [\wedge\vee\text{d}]$$

$$\textbf{cut}\ \{\textbf{affine}_{\vec{c},\vec{a}}\ a; P\ \ |a|\ \ \textbf{use}\ a; Q\} \longrightarrow \textbf{cut}\ \{P\ \ |a|\ \ Q\} \quad [\wedge\vee\text{u}]$$

$$\textbf{cut}\ \{\textbf{cell}\ c(a.P)\ \ |c|\ \ \textbf{release}\ c\} \longrightarrow \textbf{cut}\ \{P\ \ |a|\ \ \textbf{discard}\ a\} \quad [\mathbf{S}_f\ \mathbf{U}_f\ \text{f}]$$

$$\textbf{cut}\ \{\textbf{cell}\ c(a.P)\ \ |c|\ \ \textbf{take}\ c(a'); Q\} \longrightarrow \textbf{cut}\ \{P\ \ |a|\ \ \textbf{cut}\ \{\textbf{empty}\ c\ \ |c|\ \ \{a/a'\}Q\}\} \quad [\mathbf{S}_f\ \mathbf{U}_f\ \text{t}]$$

$$\textbf{cut}\ \{\textbf{empty}\ c\ \ |c|\ \ \textbf{put}\ c(a.P); Q\} \longrightarrow \textbf{cut}\ \{\textbf{cell}\ c(a.P)\ \ |c|\ \ Q\} \quad [\mathbf{S}_e\ \mathbf{U}_e\ ]$$

Figure 3.3: Reduction $P \longrightarrow Q$ rules of CLASS (Extends 2.3).

Share and sum are commutative ([Sh] and [Sm]), furthermore sum is associative ([SmSm]). The inaction process $\mathbf{0}$ is an idempotent element of sum ([0Sm]).

We can commute a (linear or unrestricted) cut, mix and share construct with a share (rules [CSh], [ShC!], [ShM] and [ShSh]). Furthermore, we can distribute all the static constructs mix, linear cut, unrestricted cut and share over sum (rules [MSm], [CSm], [C!Sm], [ShSm]).

A share with two concurrent take actions can be $\equiv$-equivalently rewritten into a sum in which the two concurrent take operations are interleaved, according to rule [TSh]. The release operation is a unit w.r.t to share-composition ([RSh]) and a put action commutes with a share construct ([PSh]).

We will now introduce the reduction relation $\longrightarrow$, but first we need a couple of definitions. First, we extend the static contexts of the pure fragment μCLL with

$$C ::= \quad \ldots \mid \textbf{share}\ x\ \{C \parallel P\} \mid \textbf{share}\ x\ \{P \parallel C\} \mid C + P \mid P + C$$

which allows to reduce processes within share and sum constructs. We also define the following auxiliary operations **release** $\vec{x}$ and **discard** $\vec{x}$, by induction on $\vec{x}$:

$$\begin{array}{llll} \textbf{release}\ [] & \triangleq \mathbf{0} & \textbf{release}\ (\vec{x} : y) & \triangleq \textbf{par}\ \{(\textbf{release}\ \vec{x}) \parallel \textbf{release}\ y\} \\ \textbf{discard}\ [] & \triangleq \mathbf{0} & \textbf{discard}\ (\vec{x} : y) & \triangleq \textbf{par}\ \{(\textbf{discard}\ \vec{x}) \parallel \textbf{discard}\ y\} \end{array}$$

which are used to model the interaction between an affine and discard process.

We can present the reduction relation.

Definition 14 (Reduction $P \longrightarrow Q$)**.** *Reduction $\longrightarrow$ for* CLASS *extends the definition of reduction for μ*CLL *(Def. 5) with the rules listed in Fig. 3.3.*

The structural congruence rules of Def. 13 and the reduction rules of Def. 14 are *complete and correct* in the sense of being sufficient and necessary to establish the metatheoretical properties of type preservation, progress, confluence and normalisation, furthermore they have an intuitive reading as behavioural identities on concurrent processes that manipulate shared mutable state, we will go though all of them in detail in the following section.

$$\frac{P \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, a : A; \Gamma}{\mathbf{affine}_{\vec{c},\vec{a}}\ v; P \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, a : \wedge A; \Gamma}\ [\text{Taffine}]$$

$$\frac{}{\mathbf{discard}\ a \vdash_\eta a : \vee A; \Gamma}\ [\text{Tdiscard}] \qquad \frac{Q \vdash_\eta \Delta, a : A; \Gamma}{\mathbf{use}\ a; Q \vdash_\eta \Delta, a : \vee A; \Gamma}\ [\text{Tuse}]$$

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\mathbf{cell}\ c(a.P) \vdash_\eta \Delta, c : \mathbf{S}_f\ A; \Gamma}\ [\text{Tcell}] \qquad \frac{}{\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma}\ [\text{Tempty}]$$

$$\frac{}{\mathbf{release}\ c \vdash_\eta c : \mathbf{U}_f\ A; \Gamma}\ [\text{Trelease}]$$

$$\frac{Q \vdash_\eta \Delta, a : \vee A, c : \mathbf{U}_e\ A; \Gamma}{\mathbf{take}\ c(a); Q \vdash_\eta \Delta, c : \mathbf{U}_f\ A; \Gamma}\ [\text{Ttake}] \qquad \frac{Q_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma \quad Q_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{put}\ c(a.Q_1); Q_2 \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma}\ [\text{Tput}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ c\ \{P \mid\mid Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_f\ A; \Gamma}\ [\text{Tsh}] \qquad \frac{P \vdash_\eta \Delta; \Gamma \quad Q \vdash_\eta \Delta; \Gamma}{P + Q \vdash_\eta \Delta; \Gamma}\ [\text{Tsum}]$$

$$\frac{P \vdash_\eta \Delta', c : \mathbf{U}_e\ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ c\ \{P \mid\mid Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_e\ A; \Gamma}\ [\text{TshL}] \qquad \frac{P \vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_e\ A; \Gamma}{\mathbf{share}\ c\ \{P \mid\mid Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_e\ A; \Gamma}\ [\text{TshR}]$$

Figure 3.4: Typing rules $P \vdash_\eta \Delta; \Gamma$ for CLASS (extends Fig. 2.4).

3.3 Type System

In this section we conclude the definition of the type system for CLASS, by presenting the typing rules associated with the affine/coaffine and the imperative state/usage modalities, as well as the rules for state sharing and nondeterminism. We will also elaborate on the operational model by presenting and commenting the associated principal cut reductions (Def. 14) and structural congruence rules (Def. 13). The introduced concepts will be illustrated with several examples.

Before introducing the type system for the imperative fragment, we need a couple of definitions. We write $\vec{A}$ to denote a finite (possibly empty) array of types. We write $\vec{x} : \vec{A}$, only if $\mathsf{length}(\vec{x}) = \mathsf{length}(\vec{A})$, to denote the typing assignment $\vec{x}[0] : \vec{A}[0], \ldots, \vec{x}[n-1] : \vec{A}[n-1]$, or $\emptyset$ in case $n = 0$. If $\vec{A}$ is an array of types with length n and M a type modality, then $\mathsf{M}\vec{A}$ is an array with length n and such that, for all $0 \leq i \leq n-1$, $(\mathsf{M}\ \vec{A})[i] = \mathsf{M}(\vec{A}[i])$. If $\vec{x}$ and $\vec{y}$ are arrays of names with the same length we let $\{\vec{x}/\vec{y}\}P$ denote the simultaneous substitution of each component $\vec{x}[i]$ by $\vec{y}[i]$ in process P.

Definition 15 (Type System)**.** *The type system of* CLASS *extends the type system of* μCLL *(Definition 6) with the typing rules listed in Fig. 3.4.*

In the following subsections we go through each of the typing rules in detail by presenting the associated structural congruence rules (Definition 13) and principal cut

conversions of reduction $\longrightarrow$ (Definition 14).

Affine, Discard and Use

In CLASS all the sessions stored by cells are affine. This is an essential restriction since reference cells must be released when no longer needed. And in order to release a reference cell, we must be able to safely discard the session it stores, otherwise leaks or deadlocks may occur, hence affinity.

Process $\mathbf{affine}_{\vec{c},\vec{a}}\ a; P$ defines an affine session on a, which can be either used or discarded, and continues as P. It is typed by rule [Taffine], which introduces the affine modality $\wedge A$

$$\frac{P \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, a : A; \Gamma}{\mathbf{affine}_{\vec{c},\vec{a}}\ a; P \ \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, a : \wedge A; \Gamma}\ [\text{Taffine}]$$

At the type-level, a session $a : A$ is promoted to an affine session $a : \wedge A$. The session being promoted depends only on sessions that can be disposed, i.e. that satisfy some form of weakening, namely

- coaffine sessions $a_1, \ldots, a_n = \vec{a}$, which type with $\vee A_1, \ldots, \vee A_n = \vee\vec{A}$, respectively;
- full cell usages $c_1, \ldots, c_m = \vec{c}$, which type with $\mathbf{U}_f\ c_1, \ldots, \mathbf{U}_f\ c_m = \mathbf{U}_f\ \vec{C}$, respectively;
- unrestricted sessions in Γ.

The dependencies on coaffine sessions $\vec{a}$ and on full cell usages $\vec{c}$ are explicitly annotated in the process construct, but sometimes we opt to omit them and we simply write $\mathbf{affine}\ a; P$.

On the coaffine endpoint $\vee\overline{A}$ of a session we have two options: either to use or discard. Process $\mathbf{use}\ a; Q$ uses a coaffine session a and continues as Q. The use operation is typed by rule [Tuse]

$$\frac{Q \vdash_\eta \Delta, a : A; \Gamma}{\mathbf{use}\ a; Q \vdash_\eta \Delta, a : \vee A; \Gamma}\ [\text{Tuse}]$$

The interaction between an affine and use processes is modelled by reduction rule

$$\mathbf{cut}\ \{\mathbf{affine}_{\vec{a},\vec{c}}\ a; P \ \ |a : \wedge A|\ \ \mathbf{use}\ a; Q\} \ \longrightarrow\ \mathbf{cut}\ \{P \ \ |a : A|\ \ Q\} \quad [\wedge\vee\text{u}]$$

A cut on an affine session $a : \wedge A$ between $\mathbf{affine}_{\vec{a},\vec{c}}\ a; P$ and $\mathbf{use}\ a; Q$ reduces to a cut on $a : A$ between the continuations P and Q.

The other alternative is to discard a coaffine session a, as expressed by process $\mathbf{discard}\ a$, the operation being typed by rule [Tdiscard]

$$\frac{}{\mathbf{discard}\ a \vdash_\eta a : \vee A; \Gamma}\ [\text{Tdiscard}]$$

The interaction between an affine and a discard processes is modelled by the reduction rule

$$\mathbf{cut}\ \{\mathbf{affine}_{\vec{c},\vec{a}}\ a; P \ \ |a : \wedge A|\ \ \mathbf{discard}\ a\} \ \longrightarrow\ \mathbf{par}\ \{\mathbf{release}\ \vec{c} \ ||\ \mathbf{discard}\ \vec{a}\} \quad [\wedge\vee\text{d}]$$

A cut between $\mathbf{affine}_{\vec{a},\vec{c}}\ a; P$ and $\mathbf{discard}\ a$ reduces to a mix-composition in which all the coaffine sessions $\vec{a}$ on which P depends are discarded, as in [6, 28], and all the full reference cell usages $\vec{c}$ are released. In the corner case, where P does not depend neither on coaffine nor on usage sessions, the parallel composition on the left-hand side of [$\wedge\vee$d] simply degenerates to inaction $\mathbf{0}$.

We may therefore conceive affine objects as well-behaved disposable values, that when disposed, safely dispose all the resources they hereditarily refer to. Interestingly, this hereditarily resource-disposing behaviour is akin to cascading deletes in relational databases.

We finish this subsection with two examples: in 9 we illustrate the operational model of affine sessions with some simple affine boolean processes, in 10 we show that in CLASS we can parametrically convert bang to affine sessions.

Example 9 (Affine Booleans)**.** *The following process defines an affine boolean on session z*

$$
\begin{array}{lcl}
P(z) & \vdash & z : \wedge\mathsf{Bool} \\
P(z) & \triangleq & \mathbf{cut}\{\mathbf{affine}\ x; \mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\ \ \mathbf{affine}\ z; \mathbf{use}\ x; \mathbf{use}\ y; \mathsf{and}(x, y, z)\}
\end{array}
$$

It composes the affine booleans true and false with an affine process that computes their logical conjunction.

The computation of the conjunction is postponed until it is necessary, furthermore it might not be carried at all if the affine session z is not used, i.e. if it is discarded. In this case, the affine booleans x and y, on which the computation depends will also be safely discarded. Operationally, this is captured with the following sequence of reductions

$$
\begin{array}{l}
\mathbf{cut}\ \{P(z)\ \ |z|\ \ \mathbf{discard}\ z\} \\
\equiv \mathbf{cut}\{\mathbf{affine}\ x; \mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\ \ \mathbf{affine}\ z; \mathbf{use}\ x; \mathbf{use}\ y; \mathsf{and}(x, y, z)\ |z|\ \mathbf{discard}\ z\} \\
\longrightarrow \mathbf{cut}\{\mathbf{affine}\ x; \mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\ \ \mathbf{par}\ \{\mathbf{discard}\ x\ ||\ \mathbf{discard}\ y\}\} \\
\equiv \mathbf{par}\ \{\mathbf{cut}\ \{\mathbf{affine}\ x; \mathsf{true}(x)\ \ |x|\ \ \mathbf{discard}\ x\}\ ||\ \mathbf{cut}\ \{\mathbf{affine}\ y; \mathsf{false}(y)\ \ |y|\ \ \mathbf{discard}\ y\}\} \\
\longrightarrow \mathbf{par}\ \{\mathbf{0}\ ||\ \mathbf{cut}\ \{\mathbf{affine}\ y; \mathsf{false}(y)\ \ |y|\ \ \mathbf{discard}\ y\}\} \\
\longrightarrow \mathbf{par}\ \{\mathbf{0}\ ||\ \mathbf{0}\} \equiv \mathbf{0}
\end{array}
$$

by successively applying principal cut conversion rule [$\wedge\vee d$].

Alternatively, we can define a process

$$
\begin{array}{lcl}
Q(z, w) & \vdash & z : \vee\overline{\mathsf{Bool}}, w : \mathsf{Bool} \\
Q(z, w) & \triangleq & \mathbf{use}\ z; \mathsf{not}(z, w)
\end{array}
$$

that uses the affine boolean z and computes its logical negation on w.

When $Q(z)$ interacts with $P(z)$ it forces the computation of the logical conjunction to be carried

out, as captured by the following sequence of reductions

$$
\begin{aligned}
&\mathbf{cut}\ \{P(z)\ \ |z|\ \ Q(z,w)\}\\
&\equiv \mathbf{cut}\{\mathbf{affine}\ x; \mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\\
&\qquad \mathbf{affine}\ z; \mathbf{use}\ x; \mathbf{use}\ y; \mathsf{and}(x,y,z)\ |z|\ \mathbf{use}\ z; \mathsf{not}(z,w)\}\\
&\longrightarrow \mathbf{cut}\{\mathbf{affine}\ x; \mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\ \ \mathbf{use}\ x; \mathbf{use}\ y; \mathsf{and}(x,y,z)\ |z|\ \mathsf{not}(z,w)\}\\
&\longrightarrow \mathbf{cut}\{\mathsf{true}(x)\ |x|\ \mathbf{affine}\ y; \mathsf{false}(y)\ |y|\ \ \mathbf{use}\ y; \mathsf{and}(x,y,z)\ |z|\ \mathsf{not}(z,w)\}\\
&\longrightarrow \mathbf{cut}\{\mathsf{true}(x)\ |x|\ \mathsf{false}(y)\ |y|\ \ \mathsf{and}(x,y,z)\ |z|\ \mathsf{not}(z,w)\}\\
&\overset{+}{\longrightarrow}\ \mathbf{cut}\{\mathsf{false}(z)\ |z|\ \mathsf{not}(z,w)\}\\
&\overset{+}{\longrightarrow} \mathsf{true}(w)
\end{aligned}
$$

where, this time, the derivation follows by successively applying reduction rule $[\wedge\vee u]$.

Example 10 (Converting Exponential to Affine Sessions)**.** *Exponential (or unrestricted) sessions $!, ?$ can be used multiple times since they satisfy contraction and can also be disposed since they satisfy weakening. On the other hand, affine/coaffine sessions can be disposed since they satisfy weakening. Therefore, exponential sessions are a particular case of affine/coaffine sessions and in this example we code a process*

$$\mathsf{conv}(f) \vdash f : \forall X.!X \multimap \wedge X$$

that witnesses this inclusion relationship.

Process $\mathsf{conv}(f)$ *converts a bang $!X$ to an affine session $\wedge X$, the definition works for all types X as explicitly indicated by the universal quantification. The definition is as follows*

$$\mathsf{conv}(f) \triangleq \mathbf{recvty}\ f(X); \mathbf{recv}\ f(x); ?x; \mathbf{affine}\ f; \mathbf{call}\ x(x_0); \mathbf{fwd}\ x_0\ f$$

$\mathsf{conv}(f)$ *inputs on f a type X, then a session $x : ?\overline{X}$ and moves it to the unrestricted typing context as $x : \overline{X}$. Then, it defines an affine session $f : \wedge X$ which, when used, call x on $x_0 : \overline{X}$, which is then forwarded to $f : X$. If the affine session is discarded no call to the replicated server ever occurs. Therefore, a bang session is encoded as an affine session which invokes the exponential server at most once.*

Exponential sessions can be used multiple times and copied, wheres affine sessions cannot, so we conjecture that there is no process that satisfies the typing context $f : \forall X.\ \wedge X \multimap !X$, obtained by reversing the arrow.

Reference Cells and Cell Usages

CLASS supports first-class higher-order reference cells. Its fine-grained type system, based on linear logic, ensures that all the cell usages operations are well-behaved at compile time and that every stateful program is deadlock-free, conditions that hold even in the presence of state sharing, as we shall see later.

In CLASS reference cells are expressed by $\mathbf{cell}\ c(a.P)$, this construct defines a cell on session c which stores an affine session a, the behaviour on a is implemented by P. It is

typed by rule [Tcell], which introduces the state full modality $\mathbf{S}_f\ A$

$$\frac{P \vdash_\eta \Delta, a : \wedge A; \Gamma}{\mathbf{cell}\ c(a.P) \vdash_\eta \Delta, c : \mathbf{S}_f\ A; \Gamma}\ [\text{Tcell}]$$

At the type-level it promotes an affine session $a : \wedge A$ to a full state session $a : \mathbf{S}_f\ A$.

On the dual session endpoint, the basic operation is to release the cell usage c, which is done with **release** c. Release is types by [Trelease] and introduces the full usage modality $\mathbf{U}_f\ \overline{A}$

$$\frac{}{\mathbf{release}\ c \vdash_\eta c : \mathbf{U}_f\ \overline{A}; \Gamma}\ [\text{Trelease}]$$

[Trelease] corresponds to DiLL [53] coweakening.

When all the threads are releasing their usages c it is necessary to safely dispose the corresponding reference cell, so as to avoid memory leaks, which is done in CLASS by discarding the affine session the reference cell stores, as modelled by the reduction rule

$$\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{release}\ c\}\ \ \rightarrow\ \ \mathbf{cut}\ \{P\ \ |a : \wedge A|\ \ \mathbf{discard}\ a\}\quad [\mathbf{S}_f\ \mathbf{U}_f\ \text{f}]$$

The other possible operation on a full usage is to take the session stored by the reference cell: process **take** $c(a'); Q$ takes session stored in c on input parameter a' and then continues as Q. Take acquires the implicit lock associated with the reference cell c. It is typed by rule [Ttake]

$$\frac{Q \vdash_\eta \Delta, a' : \vee\overline{A}, c : \mathbf{U}_e\ \overline{A}; \Gamma}{\mathbf{take}\ c(a'); Q \vdash_\eta \Delta, c : \mathbf{U}_f\ \overline{A}; \Gamma}\ [\text{Ttake}]$$

After the take operation the reference cell c becomes empty, notice that in [Take] the cell usage shifts from full (in the conclusion) to empty (in the premise).

Process **empty** c defines an empty cell on c and is typed by rule [Tempty], which introduces the state empty modality $\mathbf{S}_e\ A$

$$\frac{}{\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma}\ [\text{Tempty}]$$

The interaction between a cell and take operation is captured by the reduction rule

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{take}\ c(a'); Q\}\\ &\rightarrow \mathbf{cut}\ \{P\ \ |a : \wedge A|\ \ \mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c : \mathbf{S}_e\ A|\ \ \{a/a'\}Q\}\}\quad [\mathbf{S}_f\ \mathbf{U}_f\ \text{t}]\end{aligned}$$

A cut on session $c : \mathbf{S}_f\ A$ between a full cell **cell** $c(a.P)$ and a take **take** $c(a'); Q$ reduces to a process expression with two cuts, both composed with the continuation $\{a/a'\}Q$ of the take operation. The outer cut on $a : \wedge A$ composes with the stored affine session, implemented by process P, which was successfully acquired by the take operation. The inner cut on $c : \mathbf{S}_e\ A$ composes with the empty cell c. In the continuation of the take, he input parameter a' is substituted by the session a stored in the cell, as denoted by $\{a/a'\}Q$.

The cell can be restored to full again, with a put operation which also releases the implicit lock. Process **put** $c(a'.Q_1); Q_2$ puts a new coaffine session a' in the empty cell c

and continues as Q_2, the behaviour at a' is provided by process Q_1. It is typed by rule [Tput]

$$\frac{Q_1 \vdash_\eta \Delta_1, a' : \wedge A; \Gamma \qquad Q_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ \overline{A}; \Gamma}{\mathbf{put}\ c(a'.Q_1); Q_2 \ \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ \overline{A}; \Gamma}\ [\text{Tput}]$$

The put operation acts as a symmetric operation of the take operation, in fact, if we read rule [Ttake] from the conclusion to the premise, then take shifts from a full $\mathbf{U}_f\ \overline{A}$ to an empty usage $\mathbf{U}_e\ \overline{A}$, whereas [Tput] does the converse.

The interaction between an empty cell and a put operation is modelled by the reduction rule

$$\mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c : \mathbf{S}_e\ A|\ \ \mathbf{put}\ c(a'.Q_1); Q_2\} \ \rightarrow\ \mathbf{cut}\ \{\mathbf{cell}\ c(a'.Q_1)\ \ |c : \mathbf{S}_f\ A|\ \ Q_2\} \quad [\mathbf{S}_e\ \mathbf{U}_e\ ]$$

The put operation restores an empty cell c, which then becomes full, storing session a'. The full cell c is then accessed by the continuation process Q_2.

Whereas a full usage can either be taken or released, an empty usage needs to be restored to full with a put operation, an *obligation* which guarantees that other potential concurrent take operations that are waiting for the cell to become full will eventually succeed, thereby avoiding deadlocks. Therefore, in CLASS, the take-put dynamics follows the usual safe pattern of mutex-protected objects in which each lock-acquire (take) must be followed by a lock-release (put).

Furthermore, in CLASS, the lock and the data it protects are tightly associated (one must acquire the data with a take operation before any update), which contrasts with the mutex-based style of programming, e.g mutex synchronisation in POSIX threads API, in which one must not forget to acquire the lock before updating the data it protects.

Typing rules [Ttake] and [Tput] incorporate codereliction, but also follow the general structure of the typing rules for session input ([T⅋]) and output ([T⊗]).

Before introducing state sharing and nondeterminism, we present three examples. In 11 we illustrate the sequential cell usage operations with a basic mutable reference cell that stores a natural and supports two increment and get operations. In 12 we code mutable pairs and a swap operation. In the subsequent chapter we will code further pointed data structures such as linked lists. We conclude with 13, by coding exponential reference cells, i.e. cells that store unrestricted sessions typed by bang !.

Example 11. *In Fig. 3.5 we define a process* $\mathsf{init}(c)$*, that declares reference cell* c *storing the natural* 42*. We also show code for two basic operations, one to increment and other to get the natural stored in the reference cell.*

Process $\mathsf{inc}(c)$ *takes the affine natural* n *stored in the reference cell* c*. Then, it puts back the successor of* n*, after which it releases its cell usage. Notice that we explicitly indicate that we are using the taken affine natural* n*, the other possibility would be to discard it. The take-put defines a critical section.*

Process $\mathsf{get}(c, x)$ *takes the natural* n *stored in the reference cell* c *and outputs* n *in the session channel* x*, after which it closes* x*. In parallel,* $\mathsf{get}(c, x)$ *resets the cell to* $\mathbf{0}$ *and releases its cell usage.*

$$
\begin{array}{lcl}
\mathsf{init}(c) & \vdash & c : \mathbf{S}_f\ \mathsf{Nat} \\
\mathsf{init}(c) & \triangleq & \mathbf{cell}\ c(n.\mathbf{affine}\ n; \mathsf{V}_{42}(n))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{inc}(c) & \vdash & c : \mathbf{U}_f\ \overline{\mathsf{Nat}} \\
\mathsf{inc}(c) & \triangleq & \mathbf{take}\ c(n); \\
 & & \mathbf{put}\ c(m.\ \mathbf{affine}\ m; \\
 & & \qquad \mathbf{use}\ n; \\
 & & \qquad \mathsf{succ}(n, m)); \\
 & & \mathbf{release}\ c
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{get}(c, x) & \vdash & c : \mathbf{U}_f\ \overline{\mathsf{Nat}}, x : \mathsf{Nat} \otimes \mathbf{1} \\
\mathsf{get}(c, x) & \triangleq & \mathbf{take}\ c(n); \\
 & & \mathbf{par}\{ \\
 & & \quad \mathbf{use}\ n; \\
 & & \quad \mathbf{send}\ x(n); \\
 & & \quad \mathbf{close}\ x \\
 & & \quad || \\
 & & \quad \mathbf{put}\ c(m.\mathbf{affine}\ m; V_0(m)); \\
 & & \quad \mathbf{release}\ c \\
 & & \quad \}
\end{array}
$$

Figure 3.5: A reference cell with increment and get operations.

Finally, we define two simple systems that composes, via cuts on session c, the reference cell and with the operations to increment and get

$$
\begin{array}{lcl}
\mathsf{system}_1 & \vdash & \emptyset \\
\mathsf{system}_1 & \triangleq & \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathsf{inc}(c)\}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{system}_2(x) & \vdash & x : \mathsf{Nat} \otimes \mathbf{1} \\
\mathsf{system}_2(x) & \triangleq & \mathbf{cut}\ \{\mathsf{init}(c)\ \ |c|\ \ \mathsf{get}(c, x)\}
\end{array}
$$

When evaluating system_1*, the cell on c will end up storing the natural* 43*. On the other,* $\mathsf{system}_2(x)$ *reduces to a process in which the natural* 42 *is being transmitted on channel x, this time the cell c ends up storing the natural* 0*. Both threads* $\mathsf{inc}(c)$ *and* $\mathsf{get}(c, x)$ *release their usages c and, in this case, since the cell is not being shared by any other thread, this leads to cell deallocation in both systems. We illustrate the operational model with the reduction sequence for* $\mathsf{system}_2(x)$

$$
\begin{array}{l}
\mathsf{system}_2(x) \\
\longrightarrow \mathbf{par}\ \{\mathbf{cut}\ \{\mathsf{V}_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}\ || \\
\qquad \mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c|\ \ \mathbf{put}\ c(m.\mathbf{affine}\ m; \mathsf{V}_0(m)); \mathbf{release}\ c\}\}\ \textit{(1)} \\
\longrightarrow \{\mathbf{cut}\ \{\mathsf{V}_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}\ ||\ \mathbf{cut}\ \{\mathbf{cell}\ c(m.\mathbf{affine}\ m; \mathsf{V}_0(m))\ \ |c|\ \ \mathbf{release}\ c\}\}\ \textit{(2)} \\
\longrightarrow \{\mathbf{cut}\ \{\mathsf{V}_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}\ ||\ \mathbf{cut}\ \{\mathbf{affine}\ m; \mathsf{V}_0(m)\ \ |m|\ \ \mathbf{discard}\ m\}\}\ \textit{(3)} \\
\longrightarrow \{\mathbf{cut}\ \{\mathsf{V}_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}\ ||\ \mathbf{0}\} \equiv \mathbf{cut}\ \{\mathsf{V}_{42}(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}\ \textit{(4)}
\end{array}
$$

After the take operation the system evolves to a parallel composition of two cuts (1). The cut on the left-hand side of the parallel composition connects the acquired natural, which was already used, with the output process on session x. On the other hand, the cut on the right-hand side connects the empty cell with the process that resets the cell before releasing. Then, the empty cell is set with the natural 0*, just before releasing (2). Releasing safely leads to cell deallocation since no other thread has access to usage c, which is done by discarding the affine session m stored in c (3). Since the affine session m stored in the cell does not depend on other resources, discarding reduces simply to the inaction process (4).*

Example 12 (Mutable Pairs)**.** *The type of mutable pairs, whose first component is of type A and second component is of type B is denoted by* $\mathsf{Pair}(A, B)$ *and defined by*

$$\mathsf{Pair}(A, B) \triangleq \mathbf{S}_f\ [(\wedge A) \otimes \mathbf{S}_f\ B]$$

Process $\mathsf{init}(a, b, c_1) \vdash a : \vee\overline{A}, b : \vee\overline{B}, c_1 : \mathsf{Pair}(A, B)$ *defines a pair on session* c_1 *whose first component is forwarded to* a *and second component is forwarded to* b*:*

$$\mathsf{init}(a, b, c_1) \triangleq \mathbf{cut}\ \{\mathbf{cell}\ c_1(x_1.\mathbf{affine}\ x_1; \mathbf{send}\ x_1(a); \mathbf{fwd}\ x_1\ c_2)\ \ |c_2|\ \ \mathbf{cell}\ c_2(x_2.\mathbf{fwd}\ x_2\ b)\}$$

Process

$$\mathsf{swap}(c_1, c_1') \vdash c_1 : \overline{\mathsf{Pair}(A, A)}, c_1' : \mathsf{Pair}(A, A)$$

swaps the components of pair c_1 *and outputs the updated pair on* c_1'*, being defined by*

$$\begin{aligned}\mathsf{swap}(c_1, c_1') \triangleq\ & \mathbf{take}\ c_1(c_2); \mathbf{use}\ c_2; \mathbf{recv}\ c_2(a); \mathbf{take}\ c_2(b);\\ & \mathbf{put}\ c_2(a'.\mathbf{fwd}\ a\ a'); \mathbf{put}\ c_1(c_2'.\mathbf{affine}\ c_2'; \mathbf{send}\ c_2'(b); \mathbf{fwd}\ c_2'\ c_2); \mathbf{fwd}\ c_1\ c_1'\end{aligned}$$

Cell c_1 *is taken and unpacked with a use and receive operations, after which we obtain the first component* $a : \vee\overline{A}$*, an empty cell usage* $c_1 : \boldsymbol{U}_e\ \overline{A}$ *and a full cell usage* $c_2 : \boldsymbol{U}_f\ \overline{A}$*. We take the second component* $b : \vee\overline{A}$ *from the cell usage* c_2 *and then we start to restore the empty usages* c_1 *and* c_2*: we put on* c_2 *the first component* a *and we put on* c_1 *an affine session* c_2'*, in which we send the the second component* b *and continue as a cell* c_2' *that forwards to* c_1*. Finally, after the update operations, we forward the usage* c_1 *to* c_1'*.*

As expected, one obtains

$$\mathbf{cut}\ \{\mathsf{init}(a, b, c_1)\ \ |c|\ \ \mathsf{swap}(c_1, c_1')\} \xrightarrow{+} \mathsf{init}(b, a, c_1')$$

where $\xrightarrow{+}$ *denotes the transitive-closure of* $\longrightarrow$*.*

In CLASS *sessions are safely disposed, when they are no longer used. For example, when we release a mutable pair, a chain of cell-release and affine-discard interactions is triggered that ends up in a process that discards the resources* a *and* b *in parallel:*

$$\mathbf{cut}\ \{\mathsf{init}(a, b, c_1)\ \ |c_1|\ \ \mathbf{release}\ c_1\} \xrightarrow{+} \mathbf{par}\ \{\mathbf{discard}\ b\ ||\ \mathbf{discard}\ a\}$$

The dynamics is captured by the following reduction sequence

$$\begin{aligned}&\mathbf{cut}\ \{\mathsf{init}(a, b, c_1)\ \ |c_1|\ \ \mathbf{release}\ c_1\}\\ &\equiv \mathbf{cut}\ \{\mathbf{cell}\ c_2(y.\mathbf{affine}\ y; \mathbf{fwd}\ y\ b)\ |c_2|\ \ \mathbf{cell}\ c_1(x.\mathbf{affine}\ x; \mathbf{send}\ x(a); \mathbf{fwd}\ x\ c_2)\ |c_1|\ \mathbf{release}\ c_1\}\\ &\longrightarrow \mathbf{cut}\ \{\mathbf{cell}\ c_2(y.\mathbf{affine}\ y; \mathbf{fwd}\ y\ b)\ |c_2|\ \ \mathbf{affine}\ x; \mathbf{send}\ x(a); \mathbf{fwd}\ x\ c_2\ |x|\ \ \mathbf{discard}\ x\}\\ &\qquad\qquad (\longrightarrow [\mathbf{S}_f\ \mathbf{U}_f\ \mathrm{f}])\\ &\longrightarrow \mathbf{par}\ \{\mathbf{cut}\ \{\mathbf{cell}\ c_2(y.\mathbf{affine}\ y; \mathbf{fwd}\ y\ b)\ \ |c_2|\ \ \mathbf{release}\ c_2\}\ ||\ \mathbf{discard}\ a\} \qquad (\longrightarrow [\wedge\vee\mathrm{d}])\\ &\longrightarrow \mathbf{par}\ \{\mathbf{cut}\ \{\mathbf{affine}\ y; \mathbf{fwd}\ y\ b\ \ |y|\ \ \mathbf{discard}\ y\}\ ||\ \mathbf{discard}\ a\} \qquad (\longrightarrow [\mathbf{S}_f\ \mathbf{U}_f\ \mathrm{f}])\\ &\longrightarrow \mathbf{par}\ \{\mathbf{discard}\ b\ ||\ \mathbf{discard}\ a\} \qquad (\longrightarrow [\wedge\vee\mathrm{d}])\end{aligned}$$

This example as well as some tests are coded in state/mutable-pairs.clls.

Example 13. *Exponential Cells, Read and Write Memory cells in* CLASS *are like Haskell MVars and store affine resources whose values can be taken and updated with a put operation. In this*

example we show that, nevertheless, we can code ML-like memory cells that support a reading operation that necessarily duplicates the value that the cell stores.

This is simply done in CLASS *by storing exponential sessions, these cells are typed by* $\boldsymbol{S}_f\ !A$ *and we call them* exponential cells. *Additionally, this examples hints how the session-typed language* πSSL, *with first-class exponential reference cells and locks, defined in [134], can be embedded in* CLASS.

Since exponential sessions can be copied, this allows us to define a read operation on exponential cells. More specifically, define the process constructs

$$\begin{array}{lcl} \mathsf{cell}_! \ c(a.P) & \triangleq & \mathbf{cell}\ c(a.\mathbf{affine}\ a; P) \\ \mathbf{read}\ c(a_0); P & \triangleq & \mathbf{take}\ c(a); \mathbf{use}\ a; ?a; \mathbf{put}\ c(a_1.\mathbf{affine}\ a_1; \mathbf{fwd}^!\ a_1\ a); \mathbf{cut}\ \{\mathbf{fwd}^!\ a_0\ a \ \ |a_0|\ \ P\} \\ \mathbf{wrt}\ c(a_0.P); Q & \triangleq & \mathbf{take}\ c(a); \mathbf{put}\ c(a_0.\mathbf{affine}\ a_0; P); \mathbf{par}\ \{\mathbf{discard}\ a \parallel Q\} \end{array}$$

The following typing rules are derivable from the typing rules of CLASS

$$\frac{P \vdash_\eta a : !A; \Gamma}{\mathsf{cell}_!\ c(a.P) \vdash_\eta c : \boldsymbol{S}_f\ !A; \Gamma}\ [Tcell!] \qquad \frac{P \vdash_\eta \Delta, c : \boldsymbol{U}_f\ ?A, a_0 : ?A; \Gamma}{\mathbf{read}\ c(a_0); P \vdash_\eta \Delta, c : \boldsymbol{U}_f\ ?A; \Gamma}\ [Tread]$$

$$\frac{P \vdash_\eta a_0 : !\overline{A}; \Gamma \quad Q \vdash \Delta, c : \boldsymbol{U}_f\ ?A; \Gamma}{\mathbf{wrt}\ c(a_0.P); Q\ \vdash_\eta \Delta, c : \boldsymbol{U}_f\ ?A; \Gamma}\ [Twrite]$$

We will now describe the code for each of the operations read and write. $\mathbf{read}\ c(a_0); P$ *reads the exponential reference cell* c *on parameter* a_0 *and continue as* P. *It is encoded in* CLASS *as an affine (full) usage operation* $c : \boldsymbol{U}_f\ ?A$ *that takes the exponential stored session* $a : \vee ?A$ *from the affine cell, uses the stored session* $a : ?A$ *and moves it to the unrestricted typing context, typed as* $a : A$.

Once in the unrestricted typing context, the stored session is then copied, via two exponential forwarders, to produce two new sessions: a_0 *and* a_1. *Session* a_1 *is used to restore the empty cell whereas* a_0 *is composed with the continuation of the read process* P.

Operation $\mathbf{wrt}\ c(a_0.P); Q$ *updates an exponential reference cell* c *with a new session* a_0, *the behaviour on* a_0 *is offered by* P, *and continues as* Q. *It is implemented as an affine usage operation* $c : \boldsymbol{U}_f\ ?A$ *that takes the session* $a : \wedge ?A$ *stored on* c, *puts a new session* $a_0 : \wedge !\overline{A}$ *and continues as* Q. *In parallel it discards the taken session* a.

Alternatively, notice that we can dispose the taken session $x : \vee ?A$ *by first using it* $x : ?A$ *and then moving to the unrestricted typing context, to be typed as* $x : A$, *whereas it would be weakened (as in* $\mathbf{use}\ x; ?x; -$*).*

The read and write operations are used frequently in CLASS *in examples that manipulate reference cells storing exponential sessions, as we will see in Example 4.2.*

This example and some tests are coded in state/exponential-cells.clls. *Furthermore, in* state/exponential-cells-and-locks.clls *we extend the encoding with first-class locks over exponential cells, which then allows us to define indivisible sequences of read/write operations.*

State Sharing

Until now, we have only considered sequential cell usage operations, however one of the main benefits of having mutable state is to be able to share it among cooperative threads.

In CLASS, state sharing is supported by the logically motivated construct **share** $c\ \{P \parallel Q\}$, which allows a cell usage c to be shared by an arbitrary number of concurrent threads.

Sharing is typed by rule [Tsh]

$$\frac{P \vdash_\eta \Delta', c : \mathbf{U}_f\ \overline{A}; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_f\ \overline{A}; \Gamma}{\mathbf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_f\ \overline{A}; \Gamma}\ [\text{Tsh}]$$

Typing rule [Tsh] enforces that processes P and Q may linearly interact at the shared reference c, but not on other linear objects.

Typing rule [Tsh] for the share construct corresponds precisely to cocontraction of the cell usage modality (cf. DiLL [53]). Contraction of the exponentials allows a proposition $?A$ to be copied, similarly cocontraction produces two cell usages $\mathbf{U}_f\ A$ from a single one. However, contraction allows a flow between the two contracted propositions whereas cocontraction ensures that there is now such flow, by splitting the linear region of the conclusion by the two premises being cocontracted.

Cocontraction relates with cut rule [Tcut], where two threads cannot interact on more than one channel and ensures the basic acyclicity of linear logic [23, 157]. This is an essential condition to establish deadlock-freedom of the typed calculus (Theorem 2) by purely logical means. Because suppose that we could share more than one cell usage between two concurrent threads. Then, we could type the blocked program

$$\mathbf{share}\ c_1, c_2\ \{\mathbf{take}\ c_1(a); \mathbf{take}\ c_2(b); P \parallel \mathbf{take}\ c_2(b); \mathbf{take}\ c_1(a); Q\}$$

unless we impose some sort of restriction on top of the logical system such as partial orders [11].

A priori, this restriction of sharing only one reference cell might seem quite restrictive. However, notice that a single shared cell may group all the state shared by the two threads (e.g. in a resource bundle, a sequence of values). In fact, sharing in CLASS allows quite expressive stateful programs to be typed, such as shared mutable ADTs with dynamic sharing topologies, in which clients of the ADT can be connected on the fly. These sharing topologies can grow potentially unbounded, this contrasts with some session-typed languages (e.g., [91]) in which the number of participants sharing a session is statically bounded by the type system. Furthermore, as we shall see later, in CLASS we can even code shared resource hierarchies, without resorting to partial orders, and where deadlock-freedom follows *just* by linearity.

In CLASS, besides cocontracting two full usages we can also cocontract two usages where one of them is full and the other is empty, as typed [TshL] and [TshR]

$$\frac{P \vdash_\eta \Delta', c : \mathbf{U}_e\ \overline{A}; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_f\ \overline{A}; \Gamma P \vdash_\eta \Delta', c : \mathbf{U}_f\ \overline{A}; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_e\ \overline{A}; \Gamma}{\mathbf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_e\ \overline{A}; \Gamma \quad \mathbf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_e\ \overline{A}; \Gamma}$$

The cell usage is propagated as full if both premises type with a full usage (rule [Tsh]); otherwise one and only one of the premises type with an empty usage, in which case we

propagate an empty usage (rules [TshL] and [TshR]). Therefore, typing rules [Tsh], [TshL], [TshR] guarantee that in any sharing topology at most one thread types with an empty usage, that is, in any sharing topology at most one thread has taken the cell.

Notice that the share construct $\mathbf{share}\ c\ \{- \parallel -\}$ is *not* a binding operator, in fact c is present in the typing context of the conclusion in rules [Tsh], [TshL] and [TshR] and actions on c will be propagated by applying one of the structural congruence $\equiv$ rules [TSh], [RSh] or [PSh] (Def. 13), which we will now introduce.

Structural congruence law [TSh] is written as

$$\begin{aligned}&\mathbf{share}\ c\ \{\mathbf{take}\ c(a_1); P_1 \parallel \mathbf{take}\ c(a_2); P_2\}\\ &\quad\equiv \mathbf{take}\ c(a_1); \mathbf{share}\ c\ \{P_1 \parallel \mathbf{take}\ c(a_2); P_2\} + \mathbf{take}\ c(a_2); \mathbf{share}\ c\ \{\mathbf{take}\ c(a_1); P_1 \parallel P_2\}\end{aligned}$$

On the left-hand side of $\equiv$ law [TSh], two threads are racing to take the session stored in the reference cell c. The two possible interleavings of the concurrent take actions are expressed as a nondeterministic sum on the right-hand side, where, in each summand, the successful take operation commutes with the share construct whereas the other take is postponed. The share construct on the left-hand side of the congruence rule is typed by [TSh], whereas the left and right summands on the right-hand side are typed by [TShL] and [TShR], respectively, thereby indicating which thread has taken the session stored in the cell.

The share construct satisfies the following structural congruence $\equiv$ law

$$\mathbf{share}\ c\ \{\mathbf{release}\ c \parallel P\} \quad\equiv\quad P \qquad \text{[RSh]}$$

Algebraically, rule [RSh] expresses that the release operation acts as an identity for share-composition. Computationally, it allows a release operation to be discarded (when we read the congruence from left to right), provided it occurs within a share block.

When all processes in a sharing topology are releasing a common cell usage we can successively apply [RSh] in order to obtain a single release operation that no longer occurs within a share block and, therefore it can safely interact with the reference cell leading to cell deallocation, as in the following derivation

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c|\ \ \mathbf{share}\ c\ \{\mathbf{share}\ c\ \{\mathbf{release}\ c \parallel \mathbf{release}\ c\} \parallel \mathbf{release}\ c\}\}\\ &\equiv \mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c|\ \ \mathbf{share}\ c\ \{\mathbf{release}\ c \parallel \mathbf{release}\ c\}\}\\ &\equiv \mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c|\ \ \mathbf{release}\ c\} \longrightarrow \mathbf{cut}\ \{P\ \ |a|\ \ \mathbf{discard}\ a\}\end{aligned}$$

Rule [RSh] applies to share constructs that can either be typed by [TSh] or [TShR].

Finally, the share construct satisfies $\equiv$ law

$$\mathbf{share}\ c\ \{\mathbf{put}\ c(a.P); Q \parallel R\} \quad\equiv\quad \mathbf{put}\ c(a.P); \mathbf{share}\ c\ \{Q \parallel R\} \qquad \text{[PSh]}$$

which allows us to commute a put with a share construct, thereby *bubbling up* a put action in a share topology which eventually reaches a reference cell. If a process P has taken the cell contents in a sharing topology, then all the other take operations are blocked until

P puts back a new session, after which the previously blocked take operations can now compete to access the cell according to rule [TSh]. The left-hand side of rule [PSh] types with [TShL], whereas the share on right-hand side types with [TSh].

We conclude this subsection with an examples that illustrates the dynamic nature of sharing in CLASS.

Example 14 (Dynamic Sharing)**.** *The topology of the sharing trees arising from nested share blocks changes dynamically during computation, because of the extrusion of cell references along session channels to outside the share block. Consider the following reduction*

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{share}\ c\ \{R \parallel \mathbf{send}\ x(c); P\}\ \ |x|\ \ \mathbf{recv}\ x(c'); \mathbf{share}\ c'\ \{Q \parallel S\}\}\\ &\xrightarrow{+} \mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{share}\ c\ \{R \parallel \mathbf{share}\ c\ \{\{c/c'\}Q \parallel \{c/c'\}S\}\}\}\end{aligned}$$

Here, a shared alias of c is sent along x to a partner receive process, that inputs the alias and further shares it between threads Q and S. Hence, access to c, initially only shared between the two threads R and **send** $x(c); P$*, ends up being shared among the three threads R, $\{c/c'\}Q$ and $\{c/c'\}S$.*

The thread **send** $x(c); P$ *transfers ownership of c on output – references aliases are linear values, whose visibility may only be duplicated by the share construct. Indeed, linear typing of session send ensures that P must lose access to c.*

The fact that sharing in CLASS *is dynamic allows us to represent interesting programming patterns in which users of a shared resource are coming and leaving on the fly as the system evolves, like for example in the imperative queue Example 4.2, where we can have a dynamic configuration of producers (processes that enqueue) and consumers (processes that dequeue).*

We show typings for components of the above reduction

(1) $R \vdash \Delta_1, c : \boldsymbol{U}_f\ \overline{A}; \Gamma$ (2) $P \vdash \Delta_2, x : B; \Gamma$
(3) $Q \vdash \Delta_3, c' : \boldsymbol{U}_f\ \overline{A}, x : \overline{B}; \Gamma$ (4) $S \vdash \Delta_4, c' : \boldsymbol{U}_f\ \overline{A}; \Gamma$
(5) $\mathbf{send}\ x(c); P \vdash \Delta_2, c : \boldsymbol{U}_f\ \overline{A}, x : \boldsymbol{S}_f\ A \otimes B; \Gamma$ *(Proposition 1([T⊗$_f$]), (2))*
(6) $\mathbf{share}\ c\ \{R \parallel \mathbf{send}\ x(c); P\} \vdash \Delta_1, \Delta_2, c : \boldsymbol{U}_f\ \overline{A}, x : \boldsymbol{S}_f\ A \otimes B; \Gamma$ *([Tsh], (1), (5))*
(7) $\mathbf{share}\ c'\ \{Q \parallel S\} \vdash \Delta_3, \Delta_4, c' : \boldsymbol{U}_f\ \overline{A}, x : \overline{B}; \Gamma$ *([Tsh], (3), (4))*
(8) $\mathbf{recv}\ x(c'); \mathbf{share}\ c'\ \{Q \parallel S\} \vdash \Delta_3, \Delta_4, x : \boldsymbol{S}_f\ A \multimap \overline{B}; \Gamma$ *([T⅋], (7))*
(9) $\mathbf{cut}\ \{\mathbf{share}\ c\ \{R \parallel \mathbf{send}\ x(c); P\}\ \ |x|\ \ \mathbf{recv}\ x(c'); \mathbf{share}\ c'\ \{Q \parallel S\}\} \vdash \Delta_1, \Delta_2, \Delta_3, \Delta_4, c : \boldsymbol{U}_f\ \overline{A}; \Gamma$
([Tcut], (6), (8))

as well as its derivation

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{share}\ c\ \{R \parallel \mathbf{send}\ x(c); P\}\ \ |x|\ \ \mathbf{recv}\ x(c'); \mathbf{share}\ c'\ \{Q \parallel S\}\}\\ &\equiv\ \mathbf{share}\ c\ \{R \parallel \mathbf{cut}\ \{\mathbf{send}\ x(c); P\ \ |x|\ \ \mathbf{recv}\ x(c'); \mathbf{share}\ c'\ \{Q \parallel S\}\}\} && (\equiv \text{rule [CSh]})\\ &\xrightarrow{+} \mathbf{share}\ c\ \{R \parallel \mathbf{cut}\ \{P\ \ |x|\ \ \{c/c'\}\mathbf{share}\ c'\ \{Q \parallel S\}\}\} && (\text{Proposition 1([}\otimes\parr_f\text{])})\\ &= \mathbf{share}\ c\ \{R \parallel \mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{share}\ c\ \{\{c/c'\}Q \parallel \{c/c'\}S\}\}\} && (\text{name substitution})\\ &\equiv \mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{share}\ c\ \{R \parallel \mathbf{share}\ c\ \{\{c/c'\}Q \parallel \{c/c'\}S\}\}\} && (\equiv \text{rule [CSh]})\end{aligned}$$

Nondeterminism

In CLASS nondeterminism naturally emerges through racy concurrent manipulations, which we internalise with sums. This internalisation is crucial if we want to keep within a propositions-as-types approach and be able to reason algebraically about imperative program behaviour.

Nondeterministic sums in CLASS are typed by the following rule

$$\frac{P \vdash_\eta \Delta;\Gamma \quad Q \vdash_\eta \Delta;\Gamma}{P + Q \vdash_\eta \Delta;\Gamma}\ [\text{Tsum}]$$

A sum process represents a nondeterministic choice between two alternatives P and Q, each offering the same typing context $\Delta;\Gamma$. Sums are also present in DiLL [53], where cut elimination needs to generate sums of proofs.

In our model, sum satisfy the expected axioms of nondeterministic sums of process algebras [67], like commutativity ($P+Q \equiv Q+P$), associativity ($P+(Q+R) \equiv (P+Q)+R$) and idempotency ($P + P \equiv P$) and the already presented interleaving law $\equiv$ [TSh] that connects shared state with nondeterminism at the logical level.

Notice that when defining structural congruence $\equiv$ for CLASS (Def. 13) we only list an idempotency law for the inaction process $\mathbf{0}$

$$\mathbf{0} + \mathbf{0} \quad \equiv \quad \mathbf{0} \quad [\mathbf{0}\text{Sm}]$$

Nevertheless, since $\mathbf{0}$ is an unit w.r.t. to parallel composition ($\equiv$ law [$\mathbf{0}$M]) and since mix distributes over sum ($\equiv$ law [MSm]), we can derive idempotency of sum for all processes P in general as shown by

$$P \equiv \mathbf{par}\ \{P \parallel \mathbf{0}\} \equiv \mathbf{par}\ \{P \parallel (\mathbf{0} + \mathbf{0})\} \equiv (\mathbf{par}\ \{P \parallel \mathbf{0}\}) + (\mathbf{par}\ \{P \parallel \mathbf{0}\}) \equiv P + P$$

We conclude this chapter by showing how sums allows us to reason about the behaviour of concurrent programs that manipulate shared state in CLASS by doing simple algebraic-like manipulations. In the next chapter we will see further examples that showcase the expressiveness of CLASS language and type system.

Example 15 (Concurrency, Nondeterminism and Algebraic Reasoning)**.** *The purpose of this example is to illustrate how* CLASS *allows us to reason algebraically about concurrent processes that manipulate shared state.*

We revisit Example 11 in which we defined a reference cell storing a natural number and two operations

$$\mathsf{inc}(c) \vdash c :\vdash c : \boldsymbol{U}_f\ \overline{\mathsf{Nat}} \quad \mathsf{get}(c,x) \vdash c : \boldsymbol{U}_f\ \overline{\mathsf{Nat}}, x : \mathsf{Nat} \otimes \mathbf{1}$$

that increment and get the contents of the cell, respectively. Process $\mathsf{get}(c,x)$ *outputs the natural stored in the cell on session* x*, after which the cell is reseted to* 0*. Suppose that we also have an operation*

$$\mathsf{double}(c) \vdash c :\vdash c : \boldsymbol{U}_f\ \overline{\mathsf{Nat}}$$

that takes the natural stored in the cell and puts back its double, after which the usage is released.

We can then define a system

$$\mathsf{system}(x) \vdash x : \mathsf{Nat} \otimes \mathbf{1}$$

that composes a reference cell c, initially storing 3, with the three concurrent threads

$$\begin{array}{l}\mathsf{system}(x) \triangleq \\ \mathbf{cut}\ \{ \\ \quad \mathbf{cell}\ c(n.\mathbf{affine}\ n; V_3(n)) \\ \quad |c| \\ \quad \mathbf{share}\ c\ \{\mathsf{inc}(c) \mid\mid \mathsf{double}(c) \mid\mid \mathsf{get}(c, x)\} \\ \}\end{array}$$

$\mathsf{system}(x)$ *will output a natural on session x, the precise value will depend on the interleaving of the atomic operations performed by the three concurrent threads, and it is described by the following map that associates each possible way of sequencing the three thread operations increment (I), double (D) and get (G) with the outputted natural on x*

$$\begin{array}{ll} IDG \mapsto 8 & DGI \mapsto 6 \\ IGD \mapsto 4 & GID \mapsto 3 \\ DIG \mapsto 7 & GDI \mapsto 3 \end{array}$$

*For example, $IDG \mapsto 8$, since the natural sent on session x will be result of first incrementing 3 (*inc*), then doubling (*double*) and finally getting and natural and communicating it on session x(*get*):* $(3 + 1) * 2 = 8$.

In CLASS *we can capture all these possible outcomes using nondeterministic sums. By applying a set of simple congruence laws (Def. 13) and reduction rules (Def. 14) we can reason algebraically about process* $\mathsf{system}(x)$ *in order to compute a simplified form. But first we introduce the following definition*

$$\mathsf{out}_k(x) \triangleq \mathbf{cut}\ \{V_k(n)\ \ |n|\ \ \mathbf{send}\ x(n); \mathbf{close}\ x\}$$

Process $\mathsf{out}_k(x) \vdash x : \mathsf{Nat} \otimes \mathbf{1}$ *is parametric on a natural k: it sends k on session x and then closes x.*

Then, in CLASS *we can capture the nondeterministic evolution of* $\mathsf{system}(x)$ *by*

$$\mathsf{system}(x) \xrightarrow{+} \mathsf{out}_3(x) + \mathsf{out}_4(x) + \mathsf{out}_6(x) + \mathsf{out}_7(x) + \mathsf{out}_8(x)$$

After some internal reductions, we can express $\mathsf{system}(x)$ *as sum of processes, each one of the summands is just communicating a natural over channel x. Arguably, it's easier to reason about the behaviour of these summands than about the behaviour of the equivalent original* $\mathsf{system}(x)$.

Sums play a key role in the metatheoretical model of CLASS *as they allows us to explicitly capture all the possible nondeterministic evolutions of a stateful system and, because of that, to have a confluent notion of reduction as will see in Chapter 7. Even though sums are not explicitly*

present in our practical implementation (see Chapter 5) - in a race between two concurrent take operations the interpreter arbitrarily picks one of them and postpones the other - they can still be used to reason about the outcome of a program and possibly to do program simplifications by replacing a program by a simpler equivalent one.

Notice that the original system(x) *is a process that internally manipulates shared state but that can be expressed, after some internal reductions, as a sum of* pure *functional processes that do not use imperative constructs at all. This is not a coincidence, but a particular instance of a general result that will pop up when studying cut normalisation (Section 8.2). More specifically, we will see how to mechanically compute an* equivalent *sum of pure functional processes from any process that manipulates shared state internally but that types with a pure typing context (without state/usage modalities).*

3.4 Further Discussion and Related Work

Resource-Aware Systems for Shared State

Many resource-aware logics and type systems to tame shared state and interference have been proposed [4, 104, 155, 80, 21, 112, 116, 31]. These systems use forms of linearity or affinity, affine types apply to general resourceful programming [150, 38] and to model failures/exceptions [33, 110, 28, 57, 94]. The monadic session-discarding behaviour of affine sessions in CLASS is present in many works, e.g. [6]. The work [3] provides a very general semantic model for the notion of "uniqueness" pervasive in many substructural systems.

Propositions-as-Types and Shared State

CLASS builds on top of the propositions-as-types correspondence with Linear Logic [23, 24, 157], the logical principles for the state modalities being inspired by DiLL [53]. Recent works [10, 11, 7, 91, 128, 134] have also addressed the problem of sharing and nondeterminism in the setting of logical interpretations of sessions.

The work in [134] develops a concurrent session-typed language πSSL with first-class reference cells and locks, based on a PaT correspondence with Classical Linear Logic, the logical principles governing the state modalities being inspired by DiLL [53], and shown to enjoy confluence, cut normalisation and deadlock-freedom, even in the presence of locking primitives. In [134], reference cells may only store *exponential* persistent sessions of type $!A$ and, therefore, cannot refer to other reference cells (reference cells can only be used locally when allocated by persistent sessions). CLASS is a novel, more fundamental approach, able to flexibly handle affine state. While a linear object must be used exactly once (e.g. it cannot be duplicated nor discarded, cf., the absence of contraction and weakening in Linear Logic), an affine object cannot be duplicated, but can be discarded (cf. the weakening principle that holds in the affine type discipline). Therefore, affine state accommodates exponential state as a special case (see Example 13), but is much more expressive, allowing

CLASS to support programs that manipulate shared linear objects, in particular, concurrent shareable linked data structures (see Example 4.1), while still adhering to the PaT approach. Furthermore, recursion in [134] was based on System F encodings, therefore essentially functional, which has motivated in this work the introduction of inductive/coinductive types in order to model stateful recursive structures.

The works [10, 11] introduced manifest sharing, the first proposal to represent shared state on top of a session types Linear Logic interpretation. Although the resulting system is grounded on the Curry-Howard correspondences of [23, 24, 157] it departs from a *pure* PaT interpretation in its stateful extension, unlike the work presented in this paper, and explores a different route, based on a special purpose operational semantics, designed to keep track of shared channels availability to control resource acquisition and perform context switching.

The key idea of manifest sharing is to serialise concurrent access to linear objects by two modal operators (acquire/release), which induce a stratification [124] of session types in two layers, and provide locking / unlocking behaviour at computation points where a resource invariant holds. We follow a related approach in CLASS, where take/put operations play the role of acquire/release in [10, 11], which also allows us to express general corecursive protocols that alternate between a linear and a shared phase, as we will show with Example 4.4. Moreover, in our approach state sharing also essentially differs from [10, 11], since we rely on a logical interpretation of sharing by the DiLL cocontraction rule, where the single DiLL introduction form for codereliction is decomposed in the two take and put primitives, which are subject to the corecursive alternating protocol, coordinated by the sharing type rules [Tsh], [TshL], [TshR]. This interpretation is fundamental to ensure deadlock absence.

Reduction for the stateful fragment in [10, 11] cannot solely be seen as proof simplification, as the operational semantics relies on proof construction / deconstruction steps, where the “wait" computation states are seen as an “incomplete proof", possibly introducing deadlock. Moreover, in manifest sharing, computation is not confluent, and thus cannot be seen as a proof simplification, also for this reason. This is unlike our system in which confluence holds due to the introduction of sums and cocontraction-based sharing.

In [11], shared resources can only depend on shared resources, likewise, in CLASS, reference cell modalities can only depend on affine and reference cell modalities. In [11], the shared modalities satisfy the logical principles of weakening and contraction, however these principles are not process-interpreted, as in CLASS, where the logical principles of cocontraction and coweakening are explicitly annotated with the cell usage sharing and release process constructs, respectively. In [11] it is conjectured that manifest sharing admits a reference counting garbage collector by transforming the typing derivation to make implicit applications of weakening and contraction explicit, but leave this as a conjecture.

The work [128] introduces the session-typed calculus CSLL, by extending a hypersequent formulation of Classical Linear Logic with a pair of type-duality related coexponentials ¡A and ¿A that type stateful servers and their clients, respectively. CSLL satisfies type preservation and progress and it is capable of encoding various examples such as compare-and-set and a functional queue. Language CSLL also draws inspiration on DiLL but, as opposed to DiLL and CLASS, it does not include a general cocontraction rule and it does not internalise nondeterminism with sums and hence lacks confluence.

In CSLL, stateful servers of type $c : ¡A$ are comprised of three components: (i) a state of type $i : B$ (which must be initialised), (ii) an update rule of type $upd : !(B \multimap B \otimes A)$, which, from a previous state $s : B$, produces a new state $s' : B$ and an observable $a : A$ to the client that requested the update and (iii) a finaliser of type $f : \overline{B}$ that consumes the current state B, when there are no more incoming client requests to process.

Client requests $\mathsf{req}\ c(a); Q$ on a server $c : ¿A$ are typed by rule QueA. This rule restricts the threads $Q_1(a)$, that accesses the observable a of the request, and $Q_2(c)$, that accesses the client c for possible more requests, to be parallel i.e.: $Q \equiv \mathbf{par}\ \{Q_1(a) \mid\mid Q_2(c)\}$, due to the hypersequent structure of the premise. By iterated application of rule QueA we can form a sequence of client requests, this sequence is then ordered non-deterministically before being processed by the server, by applying a structural congruence rule (Que − Que) that commutes two arbitrary client requests $\mathsf{req}\ c_1(a_1); \mathsf{req}\ \ c_2(a_2); Q \equiv \mathsf{req}\ c_2(a_2); \mathsf{req}\ c_1(a_1); Q$.

We may model CSLL stateful servers in CLASS by using a reference cell to store the server state B and the update rule

$$\mathsf{SS}(A, B) \triangleq \mathbf{S}_f\ [\wedge B \otimes !(\wedge B \multimap \wedge B \otimes A)]$$

and by expressing client requests as follows

$$\begin{aligned}
\mathsf{req}\ c(a); (\mathbf{par}\ \{Q_1(a) \mid\mid Q_2(c)\}) = \quad & \mathbf{share}\ c\ \{ \\
& \mathbf{take}\ c(x); \mathbf{use}\ x; \mathbf{recv}\ x(s); ?x; \\
& \mathbf{call}\ x(upd); \mathbf{send}\ upd(s); \mathbf{recv}\ upd(s'); \\
& \quad \mathbf{put}\ c(x'.\mathbf{affine}\ x'; \mathbf{send}\ x'(s'); \mathbf{fwd}^!\ x'\ x); \\
& \quad \mathbf{par}\ \{Q_1(upd) \mid\mid \mathbf{release}\ c\} \\
& \quad \mid\mid \\
& \quad Q_2(c)\}
\end{aligned}$$

Given the encoding displayed above, the following typing rule is admissible in CLASS

$$\frac{Q_1(a) \vdash \Delta_1, a : A; \Gamma \quad Q_2(c) \vdash \Delta_2, c : \overline{\mathsf{SS}(A, B)}; \Gamma}{\mathsf{req}\ c(a); (\mathbf{par}\ \{Q_1(a) \mid\mid Q_2(c)\}) \vdash \Delta_1, \Delta_2, c : \overline{\mathsf{SS}(A, B)}; \Gamma}$$

The implementation of $\mathsf{req}\ c(a); (\mathbf{par}\ \{Q_1(a) \mid\mid Q_2(c)\})$ shares the usage c by two concurrent threads, one of which is simply $Q_2(c)$. The other concurrent thread takes the state s stored in the cell c and applies the update rule upd to compute the new state s' and the observable. It then updates the cell c by putting back the new state s', after which it releases the usage

c and in parallel continues as $Q_1(upd)$. Notice that $Q_1(upd)$ will no longer be able to access the stateful server c.

In [128], the authors hint at a connection between DiLL's cocontraction and rule QueA, but leave this "exact relationship" as a future research direction. We believe that this implementation of CSLL stateful servers in CLASS might be elucidative. In particular, notice that the usage for stateful servers in CSLL is propagated linearly to the continuation of the request and cannot be cocontracted "freely" as in CLASS, as the implementation shows: the usage is concontracted only to perform the update, after which the thread that receives the observable releases its usage.

The client modality ¡A satisfies weakening, as typed by QueW, which corresponds to cell usage release in CLASS. Interestingly, in CSLL, the problem of safely discarding a resource is solved in a very distinct way, where each stateful server has an explicitly manually defined finaliser, whereas in CLASS this is done automatically through the interplay of a couple of process manipulations. It would be interesting to investigate how to model manually defined finalisers in CLASS.

Both works [10, 11, 128] tackle a theme common to what is explored in this thesis, with different contributions, directions, and merits.

Recently, [7, 91] also studied nondeterminism in the setting of logical interpretations of session types. The work [7] studies extensions of Wadler's CP calculus obtained by conflating dual types of Linear Logic. By conflating the multiplicatives $\otimes$ and $\parr$ they allow two processes to interact in more than one session, which logically corresponds to the multi-cut rule and introduces the possibility of deadlock programs. By conflating the additives $\&$ and $\otimes$ they obtain a simple form of local coin-flipping nondeterminism which differs from our global nondeterminism, the latter being inherently related with racy manipulation of shared state by concurrent programs. Finally, by conflating the exponentials ! and ?, they obtain access points which allows to non-deterministically match processes that offer a behaviour of type A with processes that offer a dual behaviour $\overline{A}$. They show how reference cells can be implemented through access points and how their system allows non-terminating programs such as the Landin's knot to be typed.

The work [91] extends a hypersequent formulation of Wadler's CP with a form of session sharing that leads to races and nondeterminism, while preserving termination and deadlock-freedom. Their type system draws on bounded Linear Logic, by introducing two integer-subscripted modalities that keep track of the number of statically fixed client-server interactions. This contrasts with our system, in which reference cells can be shared by an arbitrary number of clients that evolves dynamically, as illustrated by Example 14.

Differential Linear Logic (DiLL)

DiLL [53] works on top of linear logic by extending the exponential bang ! modality with new rules, namely coweakening, codereliction and cocontraction, which can be thought of logical principles dual of weakening, contraction and dereliction, these latter apply to

the ? modality.

From the proof-theoretic side, we have built on ideas from DiLL to obtain a logical perspective on state sharing and nondeterminism. We have seen the presence of coweakening and codereliction and cocontraction, this latter being computationally interpreted by mutable state sharing in CLASS.

In DiLL, the meaning of a proof (or process) brought explicit by cut normalisation is represented by a sum

$$P + \cdots + Q$$

understood as the set of alternative outcomes of a computation, as necessary, e.g., to capture the result of a cut-reduction between cocontraction and contraction. In our interpretation, sums play a key role to internalise the nondeterminism that naturally emerges from concurrent state manipulations. Sums allow confluence of cut-reduction to be preserved, allowing non-deterministic proof-reductions to be understood equationally, cf. behavioural equivalence in process algebras [67] or program equivalence in power-domain denotational semantics [125].

Concurrent Haskell MVars and Rust Mutexes

The reference cells of CLASS and their corresponding cell usage operations - put and take - bear resemblance to Concurrent Haskell MVars [83, 102], which have associated the following basic operations

$$\begin{array}{lcl} \mathsf{newEmptyMVar} & :: & \mathsf{IO}\ (\mathsf{MVar}\ A) \\ \mathsf{newMVar} & :: & A \to \mathsf{IO}\ (\mathsf{MVar}\ A) \\ \mathsf{takeMVar} & :: & \mathsf{MVar}\ A \to \mathsf{IO}\ A \\ \mathsf{putMVar} & :: & \mathsf{MVar}\ A \to A \to \mathsf{IO}\ () \end{array}$$

MVar A is the type of mutable locations, which can either be empty (created through newEmptyMVar) or full (created through newMVar), in the latter case they contain a value of type A. The operation takeMVar takes a value from a full MVar and blocks if the MVar is empty. Conversely, the operation putMVar fills in a value into an empty MVar. Although simple, MVars are Haskell building blocks for more complex concurrent data structures.

However, the fine-grained type system of CLASS distinguishes between full and empty state at the type level, whereas in Haskell both full and empty MVars are aggregated under a single type. As a consequence, programs that use MVars in Haskell can cause runtime errors, aborting the whole computations, if a put is ever attempted in a full MVar, whereas in CLASS these *wrong* cell usage scenarios are excluded at compile time. Additionally, programs in Haskell can block, for example if a take is waiting forever for an empty cell to become full. This contrasts with CLASS, in which the take-put dynamics follows the usual safe pattern of mutex-protected objects where each lock-acquire (take) must be followed by a lock-release (put). Similar to what happens in Concurrent Haskell MVars, in CLASS

the lock and the data it protects are tightly associated since one must acquire the data with a take operation before any update.

Also in Rust [86] mutexes are tightly associated with the data they protect and, like in CLASS, the type system distinguishes between locked and unlocked state. However, programs in Rust can still block due to aliasing or because threads can share more that one mutex, whereas, in CLASS, programs are guaranteed to be deadlock-free by the linear type system (cocontraction).

Memory Management in Rust

In our system, a simple form of ownership and ownership transfer results naturally from the underlying linear typing discipline. It is interesting to compare the memory management model of CLASS with the safe manifest presentation of the Rust approach, to which it turns out to be quite similar.

In Rust [86], references may be shared, but mutation is considered an unsafe operation, tamed with the so called “interior mutability" pattern. In CLASS references may be shared using the explicitly share construct, whose typing principle (cocontraction) ensures safety. Constructing cyclic data structures in Rust is unsafe, as it may lead to memory leaks due to the reference counting technique used for recycling storage. In CLASS, well-typed programs do not leak memory: we may understand the release construct as an explicit declaration of ownership dropping (that may be used to decrement a reference count, which is what we do in our implementation to dispose cells). CLASS does not allow cyclic data structures, ensuring strong normalisation and safe sharing, which are fundamental baseline properties. As Rust assumes immutability by default and sharing as a potentially unsafe feature, a pragmatic implementation of CLASS could conceivably adopt a similar perspective for acyclicity, but of course this requires further research.

4

Programming in CLASS

In this chapter we continue to exhibit the expressive power of CLASS language and type system, by coding several examples. The plan is the following

Example 4.1: We start with fundamental concurrent imperative data structures by coding linked lists in CLASS that support memory-efficient updates in-place. This also shows how corecursion and shared mutable state fit very well together in CLASS.

Example 4.2: We implement a concurrent imperative queue ADT, implemented in CLASS using a linked list and two pointers: one to the head (for enqueueing) and other to tail (for dequeueing), which allows enqueueing and dequeueing to be in general performed concurrently in $O(1)$ time by an arbitrary number of processes. This examples shows how standard existential type quantifiers [35, 109] harmoniously combine with the basic stateful framework of CLASS in order to define stateful mutable ADTs.

Example 4.3: We show that in CLASS we can solve famous resource synchronisation problems like the dining philosophers. We implement Dijkstra's solution [51] by representing the order in the forks as a simple passive shared chain. Our solution is quite flexible: new philosophers can be added to the system on the fly. And, interestingly, deadlock-freedom is ensured simply by the linear logic based type system, without resorting to extra-logical devices such as partial orders.

Example 4.4: We show that the semantics of CLASS reference cells and the basic share operation are sufficient to express general corecursive protocols, which satisfy some resource invariant [68, 21, 10], by coding a basic shared toggle.

Example 4.5: In CLASS we can express resource synchronisation methods such as barriers, fork-joins and Hoare-style monitors in which the absence of deadlocks and livelocks is statically guaranteed by the linear type system. In this example, we provide the implementation of a simple barrier.

All these examples were validated by our implemented type checker and tested with our interpreter implementation (Chapter 5), the reader can check code and tests in the

folder *examples/*. In *examples/pure/* we code several programs involving only the pure fragment of CLASS, ranging from basic data structures such as booleans, naturals and trees to corecursive bit counters. Furthermore, for some of the structures, we offer two encodings: one following system-F style, based on polymorphism, and other based on our primitive inductive/coinductive session types.

On the other hand, in *examples/state*, we code programs involving shared mutable state, besides the ones in the chapter, the reader may find further linked data structures such as binary search trees, synchronisation methods such as fork-joins, barriers and monitors and further shared mutable concurrent ADTs such as counters, functional queues with $O(1)$ amortised time for enqueue and dequeue [118], bank accounts and imperative queues.

All these programs satisfy memory safety, confluence, deadlock-freedom and termination, even though they manipulate higher-order state with blocking primitives and recursive imperative data structures. These properties follow automatically and compositionally by our type discipline based on propositions-as-types and linear logic, which imposes crisp yet expressive acyclicity conditions on channel communication topology and shared linked data structures. In the second part of this thesis, we will go through the derivation of all of those metatheoretical properties in detail.

4.1 Linked Lists, Update In-Place

Previously, we have showed how the presence of inductive/coinductive session types allows us to represent in CLASS basic recursive data structures such as naturals (Example 7). In this example we go a step further and show how the presence of reference cells allows us to express more interesting linked data structures with memory-efficient updates in-place.

More specifically, we implement linked lists $\mathsf{LL}(A)$ which store affine sessions A (recall that in CLASS every value is represented as a session). The type $\mathsf{LL}(A)$ is expressed by the mutually recursive pair of definitions

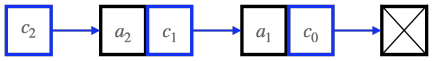
$$\mathsf{LL}(A) \triangleq \mathbf{S}_f\ \mathsf{Node}(A) \quad \mathsf{Node}(A) \triangleq \oplus\{\#\mathsf{Null} : \mathbf{1}, \#\mathsf{Next} : \wedge A \otimes \mathsf{LL}(A)\}$$

A process offering the recursive type $\mathsf{LL}(A)$ behaves as a cell which stores a node of session $\mathsf{Node}(A)$. A session of type $\mathsf{Node}(A)$ either chooses #Null if the list is empty, in which case it closes; or chooses #Next, in which case it sends an affine session $\wedge A$ representing the head element and recurs as the tail $\mathsf{LL}(A)$. We define process $\mathsf{null}(n) \vdash n : \wedge\mathsf{Node}(A)$ that chooses #Null on session n and then closes, and process $\mathsf{next}(a, l, n) \vdash a : \vee\overline{A}, l : \overline{\mathsf{LL}(A)}, n : \wedge\mathsf{Node}(A)$, that chooses #Next, sends an element $a : \vee\overline{A}$ and continues as a linked list $n : \mathsf{LL}(A)$ that forwards to $l : \overline{\mathsf{LL}(A)}$

$$\begin{array}{lcl} \mathsf{null}(n) & \triangleq & \textbf{affine}\ n; \#\mathsf{Null}\ n; \textbf{close}\ n \\ \mathsf{next}(a, l, n) & \triangleq & \textbf{affine}\ n; \#\mathsf{Next}\ n; \textbf{send}\ n(a); \textbf{fwd}\ n\ l \end{array}$$

With those auxiliary definitions, we code in Fig. 4.1 a linked list $l : \mathsf{LL}(A)$ that stores two elements $a_1 : \vee\overline{A}$ and $a_1 : \vee\overline{A}$. The upper-left shows code for a linked list with two

```
cut {
  cell c2(n2.next(a2, c1, n2))
  |c1|
  cell c1(n1.next(a1, c0, n1))
  |c0|
  cell c0(n0.null(n0))
}
```




```
1 :  append(l1, n2, l) ≜
2 :    take l1(n);
3 :    use n;
4 :    case n {
5 :      |#Null :   wait n;
6 :                 put l1(n'.fwd n2 n');
7 :                 fwd l1 l
8 :      |#Next :   recv n(a);
9 :                 cut {
10 :                  append(n, n2, l')
11 :                   |l'|
12 :                  put l1(n'.next(a, l', n'));
13 :                  fwd l1 l}
14 :   }
```

Figure 4.1: A linked list with append in-place.

elements a_2, a_1. In the bottom we draw the corresponding diagrammatic representation: blue boxes represent reference cells, black boxes other sessions which are not stateful. The dashed box represents the null node.

In Fig. 4.1, on the right side, we also define a process

$$\mathsf{append}(l_1, n_2, l) \vdash l_1 : \overline{\mathsf{LL}(A)}, n_2 : \vee\overline{\mathsf{Node}(A)}, l : \mathsf{LL}(A)$$

that updates in-place linked list l_1 by appending node n_2 to its tail, outputting the updated list in session l.

Process $\mathsf{append}(l_1, n_2, l)$ takes the node n stored in l_1, uses n and then performs case analysis (ln. 2-4). If #Null, it waits for n to be closed and puts n_2 in the list l_1, after which l_1 is forwarded to l (ln.5-7). This corresponds to the base case in which list l_1 is empty.

The coinductive step, in which l_1 has at least one element is processed when n chooses #Next. Then, it it receives the head element $a : \vee\overline{A}$ on session n (ln. 8), recursively appends n_2 to the back of the linked list $n : \overline{\mathsf{LL}(A)}$ (ln. 10) and puts on l_1 a node whose head element is a and whose tail is the result l' of the recursive call (ln. 12), after which it forwards the updated usage l_1 to l (ln. 13).

Linked lists are implemented in file *state/linked-lists.clls* where we also define further operations such as insertion sort and also define some tests. We also have an implementation of binary search trees in file *state/binary-search-trees.clls*.

4.2 A Concurrent Imperative Queue

We code a shareable mutable imperative queue which supports concurrent enqueueing and dequeueing. The queue is based on a linked list and two independent pointers: one points to the head of list, used for dequeueing, and another points to the tail of the list,

used for enqueueing. Each pointer is represented as session of type $X = \mathbf{S}_f\ \mathsf{LL}(A)$, i.e. as a reference cell to a linked list.

Each pointer can be manipulated by an arbitrary number of concurrent threads. In particular, when the queue is nonempty, each pointer can operate simultaneously without blocking each other. In Fig. 4.2 we draw a diagram illustrating an imperative queue with three elements. Head (h) and tail pointers (t) are used for dequeueing and enqueueing, respectively. They can be shared by an arbitrary number of threads and operate concurrently when the queue is nonempty. We stress that they are connected (represented with a dash line) with a par ($\parr$) and not with a tensor ($\otimes$) since they share a common structure, hence are not disjoint.

The queue offers two ADT interfaces $\mathsf{EnqT}(A)$ and $\mathsf{DeqT}(A)$: one for enqueueing and another for dequeueing, respectively. Each interface is associated with the corresponding head or tail pointer, the type definition of each is given in Fig. 4.3.

The representation type $X = \mathbf{S}_f\ \mathsf{LL}(A)$ of each interface is hidden through an existential quantification [35, 109] and, as compliant with an OOP paradigm, an object of each interface will be comprised of reference cell X that points to a given node of the linked list and which represent the object's state, and a collection of persistent (typed by a bang !) methods - $\mathsf{!MenuE}(A, X)$ for enqueueing and $\mathsf{!MenuD}(A, X)$ for dequeueing - that act on X and represent the object's interface menu. The clients are then forced to manipulate their pointers using the interface methods. Session type

$$\mathsf{Queue}(A) \triangleq \mathsf{EnqT}(A) \parr \mathsf{DeqT}(A)$$

pack together the two ADTs interfaces

Fig. 4.3 also defines the menu for each interface. Each interface can be released and shared by an arbitrary number of concurrent threads, through options #Free and #Share, which are common to both menus. Those options are uniformly implemented by the process methods $\mathsf{mfree}(m)$ and $\mathsf{mshare}(m)$:

$$\begin{array}{lcl}
\mathsf{mfree}(m) & \vdash & m : X \multimap \mathbf{1} \\
\mathsf{mfree}(m) & \triangleq & \mathbf{recv}\ m(c); \mathbf{par}\ \{\mathbf{release}\ c \parallel \mathbf{close}\ m\} \\
 & & \\
\mathsf{mshare}(m) & \vdash & X \multimap (X \parr X \parr \bot) \\
\mathsf{mshare}(m) & \triangleq & \mathbf{recv}\ m(c); \mathbf{recv}\ m(c_1); \mathbf{recv}\ m(c_2); \mathbf{wait}\ m; \\
 & & \mathbf{share}\ c\ \{\mathbf{fwd}\ c\ c_1 \parallel \mathbf{fwd}\ c\ c_2\}
\end{array}$$

Process $\mathsf{mfree}(m)$ receives on session m an usage $c : \mathbf{U}_f\ \overline{\mathsf{LL}(A)}$, then releases c and, in parallel, closes the communication session m with the client, therefore $\mathsf{mfree}(m)$ releases each inputted usage c when no longer needed.

Process $\mathsf{mshare}(m)$ receives on session m usage $c : \mathbf{U}_f\ \overline{\mathsf{LL}(A)}$, inputs two cells c_1, c_2, both of type $\mathbf{S}_f\ \mathsf{LL}(A)$, waits for the client to close m and proceeds as an operation that shares c between two threads: one forwards c to c_1, whereas the other forwards c to c_2. As a consequence, all reference cell usages on c_1 and c_2 will be redirected to common cell c.

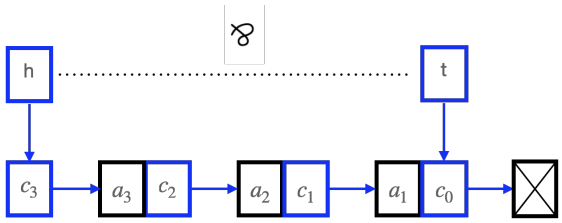


Figure 4.2: A queue with three elements: a_3, a_2 and a_1.

$$
\begin{array}{lcl}
\mathsf{EnqT}(A) & \triangleq & \exists X.!\mathsf{MenuE}(A, X) \otimes X \\
\\
\mathsf{MenuE}(A, X) & \triangleq & \&\,\{ \\
 & & \quad |\#\mathsf{Enq} : X \multimap \wedge A \multimap (X \otimes \mathbf{1}), \\
 & & \quad |\#\mathsf{Share} : X \multimap (X \parr X \parr \bot), \\
 & & \quad |\#\mathsf{Free} : X \multimap \mathbf{1} \\
 & & \} \\
\\
\mathsf{DeqT}(A) & \triangleq & \exists X.!\mathsf{MenuD}(A, X) \otimes X \\
\\
\mathsf{MenuD}(A, X) & \triangleq & \&\,\{ \\
 & & \quad |\#\mathsf{Deq} : X \multimap (\mathsf{Maybe}(\wedge A) \otimes X \otimes \mathbf{1}), \\
 & & \quad |\#\mathsf{Share} : X \multimap (X \parr X \parr \bot), \\
 & & \quad |\#\mathsf{Free} : X \multimap \mathbf{1} \\
 & & \}
\end{array}
$$

Figure 4.3: Concurrent imperative queue: interfaces for enqueueing and dequeueing.

Methods

$$
\begin{array}{lcl}
\mathsf{menq}(m) & \vdash & m : X \multimap \wedge A \multimap (X \otimes \mathbf{1}) \\
\mathsf{mdeq}(m) & \vdash & m : X \multimap (\mathsf{Maybe}(\wedge A) \otimes X \otimes \mathbf{1})
\end{array}
$$

where $X = \mathbf{S}_f\ \mathsf{LL}(A)$, implementing the menu options #Enq for enqueueing and #Deq for dequeueing, respectively, are defined in Fig. 4.4.

Method $\mathsf{mdeq}(m)$ is non-blocking, always returning a session of type $\mathsf{Maybe}(B)$. It receives on m both the reference cell usage $\mathsf{tail} : \mathbf{U}_f\ \overline{\mathsf{LL}(A)}$ that points to the tail of the list and the element $a : \vee\overline{A}$ to enqueue (ln. 2-3). Then, it takes the list $l : \vee\overline{\mathsf{LL}(A)}$ stored in the tail, which is wrapped as coaffine session and therefore needs to be explicitly used (ln. 4-5), after which we can take the empty node $n : \vee\mathsf{Node}(A)$ from l (ln. 6). A new list l' storing the empty node n is created (ln. 7), which is then shared between two threads (ln. 9). In one of the threads we update l, which was previously empty, with a new node n' that stores a and continues as the empty node n, after which we release usage l (ln. 10-11). In the other thread we update the tail, so that it stores the newly created empty list l', sends it back on session m and then closes m (ln. 13-15).

Process $\mathsf{mdeq}(m)$ receives on session m the reference cell usage $\mathsf{head} : \mathbf{U}_f\ \overline{\mathsf{LL}(A)}$ (ln. 2) that points to the head of the linked list and unpacks the structure, obtaining the list l and the node n stored in l (ln. 3-6). Then, it performs a case analysis on n (ln. 7). If the node

```
1 :  menq(m) ≜
2 :    recv m(tail);
3 :    recv m(a);
4 :    take tail(l);
5 :    use l;
6 :    take l(n);
7 :    cut {cell l'(x.fwd x n)
8 :        |l'|
9 :      share l'{
10 :       put l(n'.next(a, l', n'));
11 :       release l
12 :       ||
13 :       put tail(y.affine y; fwd y l');
14 :       send m(tail);
15 :       close m}}
```

```
1 :  mdeq(m) ≜
2 :    recv m(head);
3 :    take head(l);
4 :    use l;
5 :    take l(n);
6 :    use n;
7 :    case n {
8 :      |#Null : wait n;
9 :               put l(n'.null(n'));
10 :              put head(l'.affine l'; fwd l l');
11 :              send m(h.Nothing h; close h);
12 :              send m(head);
13 :              close m
14 :
15 :     |#Next : par {put l(n'.null(n'));
16 :              release l
17 :               ||
18 :              recv n(a);
19 :              put head(l'.affine l'; fwd l' n);
20 :              send m(h.Just h; fwd h a);
21 :              send m(head);
22 :              close m}}
```

Figure 4.4: Concurrent imperative queue: methods menq and mdeq.

chooses #Null, a null node n' is put back on l and the empty list l is restored to the head (ln.9-10), after which Nothing is sent to the client (ln. 11) on m. On the other hand, if the node chooses #Next, we receive the head element a (ln. 18) and update the head so that it points to the next list n (ln. 19). Element a is sent to the client, wrapped as a Just (ln. 20). In parallel, it releases the usage l (ln. 16), which previously corresponded to the head, but before releasing we need to put back some dummy null node n' (ln. 15). Both branches of the case end by sending the updated head on m, after which m is closed.

We have also coded in CLASS a variation of this example, inspired by the multicast operation of the buffered channel implementation given in [102]. The variation extends the menu for dequeueing with a method mdup(m), which allows two dequeue usages to be *duplicated*, i.e. when when one element is dequeued through one of the usages, it is still available to the other. This contrasts with the already defined method mshare(m) for sharing two dequeue usages: when two dequeue usages are shared, a dequeue operation on one of the usages will consume the head element, which will no longer be available to the other.

In other words, method mshare(m) shares a single dequeueing pointer, whereas mdup(m) produces two independent dequeueing pointers which share the same underling linked list structure.

Crucially, for this implementation to work, method $\mathsf{mdeq}(m)$ (Fig. 4.4) for dequeueing needs to be adapted. On each dequeue usage, the head element of the linked list to which the dequeue usage points is copied. One of the copies is sent to the invoking client of the method $\mathsf{mdeq}(m)$, after which the dequeueing pointer moves by one position. The other copy is used to restore the head element to the linked list, so that it is still available to possible independent duplicated dequeue usages.

Since we need to copy elements of the linked list, this implementation only works with linked lists $\mathbf{S}_f\ \mathsf{LL}(!A)$ that store exponentials $!A$. The dequeue operation can then be straightforwardly implemented with the exponential read operation, which was introduced previously in Example 13.

On the other hand, method

$$\mathsf{mdup}(m) \vdash m : X \multimap (X \parr X \parr \bot), \text{ where } X = \mathbf{S}_f\ \mathsf{LL}(!A)$$

that duplicates a pointer m is defined by

$$\begin{array}{lll} \mathsf{mdup}(m) & \triangleq & \mathbf{recv}\ m(c); \mathbf{recv}\ m(c_1); \mathbf{recv}\ m(c_2); \mathbf{take}\ c(l); \mathbf{use}\ l; \mathbf{wait}\ m; \\ & & \mathbf{share}\ l\ \{ \\ & & \quad \mathbf{put}\ c(l'.\mathbf{affine}\ l'; \mathbf{fwd}\ l'\ l); \mathbf{fwd}\ c\ c_1 \\ & & \quad \| \\ & & \quad \mathbf{cell}\ c_2(l'.\mathbf{affine}\ l'; \mathbf{fwd}\ l' l\ )\} \end{array}$$

It inputs on m an usage c for dequeueing and produces two duplicated usages c_1, c_2. The linked list l stored in c is shared between two concurrent threads: one puts back the list l on c and forwards c_1 to c, whereas the other creates on c_2 a new cell that stores l.

Code for the imperative queue (server side and client) as well as collection of tests can be found in *state/imperative-queue*. The variation in which exponential cells are stored and which allows the head pointer to be duplicated can be found in *state/imperative-queue-B*.

4.3 Dining Philosophers

The purpose of this example is to illustrate how we can handle resource synchronisation scenarios in CLASS, more specifically we address the famous dining philosophers problem. Dijkstra's resource hierarchy solution [51] ensures deadlock-freedom by imposing a partial order on usage of shared resources.

In our solution, we show how this partial order can be captured in CLASS by a simple passive linked data structure. Remarkably, deadlock-freedom is then guaranteed purely by typing and follows by the basic acyclicity of linear logic (manifest in the typing rules [Tcut], [Tsh], [TshL], [TshR]), without the need to rely on extra-logical devices.

Let us recall the problem: there are k ($k \geq 2$) philosophers - $P_0, P_1, \ldots, P_{k-1}$ -sitting in a round table and k forks -$f_0, f_1, \ldots, f_{k-1}$, a fork between each pair of philosophers. Philosophers alternate between eating and thinking, in order to eat they must acquire their two neighbouring forks. Namely each philosopher P_i, with $0 \leq i < k-1$ must acquire forks

f_i, f_{i+1}; philosopher P_{k-1} must acquire forks f_0 and f_{k-1}. After eating, each philosophers drops the forks and starts to think. No pair of two philosophers can communicate, the problem is then to design a decentralised fork-acquiring policy so as to avoid deadlock.

Dijkstra's resource hierarchy solution works by requiring each fork to be acquired by ascending order, i.e. if a philosophers acquires first f_i and and then f_j we have $i < j$. In other words, each philosopher P_i with $0 \leq i < k-1$ acquires first its right and left fork, whereas philosophers P_{k-1} is the symmetry-breaker that must acquire first its left and only then its right fork.

In order to code this solution in CLASS, the key idea is to represent the order in which the forks must be acquired by an explicit acyclic linked chain. More specifically, let us define the following recursive datatype structure

$$\mathsf{Fork} \triangleq \mathbf{S}_f\ \mathsf{Node} \quad \mathsf{Node} \triangleq \wedge \oplus \{\#\mathsf{Null} : \mathbf{1}, \#\mathsf{Next} : \mathsf{Fork}\}$$

A process offering the recursive type Fork behaves as a cell which stores a node of session Node. A session of type Node either chooses #Null, in which case it closes; or chooses #Next, in which case it continues as the next Fork. Define the basic operations $\mathsf{null}(n) \vdash n : \wedge\mathsf{Node}$ and $\mathsf{next}(f, n) \vdash f : \overline{\mathsf{Fork}}, n : \wedge\mathsf{Node}$ by

$$\mathsf{null}(n) \triangleq \mathbf{affine}\ n; \#\mathsf{Null}\ n; \mathbf{close}\ n \quad \mathsf{next}(f, n) \triangleq \mathbf{affine}\ n; \#\mathsf{Next}\ n; \mathbf{fwd}\ n\ f$$

A resource hierarchy on k forks $f_0 < f_1 < \ldots < f_k$ is then represented in CLASS by the passive linked chain

$$\mathbf{cut}\ \{\mathbf{cell}\ f_0(n_0.\mathsf{next}(f_1, n_0))\ |f_1|\ \mathbf{cell}\ f_1(n_1.\mathsf{next}(f_2, n1))\ |f_2|\ \ \ldots\ |f_k|\ \mathbf{cell}\ f_k(n_k.\mathsf{null}(n_k))\}$$

If a philosopher is granted access to a fork f_i, he can then access any fork f_j, with $i < j$, by traversing the pointed structure. Crucially, he must follow the path dictated by the chain and hence, cannot acquire forks f_j with $j < i$.

The chain on forks can then be shared by an arbitrary number of dining philosophers by applying cocontraction. Cocontraction ensures that philosophers do not communicate between themselves, any synchronisation happens through the shared passive ordered structure on forks.

In Fig. 4.5, on the left, we show how an example with a round table of four philosophers more specifically with two logicians - William Howard (P_0) and Jean-Yves Girard (P_3) - and two computer scientists - Robin Milner (P_1) and Kohei Honda (P_2), and the four forks numbered from 0 to 3. The order on the forks is captured in CLASS by an acyclic linked chain, represented on the right. A fork is represented with a blue box, the dashed black box represents the end of the chain. Philosophers P_0, P_1, P_2, P_3 have access to a particular fork of the chain as represented by the dashed links. Each philosophers P_0, P_1, P_2 eats by acquiring consecutive forks in the chain. The symmetry-breaker philosopher P_3 must eat by acquiring first the head fork f_0 and then needs to traverse the whole structure in order to acquire the last fork f_3.

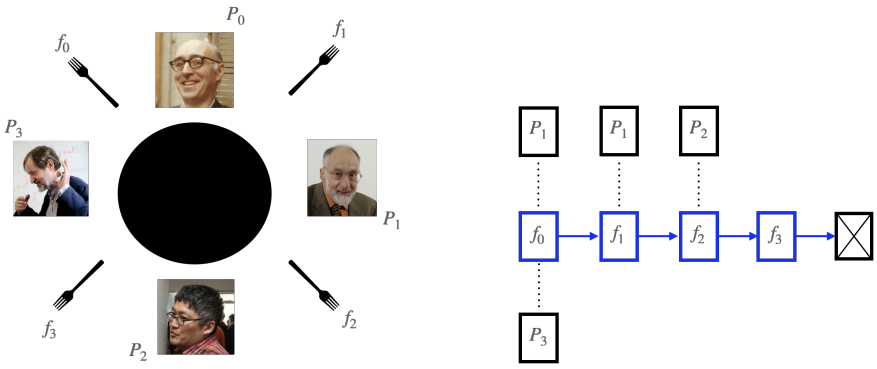


Figure 4.5: Solution to the dining philosophers problem in CLASS.

We will now describe in detail processes that implement eating for a system with k philosophers. Each philosopher P_i, with $0 \leq i < k-1$ eats by acquiring two consecutive forks, whereas philosopher P_{k-1} eats by acquiring the first f_0 and last fork f_{k-1}. Eating for a philosopher P_i, with $0 \leq i < k-1$ is implemented by process eat, whereas eat2 implements eating for philosophers P_k (Fig. 4.6).

Process $\mathsf{eat}(f, f') \vdash f : \overline{\mathsf{Fork}}, f' : \mathsf{Fork}$ takes fork f on parameter n (ln. 2), then it performs case analysis to check if n is #Null or #Next (ln. 4). If #Null it simply puts null back on f and forwards to f', this is done by auxiliary process $\mathsf{putNull}(f, f')$ (ln. 6).

The interesting case occurs when n selects #Next, in which case a further take occurs (ln. 7). The philosophers has now acquired the two forks - f and n - and can eat, we left code for eating unspecified. Then, the philosopher puts back the two forks (ln. 9-10), while preserving the original chain structure, and forwards fork f to f' (ln. 11) which can then be used by subsequent rounds of eating.

In a system with n philosophers, philosopher $P_n(f_0)$ has access to the head of the chain but eats in a distinct way, as described by process $\mathsf{eat2}(f, f') \vdash f : \overline{\mathsf{Fork}}, f' : \mathsf{Fork}$ in Fig. 4.6. As it happened before, it starts by taking the first fork which contains always a nonempty node, but then it has to take the last fork in the chain, which is done with the auxiliary process $\mathsf{takeLast}(n, x) \vdash n : \overline{\mathsf{Fork}}, x : \mathsf{Fork} \otimes \mathbf{1}$, which recursively traverses the whole chain n until it takes the last fork, after which it eats, puts back the last fork, sends the unmodified structure n on x and closes.

Inevitably, the way in which philosopher P_k eats has to be distinct so as to ensure the key symmetry-breaking, implicit in Dijkstra's original resource hierarchy solution: philosophers $P_0, \ldots, P_{k-1}$ all pick the first the left and then the right forks, whereas philosopher P_k starts by picking the right forks and then picks the left.

Interestingly, further partial orders on resources, other that simple chains, can also be explored, for example: we can model directed trees simply by redefining Node as a linked list of forks.

Code for this example can be found in *state/dining-philosophers.cls*.

```
1 : eat(f, f') ≜
2 :   take f(n);
3 :   use n;
4 :   case n {
5 :     |#Null :  wait n;
6 :               putNull(f, f')
7 :     |#Next :  take n(m);
8 :               ...//eating
9 :               put n(m'.fwd m m');
10 :              put f(n'.next(n, n'));
11 :              fwd f f'}

1 : eat2(f, f') ≜
2 :   take f(n);
3 :   use n;
4 :   case n {
5 :     |#Null :  wait n;
6 :               putNull(f, f')
7 :     |#Next :  cut {takeLast(n, x)  |x|
8 :               recv x(m);
9 :               wait x;
10 :              put f(n'.next(m, n'));
11 :              fwd f f'}}

1 : putNull(f, f') ≜
2 :   put f(n.null(n));
3 :   fwd f f'

1 : takeLast(f, x) ≜
2 :   take f(n);
3 :   use n;
4 :   case n {
5 :     |#Null :  wait n;
6 :               ...//eating
7 :               put f(n'.null(n'));
8 :               send x(f);
9 :               close x
10 :    |#Next :  cut {takeLast(n, y)  |y|
11 :              recv y(m);
12 :              put f(n'.next(m, n'));
13 :              wait y;
14 :              send x(f);
15 :              close x}}
```

Figure 4.6: Dining philosophers: implementation details.

4.4 A Shared Resource-Invariant Toggle

The semantics of CLASS reference cells and the basic share operation are sufficient to express general corecursive protocols that alternate between a linear and a shared phase. That is, shareable protocols that satisfy some resource invariant, which may be acquired, to be interacted with linearly, and then released, when the invariant upholds, to be shared again. This is a standard technique to implement and reason about resource sharing (see, e.g., [68, 21, 10]).

We illustrate with a basic toggle example. The toggle has two states - On and Off - the resource invariant is the state Off, which can be shared. The protocol is expressed by the following pair of mutually corecursive type definitions

$$\mathsf{Off} \triangleq \&\{\#\mathsf{turnOn} : \mathsf{On}\} \qquad \mathsf{On} \triangleq \&\{\#\mathsf{turnOff} : \wedge\mathsf{Off}\}$$

Type Off offers the single option #turnOn, after which it evolves to On. Conversely, type On offers the single option #turnOff, after which it evolves to an affine Off. Wrapping the state Off within an affine modality allows the toggle to be shared in the invariant type and, furthermore, provides a termination base case for a recursive process interacting with the toggle, which corresponds to discarding the affine session.

```
off(t)   ⊢  t : Off
off(t)   ≜  case t {|#turnOn : on(t)}

on(t)    ⊢  t : ∧On
on(t)    ≜  case t {|#turnOff : affine t; off(t)}

main()   ⊢  ∅
main()   ≜  cut {
              cell c(t.affine t; off(t))
              |c|
              share {
                take c(t);
                use t; #turnOn t; #turnOff t;
                put c(t'.fwd t t'); release c
                ||
                take c(t);
                use t; #turnOn t; #turnOff t;
                use t; #turnOn t; #turnOff t;
                put c(t'.fwd t t'); release c
              }
            }
```

Figure 4.7: Code for a shared resource-invariant toggle.

In Fig. 4.7, on the left-hand side, we show code for mutually corecursive processes on(t) and off(t), that follow automatically from the type definitions On and Off. More interestingly, the right-hand side displays code for a toggle, that is being shared between two clients. Each client acquires the toggle, with a take operation, in the Off invariant type and then uses the linear protocol freely, by calling #turnOn and #turnOff in strict alternation. After this linear interaction, they release the toggle (in the invariant type) by putting it back in the cell c.

Code for this example is available in *examples/state/toggle.clls*.

In *examples/state/shared-factory.clls*, we also provide code for a generic (polymorphic) wrapper factory. that for any affine object, provides a coercion to a general invariant-based sharing interface, all written in CLASS.

4.5 A Barrier for N threads

We describe in Fig. 4.8 a CLASS implementation of a simple barrier, parametric on the number N of threads to synchronise. We find it interesting to model the "real" code shown in the Rust reference page for std::sync::Mutex [86]. The code uses if-then-else and primitive integers, supported by our implementation, but that could be defined as idioms of pure CLASS processes.

We represent a barrier by a mutex cell storing a pair consisting of an integer n, holding

```
init(w_s) ⊢ w_s : ∧BState
init(w_s) ≜
  affine w_s; send w_s(N); affine w_s; nil(w_s)

awakeAll(w_s : List(∧⊥))
awakeAll(w_s) ≜
  case w_s {
    #Nil : wait w_s; 0
    #Cons :
    recv w_s(w);
    par {use w; close w || awakeAll(w_s)}

spawnAll(c; i, n) ⊢ c : Barrier; i : Int, n : Int
spawnAll(c; i, n) ≜
  if (n == 0) { release c}
  { share c {
      thread(c; i)
      ||
      spawnall(c; i + 1, n − 1)}}
```

```
thread(c; i) ⊢ c : Barrier; i : Int
thread(c; i) =
  println i + ": waiting.";
  take c(w_s); recv w_s(n); use w_s;
  if (n == 1) {
    par {
      println i + ": finished.";
      use w_s; awakeAll(w_s)
      ||
      put c(w'_s.init(w'_s));
      release c}}
  { cut {
      affine w; wait w;
      println i + ": finished."; 0
      |w| put c(w'_s.affine w'_s;
                send w'_s(n − 1);
                affine w'_s;
                cons(w, w_s, w'_s));
      release c}}
```

Figure 4.8: A Barrier for N Threads

the number of threads that have not yet reached the barrier, and a stack s of waiting threads, each represented by a session of *affine* type $\wedge\bot$ (so they will be safely aborted if at least one thread fails to reach the barrier).

The type Barrier of the barrier is $\mathbf{S}_f$ BState, where BState $\triangleq$ Int $\otimes \wedge$List$(\wedge\bot)$. Initially the barrier is initialised with $n = N$ threads and an empty stack, so that the invariant $n + depth(s) = N$ holds during execution. Each thread$(c; i)$ acquires the barrier c and checks if it is the last thread to reach the barrier (if $n == 1$): in this case, it awakes all the waiting threads (awakeAll(w_s)) and resets the barrier. Otherwise, it updates the barrier by decrementing n and pushing its continuation into the stack (the continuation for thread i just prints "finished"). The following process main() $\vdash \emptyset$ creates a new barrier c and spawns N threads, each labelled by a unique id i: main() $\triangleq$ **cut** { **cell** $c(w_s$.init$(w_s))$ $|c|$ spawnAll$(c; 0, N)$ }. Again, our type system statically ensures that the code does not deadlock or livelock.

Code for this example is available in *examples/state/barrier.clls*. We provide further synchronisation methods in the distribution such as fork-joins and Hoare style monitors.

5

Language Implementation CLLSj

5.1 Introduction

The session-typed process calculus CLASS (Chapter 3), based on the correspondence with classical linear logic and extended with first-class mutable affine-stored state and state sharing, has a practical language implementation, dubbed CLLSj.

CLLSj consists of a type checker and interpreter, both of which were implemented in the Java programming language (~ 8k loc). CLLSj demonstrates the feasibility of our propositions-as-types approach to mutable shared state and, pragmatically, validates and guides the development of complex programs written in CLASS. In fact, all the examples presented so far were validated by CLLSj type checker and tested using its interpreter.

The implementation can be dowloaded at [133]. It comes with detailed use instructions (*README.md*), the source code (folder *src/*) and an extensive (~ 6k loc) collection of examples (folder *examples/*). The file *README.md* helps you get started, shows you how to interact with the CLLSj read-eval-print loop (REPL) and also presents the full concrete syntax of the implemented language, mapping whenever necessary to the theoretical model CLASS.

In the folder *examples/* you may find several realistic examples which showcase the expressiveness of our approach: we code inductive datatypes (naturals, lists) using either primitive recursion types or using system-F style encodings, linked data structures (linked lists, binary search trees), shareable concurrent mutable ADTs (bank accounts, stacks, functional and imperative queues), resource synchronisation methods (fork-joins, barriers, dining philosophers) and several test suites. Folder *examples/* is divided into two subfolders: in *examples/pure* we list all the examples of the pure sublanguage μCLL (Chapter 2), based on the correspondence with classical linear logic; all the examples in *examples/state* involve mutable state.

The metatheoretical properties of CLASS, like deadlock-freedom (Chapter 6) and strong normalisation (Chapter 9), provide strong guarantees for our implementation CLLSj: the interpreter never blocks and always terminates when running type checked programs. This is true even when running complex examples involving the concurrent

shareable ADTs, where everything is implemented using pure session-based processes, from the high-level ADTs menus to the basic inductive datatypes such as lists, where the implementation spawns thousands of short-lived threads that synchronise perfectly thanks to linear logic.

The rest of the chapter is organised as follows. In Section 5.2 we go step-by-step through the implementation of a simple "hello world" concurrent counter in order to illustrate some features and details of CLLSj. Then, Section 5.3 gives a high-level overview of the implementation of the type checker and interpreter. Finally, 8.4 concludes with further discussion and related work.

5.2 Hello World: A Concurrent Counter

We will go step-by-step through the implementation of a simple "hello world" concurrent counter with two operations to get-and-reset and increment. The purpose is to illustrate some features and details of our language implementation CLLSj. Once in the main directory execute ./CLLSj to run the interactive REPL.

A counter is just a reference cell that stores an integer, the datatype Counter is defined with the following instruction

```
type Counter { state lint };;
```

This instructs the REPL to associate the type id Counter with the type expression state lint, which denotes a state full of a linear integer. After typing the above expression the REPL prints

```
Type Counter: defined.
```

informing that the new datatype Counter was successfully defined.

All the instructions to interact with the REPL end with a double semi-colon, processes and type expressions are enclosed in curly braces. The keyword state is the concrete syntax expression for the modality $\mathbf{S}_f$ from CLASS. We denote the empty state modality $\mathbf{S}_e$ with the concrete expression statel which stands for *state locked*. Similarly, the concrete syntax expressions usage and usagel stand for the usage modalities $\mathbf{U}_f$ and $\mathbf{U}_e$ of CLASS, respectively.

We will now define a process named counter, which offers the protocol Counter on a session c

```
proc counter(c: Counter){ cell c(n: affine lint. affine n; let n 42) };;
```

This instructs the REPL to type check the process expression enclosed in curly braces with the typing context c:Counter and, provided type checking is successful, to associate it the id counter(c).

In this case the typing context is just made of a linear part which associates the name id c with the previously defined type Counter. We use a semicolon to separate the linear and the unrestricted typing context as in n:Nat ; b:Bool. When the unrestricted typing context is empty we do not need to use the semicolon, writing c:Counter instead of c:Counter;.

The process expression cell c(n: affine lint. affine n; let n 42) denotes a reference cell on session c that is storing an affine linear integer n. The affine linear integer is initially set to 42. In our concrete process language CLLSj, terms are written with bound names type-annotated, which from a pragmatic point of view guides the type checking algorithm and eases the task of writing complex programs.

The duality related pair of modalities $\wedge/\vee$ is denoted in the language implementation with the terms affine/ coaffine. The keyword affine is also used in the process term language to introduce affine sessions in the example above, where we are defining an affine session on n and continuing as let n 42, an expression that defines on session n the linear integer 42.

For convenience, the implementation language CLLSj also includes efficient pragmatic basic datatypes (native integers, booleans, strings), ML-style let expressions and primitive operations (arithmetic sum, if-conditional, string concatenation, to name a few). These basic datatypes and primitive operations can, anyway, be encoded in the fundamental classical linear logic based pure session-typed calculus μCLL. For example, we implement the booleans and the natural datatypes in *examples/pure/booleans.clls* and *examples/pure/naturals.clls*.

After typing the above expression, the REPL outputs

```
Process counter: defined.
```

informing the process counter was defined, this implies that type cheking was successful.

We will now define some basic processes that operate in the counter. We start with the following operation

```
proc getAndReset(c: ~ Counter){
    take c(n: coaffine colint);
    use n;
    println("GOT␣"+n+"␣and␣RESET");
    put c(v: affine lint. affine v; let v 0);
    release c
};;
```

It gets the linear integer stored in the counter c, prints the integer and resets the counter. It type checks with c:~ Counter. The type concrete expression ~ A in CLLSj denotes the dual type $\overline{A}$ in CLASS. Instead of writing ~ Counter we could alternatively have written usage colint, but we let the type checker carry out the computation of the dual type.

Process getAndReset(c) starts by taking the usage c on session n: coaffine colint. Notice how the type annotation can inform us what to do next: we have to use the take session to

unstrip the coaffine modality after which we get a session n of type colint. Now we must do some operation that consumes the linear integer, which, in this case, is done by the print instruction. Finally, we put the affine linear 0, and then release the usage c.

The increment operation is defined by

```
proc inc(c: ~ Counter){
    take c(n: coaffine colint);
    println "INC";
    put c(v: affine lint. affine v; use n; let v n+1);
    release c
};;
```

It also type checks with c:~ Counter and follows a similar usage pattern, already described for getAndReset(c). It prints a message "INC" informing that the increment operation is being carried out. The taken linear integer in this case is consumed by the let expression to define the incremented value being put. After feeding the REPL with the above process definitions we obtain:

```
Process getAndReset: defined.
Process inc: defined.
```

To conclude, we assemble everything in the following process definition

```
proc system(){
    cut{
        counter(c)
        |c: ~ Counter|
         share c {
             inc(c) || getAndReset(c) || inc(c) || getAndReset(c)
        }
}};;
```

This composes a counter with 4 atomic threads, two of which are incrementing and another two are printing and resetting.

After typing the expression the REPL prints

```
Process system: defined.
```

Once a closed process (with an empty typing context) is type checked we can run it multiple times by writing the process name. For example, if we input the REPL with

```
> system();;
```

it *may* print the log

```
INC
INC
GOT 44 and RESET
GOT 0 and RESET
```

The emphasis is to stress the fact that the printed log depends on the scheduling of the atomic actions. For example, the above log corresponds to the scheduling in which the two atomic increment instructions were executed before the two get-and-reset instructions. If we run process system() again we might obtain

```
GOT 42 and RESET
INC
INC
GOT 2 and RESET
```

This corresponds to the scheduling in which the two increment instructions are executed after and before a get-and-reset.

Our implementation exposes nondeterminism arising from concurrency as real committed nondeterminism, so that the sum operator from the metatheoretical model CLASS, while crucial to establish a propositions-as-types model and to characterise the semantics of our language, is not present in our practical runtime system CLLSj. On the other hand, the metatheoretical model allows us to reason about the behaviour of concurrent programs in the implementation(as illustrated in Example 15) since the result produced by running a process in the implementation CLLSj is always a summand of a sum, obtained by $\rightarrow$-reducing the process in the metatheoretical model CLASS.

Complete code for this example can be found in *examples/state/simple-counter.clls*. The whole file can be type checked at once by typing

```
> include ‘‘examples/state/simple-counter.clls’’;;
```

5.3 Type Checker and Interpreter

We will now give a high-level overview of how the type checker and interpreter were implemented. The source code can be consulted in *src/*.

When type checking linear logic based type systems we need to guess how to split the linear typing context of the conclusion between the premises in the multiplicative rules, like [T$\otimes$], which types session output. The type checker of CLLSj deals with this issue by doing a lazy guess, as in [71], so we pass the context to the recursive type checker call on one of premises, which consumes some linear sessions, and then we pass the remaining linear context to the recursive type checker call on the other premise. In the end we check

that the linear typing context was fully consumed. The unrestricted typing context is simply shared by reference from the conclusion to the premises.

Type definitions are kind-checked against a global environment of type identifiers, this allows the definition of inductive/coinductive session types. Similarly, type checking of processes is done against a global environment that maps process identifiers to typing contexts and which allows for corecursive process definitions.

Each corecursive process definition must always provide at least one linear parameter which corresponds to the session in which corecursion is done, by convention this is the first parameter. The type checker then ensures that the corecursive call is done in a session that *hereditarily descends* from the first parameter so as to guarantee that corecursion is well-founded. We support an unsafe pattern of corecursion in which these restrictions are lifted and which allows us to type nonterminating programs (see, for example, *examples/pure/loop.clls*). We also support mutually recursive type and process definitions, for example check *examples/pure/naturals.clls* in which we provide a mutually recursive definition of the natural predicates even and odd.

We now move to the interpreter, our implementation is not a high-level "naive" rendering of the formal semantics, but an efficient implementation based on a fine-grain concurrent runtime / abstract machine system, the implementation of which relies on Doug Lea's java.util.concurrent.* package [95] provided since J2SE 5.0

The interpretation of a process depends on a global environment that associates each process identifier with its respective process definition. It depends, as well, on two further environments: one that associates each free linear name of the process being interpreted with a linear session object that mediates linear process interactions; the other maps each unrestricted free name with a server object.

The server object has essentially code for a replicated server process that is concurrently interpreted on each method call invocation. On the other hand, the purpose of a linear session object is twofold: first to mediate each binary linear interaction that occurs exactly between two partners (e.g., close and wait, send and receive) and, secondly, to mediate the cell-usage interactions, which occur between a reference cell and an arbitrary number of cell usage concurrent threads.

Each binary interaction is handled by a blocking channel object that supports two methods for sending and receiving arbitrary objects, send blocks until a receive is invoked and vice-versa: receive blocks until a send is invoked. This allows us to emulate the synchronous rendezvous interaction typical of process calculi in general, and of CLASS in particular.

The linear session typing discipline guarantees that each send will eventually be matched with a receive invocation on the same channel object and vice-versa, thereby guaranteeing progress of the interpretation.

When interpreting some process binary interactions of CLLSj, for example close/wait, the object sent is irrelevant (null), whereas it plays a key role on interpreting some other process interactions, like for example session output/input, where a fresh linear session

object is sent by the interpretation of the output process and received by the interpretation of the input process.

Each cell object has code for the body P of a full reference cell cell(c: affine A. P) which is concurrently invoked with each take method invocation. A take method acquires the cell object, so that further take method invocations block until a put method invocations occurs, in which case the lock is released and the object is updated with new code for the body of the reference cell.

Each cell object keeps a count of the number of threads that are currently sharing it, which is incremented on the interpretation of each share construct and decremented on the interpretation of each release action. When the count reaches zero the cell body is simply discarded.

Forwarding redirects each send and receive method invocation on a channel, as well as each take and put method invocation on a cell object. There is a symmetry in the implementation of binary session interactions in the sense that either the send or receive method invoker can be redirected, whereas in the case of cell-usage interactions this symmetry is lost and we need to redirect each usage operation.

Importantly, structural congruence does not play any explicit role in the implementation model. In particular, our implementation exposes nondeterminism arising from concurrency as real committed nondeterminism, so that the sum operator, while crucial to establish a propositions-as-types model and to characterise the semantics of our language, is not present in our practical runtime system, that is, the implementation commits to one of the summands in sums, while any of such may be nondeterministically picked. Types are not used at runtime, but to orient forwarders at state types, and handle polymorphic type variables.

When running complex examples involving the concurrent shareable ADTs, such as the functional queue (*examples/state/2queue/*), where we intensively use session processes to implement everything, down to basic inductive data types such as lists (along the lines of [161]), our implementation spawns thousands of short-lived processes. In order to reduce the overhead associated with thread creation/destruction we have opted to manage the execution of concurrent tasks through a cached thread pool.

5.4 Further Discussion and Related Work

There are several language implementations of binary and multiparty session types, for example: [113] describes an implementation, in C, that statically checks multiparty session types and [138] gives a representation of session types in Scala as well as library for session-based programming, to name just a few.

The idea of dealing with the multiplicatives of linear logic by context splitting and lazy propagation of context residuals originated in linear logic proof search [71] and has, since then, influenced the implementation of linear logic based programming languages, e.g. [126, 98].

The work [10] provides an implementation of the manifest session-typed calculus SILL_S in Concurrent C0 (CC0) [162]. CC0 is a type-safe C-like imperative language with contracts and session types. Session communication in CC0 is implemented via asynchronous buffered channels: a send never blocks, receive blocks until a message is available. In [10] forwarding is dealt with by redirecting all clients of a shared resource through a chain of forwarding pointers, similarly to what happens in the implementation of CLASS. In [10] shared channels are not deallocated, this is a challenge for their system since, in manifest sharing, weakening of shared resources is not explicitly annotated, as it is in CLASS with the release operation (coweakening). More recently, the work [36] developed an embedding of SILL_S in Rust.

Programs written in the mplementation [10] and the embedded language [36] can block since the source language SILL_S does not guarantee deadlock-freedom. This contrasts with our implementation CLLSj, in which all the executed type checked programs never block and always terminate, as implied by the linear logic based type system of CLASS.

Part II

Metatheory of CLASS

6

Safety: Type Preservation and Progress

6.1 Introduction

In this chapter we prove that the session-typed calculus CLASS satisfies the basic properties of type preservation and progress. Type preservation states that, in CLASS, both structural congruence $\equiv$ and reduction $\longrightarrow$ preserve the typing relation (Theorem 1). This is established by routinely checking that all the defining rules of structural congruence $\equiv$ (Def. 13) and reduction $\longrightarrow$ (Def. 14) are type-preserving. In fact, from each rule of $\equiv/\longrightarrow$, we can *extract* a correspondent conversion on proofs/typing derivations.

On the other hand, progress (deadlock-freedom) is a liveness property which states that all closed processes with a pending action eventually reduce (Theorem 2), thereby guaranteeing that no closed live system will ever get stuck. Deadlock-freedom follows from the basic acyclicity of the linear logic type system of CLASS, manifest in rules [Tcut] (cut) and [Tsh], [TshL], [TshR] (cocontraction), which guarantee that no thread blocks, neither through channel synchronisation nor through cell lock acquisition.

We present the type preservation result in Section 6.2, followed by progress in Section 6.3. Then, we conclude the chapter in Section 6.4 with further discussion and related work.

6.2 Type Preservation

The session-typed process calculus CLASS satisfies the basic property of type preservation (Theorem 1, a.k.a. subject reduction), which states that both structural congruence $\equiv$ and reduction $\longrightarrow$ preserve the typing relation. As a consequence, it implies that well-typed processes always evolve to well-typed processes. On the other hand, from a logical point of view, it ensures that all the algebraic manipulations on processes are justified by logical conversions on proofs / typing derivations. We now state our type preservation result. The complete proof can be found in Appendix A.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_e\ A$ (2) $\mathbf{put}\ x(y.P); Q \vdash_\eta \Delta_1, x : \mathbf{U}_e\ A; \Gamma$
(3) $R \vdash_\eta \Delta_2, x : \mathbf{U}_f\ A; \Gamma$, for some A, Δ_1, Δ_2
(By inverting $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \parallel R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1 = \Delta_{11}, \Delta_{12}$ (5) $P \vdash_\eta \Delta_{11}, y : \wedge\overline{A}; \Gamma$ (6) $Q \vdash_\eta \Delta_{12}, x : \mathbf{U}_f\ A; \Gamma$ (By inverting (2))
(7) $\mathbf{share}\ x\ \{Q \parallel R\} \vdash_\eta \Delta_{12}, \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ ([Tsh], (6) and (3))
(8) $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \parallel R\} \vdash_\eta \Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_e\ A; \Gamma$ ([Tput], (5) and (7))
(9) $\Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_e\ A = \Delta$ ((1) and (4))
(10) $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \parallel R\} \vdash_\eta \Delta; \Gamma$ ((8) and (9))

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_e\ A$ (2) $P \vdash_\eta \Delta_1, y : \wedge\overline{A}; \Gamma$
(3) $\mathbf{share}\ x\ \{Q \parallel R\} \vdash_\eta \Delta_2, x : \mathbf{U}_f\ A; \Gamma$, for some Δ_1, Δ_2, A
(By inverting $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \parallel R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2 = \Delta_{21}, \Delta_{22}$ (5) $Q \vdash_\eta \Delta_{21}, x : \mathbf{U}_f\ A; \Gamma$
(6) $R \vdash_\eta \Delta_{22}, x : \mathbf{U}_f\ A; \Gamma$, for some Δ_{21}, Δ_{22} (By inverting (3))
(7) $\mathbf{put}\ x(y.P); Q \vdash_\eta \Delta_1, \Delta_{21}, x : \mathbf{U}_e\ A; \Gamma$ ([Tput], (2) and (5))
(8) $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \parallel R\} \vdash_\eta \Delta_1, \Delta_{21}, \Delta_{22}, x : \mathbf{U}_e\ A; \Gamma$ ([TshL], (7) and (6))
(9) $\Delta_1, \Delta_{21}, \Delta_{22}, x : \mathbf{U}_e\ A = \Delta$ ((1) and (4))
(10) $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \parallel R\} \vdash_\eta \Delta; \Gamma$ ((8) and (9))

Figure 6.1: Type preservation, example: ≡ rule [PSh].

Theorem 1 (Type Preservation)**.** *The following properties hold*

(1) *If $P \vdash \Delta; \Gamma$ and $P \equiv Q$, then $Q \vdash \Delta; \Gamma$.*

(2) *If $P \vdash \Delta; \Gamma$ and $P \longrightarrow Q$, then $Q \vdash \Delta; \Gamma$.*

Proof. By induction on derivations for $P \equiv Q$ and $P \longrightarrow Q$. Follows by checking that all the conversion rules of structural congruence ≡ (Def. 13) and reduction $\longrightarrow$ (Def. 14) are type-preserving. □

In the proof of Theorem 1(1) we use an axiomatisation of ≡ equivalent to Definition 13 but in which we drop rule [symm] $P \equiv Q \supset Q \equiv P$ and assume that each commuting conversion holds from left to right and right to left. We illustrate the proof with ≡ rule [PSh], which commutes as share with a put operation, in Fig. 6.1. The top part gives the proof for the case in which the rule is applied from left to right, bottom part for the direction from right to left.

In the proof we often need to invert the typing relation. That is, by inspecting the constituent constructors of a process P for which a typing judgement $P \vdash_\eta \Delta; \Gamma$ holds we infer some particularities of the typing contexts Δ and Γ as well as infer typing judgments for the subprocesses of P. For example, in a derivation for $\mathbf{par}\ \{P_1 \parallel P_2\} \vdash_\eta \Delta; \Gamma$ the last rule has to be [Tmix], from which we conclude that there is a partitioning Δ_1, Δ_2 of the typing context $\Delta = \Delta_1, \Delta_2$ such that $P_1 \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$.

The typing relations is not only preserved by structural congruence ≡ but also by reduction $\longrightarrow$, as stated by Theorem 1(2). We illustrate the proof with $\longrightarrow$ rule [$\mathbf{S}_e\ \mathbf{U}_e$], that

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{empty}\ c \vdash_\eta \Delta_1, c : \mathbf{S}_e\ A; \Gamma$
(3) $\mathbf{put}\ c(a.P); Q \vdash_\eta \Delta_2, c : \mathbf{U}_e\ \overline{A}; \Gamma$, for some Δ_1, Δ_2
(By inverting $\mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c : \mathbf{S}_e\ A|\ \ \mathbf{put}\ c(a.P); Q\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1 = \emptyset$ (By inverting (2))
(5) $\Delta_2 = \Delta_{21}, \Delta_{22}$ (6) $P \vdash_\eta \Delta_{21}, a : \wedge A; \Gamma$ (7) $Q \vdash_\eta \Delta_{22}, c : \mathbf{U}_f\ \overline{A}; \Gamma$ (By inverting (3))
(8) $\mathbf{cell}\ c(a.P)\ \vdash_\eta \Delta_{21}, c : \mathbf{S}_f\ A; \Gamma$ ([Tcell] and (6))
(9) $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c : \mathbf{S}_f\ A|\ \ Q\} \vdash_\eta \Delta_{21}, \Delta_{22}; \Gamma$ ([Tcut], (8) and (7))
(10) $\Delta = \Delta_{21}, \Delta_{22}$ ((1), (4) and (5))
(11) $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c : \mathbf{S}_f\ A|\ \ Q\} \vdash_\eta \Delta; \Gamma$ ((9) and (10))

$$\dfrac{\dfrac{}{\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma}\ [\text{Tempty}] \qquad \dfrac{\dfrac{\vdots}{P \vdash_\eta \Delta_1, a : \wedge A; \Gamma} \qquad \dfrac{\vdots}{Q \vdash_\eta \Delta_2, c : \mathbf{U}_f\ \overline{A}; \Gamma}\ [\text{Tput}]}{\mathbf{put}\ c(a.P); Q \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ \overline{A}; \Gamma}}{\mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c : \mathbf{S}_e\ A|\ \ \mathbf{put}\ c(a.P); Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma}\ [\text{Tcut}]$$

$$\rightarrow \dfrac{\dfrac{\dfrac{\vdots}{P \vdash_\eta \Delta_1, a : \wedge A; \Gamma}}{\mathbf{cell}\ c(a.P) \vdash_\eta \Delta_1, c : \mathbf{S}_f\ A; \Gamma}\ [\text{Tcell}] \qquad \dfrac{\vdots}{Q \vdash_\eta \Delta_2, c : \mathbf{U}_f\ \overline{A}; \Gamma}}{\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ \ |c : \mathbf{S}_f\ A|\ \ Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma}\ [\text{Tcut}]$$

Figure 6.2: Type preservation, example: $\rightarrow$ rule [$\mathbf{S}_e\ \mathbf{U}_e$].

models the interaction between an empty cell with a put operation, in Fig. 6.2. The proof yields a correspondent typed principal cut conversion, which we display in the bottom.

Theorem 1 justifies our propositions-as-types approach that connects computation to logic, by mapping all process manipulations of $\equiv$ and $\rightarrow$ to proof conversions and vice-versa. In fact, from each rule of $\equiv/\rightarrow$, we can *extract* a correspondent conversion on proofs/typing derivations, e.g. the process reduction rule $\rightarrow$ [$\mathbf{S}_e\ \mathbf{U}_e$] corresponds to a principal cut conversion on typing derivations displayed in Fig. 6.2.

6.3 Progress

In the previous section we proved type preservation. We will now prove that CLASS enjoys the progress property (Theorem 2), namely that all closed live processes reduce. Progress is a liveness property: it guarantees that closed live processes will never get stuck.

We start by defining what means for a process to be live (Definition 16) and then we introduce an observability predicate (Definition 17) which characterises the potential interactions of an open typed processes and which is instrumental to prove progress compositionally.

Definition 16 (Live Process)**.** *A process P is live if $P = C[\mathcal{A}]$ or $P = C[\mathbf{fwd}\ x\ y]$ for some static context C and action $\mathcal{A}$.*

Intuitively, a process is live if it presents an unguarded action or forwarder waiting to interact, that action lies only under the scope of a static construct (mix, linear or unrestricted cut, share, sum). As a consequence of our linear typing discipline, all the typed processes $P \vdash_\eta \Delta;\Gamma$ that (i) type with a nonempty linear context Δ and (ii) with an empty map η are necessarily live, as established by the following lemma. The latter condition (ii) is necessary so as to exclude processes variables $X(\vec{y})$ since they offer no structure for interaction, they are not live.

Lemma 1. *If $P \vdash_\emptyset \Delta;\Gamma$ and $\Delta \neq \emptyset$, then P is live.*

Proof. By induction on a derivation of $P \vdash_\emptyset \Delta;\Gamma$. Case [T**0**] holds vacuously because it types inaction **0** with an empty linear context. Case [Tvar] holds vacuously because it types a variable with a nonempty recursion map η.

Cases which introduce the forwarder construct or an action hold trivially since P can be written as $-[\mathbf{fwd}\ x\ y]$ or $-[\mathcal{A}]$, where $-$ is the empty static process context and $\mathcal{A}$ is an action.

The remaining cases are [Tmix], [Tcut], [Tcut!], [Tsh], [TshL], [TshR] and [Tsum]. In these cases, from the fact that the conclusion types with a nonempty linear context we can infer that at least one of the premises types with a nonempty linear context as well, so that we can apply the inductive hypotheses to infer liveness of one of the arguments of P, which then implies liveness of P. We illustrate with cases [Tmix] and [Tsh].

Case [Tmix]

We have

$$\dfrac{\dfrac{\vdots}{P_1 \vdash \Delta_1;\Gamma} \quad \dfrac{\vdots}{P_2 \vdash \Delta_2;\Gamma}}{\mathbf{par}\ \{P_1 \mid\mid P_2\} \vdash \Delta_1,\Delta_2;\Gamma}\ [\text{Tmix}]$$

where $P = \mathbf{par}\ \{P_1 \mid\mid P_2\}$ and $\Delta = \Delta_1, \Delta_2$.

Since $\Delta \neq \emptyset$, then either $\Delta_1 \neq \emptyset$ or $\Delta_2 \neq \emptyset$.

Assume w.l.o.g. that $\Delta_1 \neq \emptyset$.

By applying the i.h. to $P_1 \vdash \Delta_1;\Gamma$ we conclude that $P_1 = C_1[X]$, where C is a static context and X is either an action or a forwarder.

Let $C = \mathbf{par}\ \{C_1 \mid\mid P_2\}$. Then, C is static and $P = C[X]$.

$$\frac{}{\textbf{fwd}\ x\ y \downarrow_{x:\text{fwd}}}\ [\text{fwd}] \quad \frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_{x:\text{act}}}\ [\text{act}]$$

$$\frac{P \downarrow_{x:\sigma}}{(\textbf{par}\ \{P \parallel Q\}) \downarrow_{x:\sigma}}\ [\text{mix}] \quad \frac{P \downarrow_{y:\sigma} \quad y \neq x}{(\textbf{cut}\ \{P\ \ |x|\ \ Q\}) \downarrow_{y:\sigma}}\ [\text{cut}] \quad \frac{Q \downarrow_{z:\sigma} \quad z \neq x}{(\textbf{cut!}\ \{y.P\ |x|\ Q\}) \downarrow_{z:\sigma}}\ [\text{cut!}]$$

$$\frac{P \downarrow_{y:\sigma} \quad y \neq x}{(\textbf{share}\ x\ \{P \parallel Q\}) \downarrow_{y:\sigma}}\ [\text{share}] \quad \frac{P \downarrow_{x:\sigma}}{(P + Q) \downarrow_{x:\sigma}}\ [\text{sum}] \quad \frac{P \equiv Q \quad Q \downarrow_{x:\sigma}}{P \downarrow_{x:\sigma}}\ [\equiv]$$

Figure 6.3: Observability predicate $P \downarrow_{x:\sigma}$, $\sigma \in \{\text{fwd}, \text{act}\}$

Case [Tsh].

We have

$$\frac{\begin{array}{cc}\vdots & \vdots \\ P_1 \vdash \Delta_1, x : \mathbf{U}_f\ A; \Gamma & P_2 \vdash \Delta_2, x : \mathbf{U}_f\ A; \Gamma\end{array}}{\textbf{share}\ x\ \{P_1 \parallel P_2\} \vdash \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma}\ [\text{Tsh}]$$

where $P = \textbf{share}\ x\ \{P_1 \parallel P_2\}$ and $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_f\ A$.

By applying the i.h. to $P_1 \vdash \Delta_1, x : \mathbf{U}_f\ A; \Gamma$ we conclude that $P_1 = C_1[\mathcal{Y}]$, where C is a static context and $\mathcal{Y}$ is either an action or a forwarder.

Let $C = \textbf{share}\ x\ \{C_1 \parallel P_2\}$. Then, C is static and $P = C[\mathcal{Y}]$.

Notice that in this case both premises type with a nonempty linear context, independently of the conclusion, and so the hypothesis that Δ is nonempty is superfluous. We could have opted to establish liveness of $\textbf{share}\ x\ \{P_1 \parallel P_2\}$ by applying the i.h. to $P_2 \vdash \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ instead. A similar situation happens for [Tcut].

□

The progress Theorem 2 states that a closed, i.e. typed with an empty typing context $P \vdash_\emptyset \emptyset; \emptyset$ and empty map η, and live process P reduces. If one tries to prove this statement by induction on a typing derivation for $P \vdash_\emptyset \emptyset; \emptyset$ one soon realises, when analysing the case [Tcut], that we need to say something about open processes. That is, to compositionally prove progress we need to characterise the potential interactions of (possibly open) typed processes, for which we define the following observability predicate, which is akin to π-calculus observability (cf. [137]). Our proof is along the lines of [23], but here we rely in an observability predicated, whereas in [23] progress is established by relying on a labelled transition system instead.

Definition 17 (Observability Predicate)**.** *The relation $P \downarrow_{x:\sigma}$, where $\sigma = fwd$ or $\sigma = act$, is defined by the rules of Figure 6.3. We say that x is an observable of P or that we can observe x in P, written $P \downarrow_x$, if either $P \downarrow_{x:fwd}$ or $P \downarrow_{x:act}$. If $P \downarrow_{x:act}$, we say that x is an observable action of P. If $P \downarrow_{x:fwd}$, we say that x is an observable forwarder of P.*

The definition of $P \downarrow_x$ is explicitly closed under $\equiv$ (rule [$\equiv$]) and propagates observations on the various static operators. For example, x is an observable of a mix **par** $\{P \mid\mid Q\}$, provided x is an observable of one of its arguments P or Q. The same principle applies to the cut construct with the proviso that we can never observe the name x in a cut **cut** $\{P \ |x| \ Q\}$ since it is kept private to the interacting processes P and Q.

We can always observe the subject of an action (rule [act]) and we can observe the constituent names x, y of a forwarder **fwd** $x\ y$: observation of x is direct from rule [fwd], whereas observation of y follows because of the $\equiv$ commuting rule [fwd] **fwd** $x\ y \equiv$ **fwd** $y\ x$

$$\dfrac{\textbf{fwd}\ x\ y \ \equiv \textbf{fwd}\ y\ x \qquad \dfrac{}{\textbf{fwd}\ y\ x \downarrow_y}\ [\text{fwd}]}{\textbf{fwd}\ x\ y \downarrow_x}\ [\equiv]$$

In a share **share** $x\ \{P \mid\mid Q\}$, processes P and Q run concurrently freely communicating with the external context and sharing memory cell x. As a consequence, and similar to the cut construct, the share construct **share** $x\ \{P \mid\mid Q\}$ propagates all the observations y for which $y \neq x$ (rule [share]).

Intuitively, x is an observable of a process P iff we can rewrite P in an $\equiv$-equivalent form Q so as to expose an action with subject x or forwarder **fwd** $x\ y$ and, furthermore, that action or forwarder in Q is not under the scope of a sharing construct on x. The following Definition 18 and Lemma 2 formalise this intuition

Definition 18. *We say that a hole in a static context C is guarded by a share on x iff there are static contexts C_1, C_2 and a process P s.t. $C = C_1[\textbf{share}\ x\ \{C_2 \mid\mid P\}]$ or $C = C_1[\textbf{share}\ x\ \{P \mid\mid C_2\}]$.*

Lemma 2. *Let P be a typed process. The following two propositions are equivalent*

(1) $P \downarrow_x$.

(2) $P \equiv C[\mathcal{A}]$ *or* $P \equiv C[\textbf{fwd}\ x\ y]$, *where* $s(\mathcal{A}) = x$, $x \in \mathsf{fn}(P)$, *for some static context C where the hole is not guarded by a share on x.*

Proof. **Proof of (1) $\Longrightarrow$ (2):** By induction on a derivation tree for $P \downarrow_x$ and case analysis on the root rule. We illustrate with some cases

Case: [fwd].
We have $P =$ **fwd** $x\ y$. Let C be the empty context $-$. A similar analysis for case [act].

Case: [share].
We have

$$\dfrac{\dfrac{\vdots}{P_1 \downarrow_x} \qquad y \neq x}{\textbf{share}\ y\ \{P_1 \mid\mid P_2\} \downarrow_x}\ [\text{share}]$$

where $P =$ **share** $y\ \{P_1 \mid\mid P_2\}$.

By applying the i.h. to $P_1 \downarrow_x$ we infer the existence of a static context C_1 for which (2) is true of P_1.

Let $C = \mathbf{share}\ y\ \{C_1 \mid\mid P_2\}$. Notice that the hole in C is still no guarded by a share on x since $y \neq x$.

Case: [≡].

We have

$$\frac{P \equiv P' \qquad \dfrac{\vdots}{P' \downarrow_x}}{P \downarrow_x}\ [\equiv]$$

Follows by applying i.h. to $P' \downarrow_x$ and by transitivity of ≡ (≡ rule [trans]).

Proof of (2) $\Longrightarrow$ (1): We prove by induction on the static context C that $C[\mathcal{X}] \downarrow_x$, where $\mathcal{X}$ stand for the action $\mathcal{A}$ or the forwarder $\mathbf{fwd}\ x\ y$. Then, by applying rule [≡] we conclude that $P \downarrow_x$. We illustrate with some cases

Case: $C = -$.

Follows by applying either [fwd] or [act].

Case: $C = \mathbf{cut}\ \{C'\ \ |y|\ \ Q\}$.

Applying i.h. to $C'[\mathcal{X}]$ yields $C'[\mathcal{X}] \downarrow_x$.

Since $x \in \mathsf{fn}(C[\mathcal{X}])$ (as $x \in \mathsf{fn}(P)$, $P \equiv C[\mathcal{X}]$ and ≡ preserves free names) we conclude that $y \neq x$.

Hence, we can apply rule [cut] to obtain $\mathbf{cut}\ \{C'[\mathcal{X}]\ \ |y|\ \ Q\} \downarrow_x$.

Case: $C = \mathbf{share}\ y\ \{C' \mid\mid Q\}$.

Applying i.h. to $C'[\mathcal{X}]$ yields $C'[\mathcal{X}] \downarrow_x$.

Since the hole in C is not guarded by a share on x, we conclude that $y \neq x$.

Hence, we can apply rule [share] to obtain $\mathbf{share}\ y\ \{C'[\mathcal{X}] \mid\mid Q\} \downarrow_x$.

□

We will now present some properties (Lemma 3) concerning the observability predicate, which will play a key role to derive progress.

Lemma 3 (Properties of $P \downarrow_x$)**.** *The following properties hold*

(1) *Let $P \vdash_\eta \Delta, x : \mathbf{U}_f\ A; \Gamma$ and $Q \vdash_\eta \Delta', x : \mathbf{U}_f\ A; \Gamma$ be processes for which $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, $\mathbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:act}$.*

(2) *Let $P \vdash_\eta \Delta, x : \mathbf{U}_e\ A; \Gamma, Q \vdash_\eta \Delta, x : \mathbf{U}_f\ A; \Gamma$. If $P \downarrow_{x:act}$, then $\mathbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:act}$.*

(3) *Let $P \vdash_\eta \Delta, x : \mathbf{U}_f\ A; \Gamma, Q \vdash_\eta \Delta, x : \mathbf{U}_e\ A; \Gamma$. If $Q \downarrow_{x:act}$, then $\mathbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:act}$.*

(4) *Let $P \vdash_\eta \Delta, x : \overline{A}; \Gamma$ and $Q \vdash_\eta \Delta', x : A; \Gamma$ be processes for which $P \downarrow_{x:act}$ and $Q \downarrow_{x:act}$. Then, $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ reduces.*

(5) *Let* $P \vdash_\eta \Delta, x : A; \Gamma$, $Q \vdash_\eta \Delta', x : A; \Gamma$ *be processes for which* $P \downarrow_{x:fwd}$. *Then,* **cut** $\{P \ |x| \ Q\}$ *reduces.*

(6) *Let* $P \vdash_\eta y : \overline{A}; \Gamma$ *and* $Q \vdash_\eta \Delta; \Gamma, x : A$ *be processes for which* $Q \downarrow_x$. *Then,* **cut!** $\{y.P \ |x| \ Q\}$ *reduces.*

(7) *Let* $P \vdash_\eta \Delta, x : A; \Gamma$ *and suppose that* $A \neq \mathbf{S}_f \ B$ *and* $A \neq \mathbf{S}_e \ B$. *If* $P \downarrow_{x:fwd}$, *then either (i)* $P \downarrow_{y:fwd}$ *for some* $y : \overline{A} \in \Delta$ *or (ii)* P *reduces.*

Proof. We give a proof sketch. Complete proofs for all properties ca be found in Appendix B.

(1) By double induction on derivation trees for $P \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$. For the base cases we apply either one of ≡ rules [RSh] or [TSh] in order to expose an observable action. For the inductive cases we consider that we are given a derivation tree for $P \downarrow_x$. This is w.l.o.g. since **share** $x \ \{P \ || \ Q\} \equiv$ **share** $x \ \{Q \ || \ P\}$. In cases [mix], [cut], [cut!], [share] we commute the share on x with the principal form of P by applying either ≡ rule [ShM], [CSh], [ShC!], [ShSh] or [ShSm]. The inductive case [≡] follows immediately because the relation ≡ is a congruence, i.e. satisfies ≡ rule [cong].

(2) By induction on the structure of a derivation for $P \downarrow_{x:act}$ and case analysis on the root rule. The base case [act] follows by applying ≡ rule [PSh] in order to expose the put action. The inductive cases [mix], [cut], [cut!], [share], [sum] and [≡] are handled in the same way as (1).

(3) Follows by (2) and because share is commutative: **share** $x \ \{P \ || \ Q\} \equiv$ **share** $x \ \{Q \ || \ P\}$.

(4) By double induction on derivation trees for $P \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$. For the base cases we apply one of the principal cut conversions. For the inductive cases we consider that we are given a derivation tree for $P \downarrow_x$. This is w.l.o.g. since **cut** $\{P \ |x| \ Q\} \equiv$ **cut** $\{Q \ |x| \ P\}$. For cases [mix], [cut], [cut!], [share] we commute the cut on x with the principal form of P by applying either ≡ rule [CM], [CC], [CC!], [CSh] or [CSm]. The inductive case $P \downarrow_x$ rule [≡] follows immediately because the relation $\longrightarrow$ is closed by structural congruence.

(5) By induction on a derivation trees for $P \downarrow_{x:\text{fwd}}$. The base case [fwd] follows by applying reduction rule $\longrightarrow$ [fwd]. The inductive cases [mix], [cut], [cut!], [share], [sum] and [≡] are handled in the same way as (4).

(6) By induction on a derivation tree for $Q \downarrow_x$ and case analysis on the root rule. The base case [act] follows by applying reduction rule $\longrightarrow$ [call]. The inductive cases [mix], [cut], [cut!], [share] and [sum] follow by distributing the unrestricted cut over the arguments of Q (with ≡ rules [D-C!M], [D-C!C], [D-C!C!], [D-C!Sh] or [C!Sm]) and then apply the inductive hypothesis. The inductive case [≡] follows because reduction $\longrightarrow$ is closed by structural congruence.

(7) By induction on a derivation tree $P \downarrow_{x:\mathrm{fwd}}$. In the inductive cases we propagate either the observable action on name y or the reduction. In the action on name y occurs with a cut $\mathbf{cut}\ \{-\ |y|\ -\}$ we trigger a reduction by applying (5). The typing constraints $A \neq \mathbf{S}_f\ B$ and $A \neq \mathbf{S}_e\ B$ ensures that we can propagate the observable action on y in case of a share.

□

Properties Lemma 3(1)-(3) describe sufficient conditions to propagate observations x on a share $\mathbf{share}\ x\ \{P \parallel Q\}$.

Lemma 3(1) states that we can observe a full usage on x in a $\mathbf{share}\ x\ \{P \parallel Q\}$ provided we can observe a full usage x on both P and Q. This full usage on x is propagated by applying either ≡ rule [RSh] or ≡ rule [TSh]. For example, by rule ≡ [RSh] we have $\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel \mathbf{take}\ x(y); P\} \equiv \mathbf{take}\ x(y); P$. Then

$$\dfrac{\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel \mathbf{take}\ x(y); P\} \equiv \mathbf{take}\ x(y); P \qquad \dfrac{s(\mathbf{take}\ x(y); P) = x}{\mathbf{take}\ x(y); P \downarrow_x}\ [\mathrm{act}]}{\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel \mathbf{take}\ x(y); P\} \downarrow_x}\ [\equiv]$$

Additionally, we can observe an empty usage x on $\mathbf{share}\ x\ \{P \parallel Q\}$ provided we can observe an empty usage x on either P or Q, as stated by Lemma 3(2)-(3). The empty usage corresponds to a put action which can always be propagated to the top by applying ≡ rule [PSh].

Properties Lemma 3(4)-(6) describe sufficient conditions for obtaining a reduction: either by observing two dual actions with subject x in a linear cut $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ (Lemma 3(4)), by observing a forwarder x on a linear cut $\mathbf{cut}\ \{P\ |x|\ Q\}$ (Lemma 3(5)) or by observing a single action x in the right argument Q of an unrestricted cut $\mathbf{cut!}\ \{y.P\ |x|\ Q\}$ (Lemma 3(6)).

Lemma 3(7) characterises the potential observation or reduction of a process that P for which $P \downarrow_{x:\mathrm{fwd}}$. Either name y occurs free, and P also offers a forwarder interaction at y, or lies in the scope of a cut $\mathbf{cut}\ \{-\ |y|\ -\}$, in which case a reduction can be triggered (Lemma 3(5)). The typing constraints $A \neq \mathbf{S}_f\ B$ and $A \neq \mathbf{S}_e\ B$ exclude processes like $\mathbf{share}\ y\ \{\mathbf{fwd}\ x\ y \parallel Q\}$, that neither reduce nor offer an interaction at y. Intuitively, in this case, the share is suspended on the availability of cell usages at name y. We will see later, when studying cut normalisation, in Chapter 8, that this kind of processes will play a key role in defining normal forms.

We now state our liveness Lemma 4 which says that a live open process either reduces or offers an interaction at some session x. This lemma implies our main progress result (Theorem 2), with which we conclude this section.

Lemma 4 (Liveness)**.** *Let $P \vdash_\emptyset \Delta; \Gamma$ be such that P is live. Either $P \downarrow_x$, for some x, or P reduces.*

Proof. By induction on a typing derivation for $P \vdash_\emptyset \Delta; \Gamma$. We illustrate with some cases. The complete proof can be found in Appendix B.

Case [T0]

We have

$$\frac{}{\mathbf{0} \vdash \emptyset; \Gamma} \text{ [T0]}$$

where $P = \mathbf{0}$. Holds vacuously because $\mathbf{0}$ is not live.

Case [Tvar]

We have $\dfrac{\eta = \eta', X(\vec{y}) \mapsto \Delta'; \Gamma'}{X(\vec{x}) \vdash_\eta \{\vec{x}/\vec{y}\}(\Delta'; \Gamma')}$ [Tvar] where $P = X(\vec{x})$. Holds vacuously because assumes a nonempty η context.

Case [T1]

Since $P = \mathbf{close}\ x$ is an action:

$$\frac{s(\mathbf{close}\ x) = x}{\mathbf{close}\ x \downarrow_x} \text{ [act]}$$

Similarly for [Tfwd] and the other typing rules which introduce an action.

Case [Tcut]

We have

$$\frac{P_1 \vdash_\emptyset \Delta_1, x : \overline{A}; \Gamma \quad P_2 \vdash_\emptyset \Delta_2, x : A; \Gamma}{\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \vdash_\emptyset \Delta_1, \Delta_2; \Gamma} \text{ [cut]}$$

where $P = \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ and $\Delta = \Delta_1, \Delta_2$.

Since both P_1 and P_2 type with nonempty linear typing context and with an empty map η, we conclude that both P_1 and P_2 are live (Lemma 1).

By applying the i.h. to $P_1 \vdash_\emptyset \Delta_1, x : \overline{A}; \Gamma$ and $P_2 \vdash_\emptyset \Delta_2, x : A; \Gamma$ we conclude that

- $P_1 \downarrow_y$ or P_1 reduces, and
- $P_2 \downarrow_z$ or P_2 reduces

We have the following cases to consider

Case ($P_1 \downarrow_y$ and $y \neq x$) or ($P_2 \downarrow_z$ and $z \neq x$)

Suppose w.l.o.g. that $P_1 \downarrow_y$ and $y \neq x$.

Then

$$\frac{P_1 \downarrow_y \quad y \neq x}{(\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}) \downarrow_y} \text{ [cut]}$$

Case $P_1 \downarrow_x$ and $P_2 \downarrow_x$

We have the following two cases

Case $P_1 \downarrow_{x:\text{fwd}}$ or $P_2 \downarrow_{x:\text{fwd}}$

Suppose w.l.o.g. that $P_1 \downarrow_{x:\text{fwd}}$.

Then, by Lemma 3(5), we conclude that $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ reduces.

Case $P_1 \downarrow_{x:\text{act}}$ and $P_2 \downarrow_{x:\text{act}}$

Then, by Lemma 3(4), we conclude that $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ reduces.

Case P_1 reduces or P_2 reduces

Because of $\longrightarrow$ rule [cong], $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ reduces.

Case [Tsh]

We have

$$\frac{P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_f\ A; \Gamma \quad P_2 \vdash_\emptyset \Delta_2, x : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ x\ \{P_1 \mid\mid P_2\} \vdash \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma}\ [\text{Tsh}]$$

where $P = \mathbf{share}\ x\ \{P_1 \mid\mid P_2\}$ and $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_f\ A$.

Since both P_1 and P_2 type with nonempty linear context and an empty map η, then both P_1 and P_2 are live (Lemma 1).

By applying the i.h. to $P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash_\emptyset \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ we conclude that

- $P_1 \downarrow_y$ or P_1 reduces, and
- $P_2 \downarrow_z$ or P_2 reduces.

We have the following cases to consider.

Case A ($P_1 \downarrow_y$ and $y \neq x$) or ($P_2 \downarrow_z$ and $z \neq x$)

Suppose w.l.o.g. that $P_1 \downarrow_y$ and $y \neq x$.

Then

$$\frac{P_1 \downarrow_y \quad y \neq x}{(\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}) \downarrow_y}\ [\text{share}]$$

Case B $P_1 \downarrow_x$ and $P_2 \downarrow_x$

We have the following two cases.

Case B1 $P_1 \downarrow_{x:\text{fwd}}$ or $P_2 \downarrow_{x:\text{fwd}}$

Suppose w.l.o.g. that $P_1 \downarrow_{x:\text{fwd}}$.

Observe that x occurs typed by $\mathbf{U}_f\ A$ in the linear typing context of P_1. Hence, we can apply Lemma 3(7) in order to conclude that either (i) $P_1 \downarrow_y$ for $y \neq x$ or (ii) P_1 reduces. If (i) go to case A. If (ii), go to case C.

Case B2 $P_1 \downarrow_{x:\text{act}}$ and $P_2 \downarrow_{x:\text{act}}$.

Then $(\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}) \downarrow_x$ (Lemma 3(1)).

Case C P_1 reduces or P_2 reduces

Because of $\longrightarrow$ rule [cong], $\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}$ reduces.

□

Theorem 2 (Progress). *Let $P \vdash_\emptyset \emptyset;\emptyset$ be a live process. Then, P reduces.*

Proof. Follows from Lemma 4 since $\mathsf{fn}(P) = \emptyset$. □

6.4 Further Discussion and Related Work

Type Preservation

In the seminal propositions-as-types correspondence [23, 24] two languages are connected: a typed synchronous π-calculus extended with guarded choice and a faithful proof term assignment for dual intuitionistic linear logic. The connection is established by exhibiting a simulation between reductions in the typed π-calculus and conversions in the proof term calculus, therefore establishing a strong form of subject reduction for the typed π-calculus.

In [23, 24], the underlying (untyped) π-calculus of the session-typed system has structural congruence and reduction rules which do not have a direct correspondence with typing derivation / proof conversion manipulations, like for example the distribution of name restriction over parallel composition $x \notin \mathsf{fn}(P) \supset P \,|(\nu x)Q \equiv (\nu x)\,(P \mid Q)$.

In our approach there is a single language which corresponds to the proof term annotation of typing derivations / proofs in CLASS. Therefore, all process manipulations in CLASS can be also understood as manipulations on typing derivations / proofs, as exhibited by our proof of type preservation (Theorem 1).

Progress

Deadlock-freedom in the context of session-based concurrent programming languages results from restricting communication topologies to be acyclic. There are two main ways in which type systems guarantee acyclicity, namely either by (a) following a logical approach based on propositions-as-types or (b) by defining extra-logical mechanisms, such as partial orders.

Approach (a) is characteristic of session-typed systems which build on the propositions-as-types correspondence to linear logic [23, 157, 91, 128], where two concurrent threads can only be connected via a binary cut. Approach (b) essentially builds on the work of Kobayashi [88, 87], and for example, the works [25, 154, 43, 49, 163] also fall in this category.

Our approach to obtain deadlock-freedom for CLASS falls in the first category (a). Binary cut [Tcut] guarantees that two concurrent threads can synchronise on only a single private session, which excludes blocked processes of the form

$$\mathbf{cut}\ \{\mathbf{send}\ x(-.-);\mathbf{send}\ y(-.-);-\ \ |x,y|\ \ \mathbf{recv}\ y(-);\mathbf{recv}\ x(-);-\} \quad \text{(I)}$$

from being typed, where the $-$ is a placeholder for a process or session name. But it also excludes non-blocked processes of the form

$$\textbf{cut}\ \{\textbf{send}\ x(-.-); \textbf{send}\ y(-.-); -\ \ |x, y|\ \ \textbf{recv}\ x(-); \textbf{recv}\ y(-); -\} \quad \text{(II)}$$

This interaction through a single session property is then transposed to the stateful extension of the language by restricting two concurrent threads to share at most one cell usage, as typed by [Tsh], [TshL] and [TshR], which logically interprets cocontraction in DiLL [53]. This automatically excludes processes of the form

$$\begin{array}{l}\textbf{cut}\ \{\textbf{cell}\ y(-.-)\ \ |y|\ \ (\textbf{cut}\ \{\textbf{cell}\ x(-.-)\ \ |x| \\ \textbf{share}\ x, y\ \{\textbf{take}\ x(-); \textbf{take}\ y(-); - \,||\, \textbf{take}\ y(-); \textbf{take}\ x(-); -\}\})\}\end{array} \quad \text{(III)}$$

which block when the left thread takes x and is waiting for taking y, whereas the right thread takes y and is waiting forever for usage x to be put back. But it also excludes non-blocking programs of the form (IV)

$$\begin{array}{l}\textbf{cut}\ \{\textbf{cell}\ y(-.-)\ \ |y|\ \ (\textbf{cut}\ \{\textbf{cell}\ x(-.-)\ \ |x| \\ \textbf{share}\ x, y\ \{\textbf{take}\ x(-); \textbf{take}\ y(-); - \,||\, \textbf{take}\ x(-); \textbf{take}\ y(-); -\}\})\}\end{array} \quad \text{(IV)}$$

While approach (b) to deadlock-freedom is less conservative, approach (a) is arguably simpler and integrates well with further extensions such as polymorphism [26], dependent types [145], control effects [28] and even recursion [146, 99]. Furthermore, the behaviour of programs like (IV) can be reproduced in CLASS by packing all shared state between threads in a single data structure so as to force resources to be acquired in a defined order, as illustrated by Example 4.3.

Partial orders do not compose well [82] and are a leaky abstraction. To illustrate this last point, consider that one needs to write a program that returns the total amount of money in a given list of stateful bank accounts. A simple program in CLASS would be expressed as the composition of three steps: (i) first acquire the lock of each bank account, then (ii) compute the sum of the money stored in all the bank accounts and finally (iii) release the lock of each bank account. Actually, this example is coded in *state/bank-account*. In an approach with partial orders, to program step (i) we would need first to order the inputted list according to the partial order. Additionally, it is well-recognised that the integration of recursion and statically unbounded structures with the partial-order based type systems of approach (b) poses some technical challenges [60, 89, 11]. See [44] for a detailed comparison between the expressiveness of the two approaches.

The type system $\mathsf{SILL}_{\mathsf{S}^+}$ of manifest sharing [11] obtains deadlock-freedom by employing a mixed strategy between approaches (a) and (b): threads can synchronise their actions on only one single channel but can share more than one stateful resource. The communication topology then forms a tree and there is partial order on top of the typing system which forces shared resources to be acquired in ascending order, i.e. to *lock-up*.

The type system $\mathsf{SILL}_{\mathsf{S}^+}$ excludes circular dependencies resulting from lock acquisitions and also between lock acquisitions and other synchronisation actions like session

input-output communication. The partial orders of $\mathsf{SILL}_{\mathsf{S}^+}$ comes with some restrictions however, for example: in $\mathsf{SILL}_{\mathsf{S}^+}$ every process has to release all its shared resources before synchronising with other process, whereas in CLASS this is not the case as, for example, we can type process **wait** y; **put** $x(-.-)$; **release** x with the linear context $x : \mathbf{U}_e\ A, y : \bot$, that synchronises on y even though it has an acquired session x. Furthermore, we can communicate empty usages (acquired sessions) on session channels, after which the processes who sends the empty usage looses access to it.

Additionally, in order to preserve the invariants that guarantee deadlock-freedom, the system in $\mathsf{SILL}_{\mathsf{S}^+}$ does not type the linear forwarder, therefore it does not provide a computational interpretation for the identity axiom of linear logic, excluding some important session-based encodings, such as the free output, from being expressed. Also, the type system $\mathsf{SILL}_{\mathsf{S}^+}$ might not scale for complex programs since the partial order on locks have to be manually defined by the programmer. Since $\mathsf{SILL}_{\mathsf{S}^+}$ does not support systems in which the partial orders are dynamically generated, it cannot type processes that spawn a statically undetermined number of shared objects, while in our work such systems are naturally allowed (see Example 14).

7

Confluence

7.1 Introduction

In this chapter we prove that the reduction relation $\xrightarrow{*}$ of language CLASS satisfies the diamond property (modulo $\equiv$) or, equivalently, that the process reduction relation $\rightarrow$ is globally confluent (satisfies the Church-Rosser Property).

Confluence is a property that holds of various functional calculi, e.g. the untyped lambda calculus [37], but that normally fails for concurrent process calculi, e.g. the π-calculus [106], where there are races on the communication channels.

As already remarked, the basic language μCLL, corresponding to the session-based interpretation of classic linear logic, is essentially functional [149] and, therefore, confluence for this language comes at no surprise: the typing system excludes races on the linear sessions and the unrestricted sessions are served by uniform replicated processes. However, as soon as we add reference cells and cell sharing to obtain CLASS, races naturally emerge from concurrent manipulations of stateful objects, as in two concurrent take operations to the same reference cell.

Nevertheless, reduction $\rightarrow$ in CLASS still enjoys the global confluence property. This is the case because the nondeterminism that naturally emerges from concurrent cell manipulations is captured by sums, via the interleaving structural congruence $\equiv$ law [TSh]

$$\begin{aligned}&\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel \mathbf{take}\ x(y_2); P_2\}\\ &\equiv \mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \parallel \mathbf{take}\ x(y_2); P_2\} + \mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel P_2\}\end{aligned}$$

and, furthermore, sums in CLASS are non-collapsing, i.e. they satisfy *neither* the law $P + Q \rightarrow P$ *nor* the law $P + Q \rightarrow Q$.

By establishing confluence we guarantee that the order in which processes are reduced in CLASS is irrelevant and we also give substance to the claim that $\xrightarrow{*}$ can be understood as a proof equivalence relation, as required by propositions-as-types. Confluence is one of the key steps towards equational reasoning, the other being strong normalisation (established in Chapter 9), which together imply the existence of unique process / proof normal forms that summarise the behaviour of each process term.

We will now describe our general strategy to establish global confluence for reduction $\rightarrow$ in CLASS (i.e., diamond property for $\overset{*}{\rightarrow}$). Details about the proof will be described in the following sections and in Appendix C. We employ the Tait and Martin-Löf technique, which consists in establishing the diamond property for a relation $\mathcal{R}$ by finding a relation $\mathcal{S}$ s.t. (1) $\mathcal{S}$ satisfies the diamond property and (2) $\mathcal{S}^* = \mathcal{R}$, i.e. the reflexive-transitive closure of $\mathcal{S}$ is the same as a $\mathcal{R}$.

Since $\overset{*}{\rightarrow}$ is the reflexive-transitive closure of reduction $\rightarrow$, a natural proof strategy would be demonstrating the diamond property for $\rightarrow$, but this property fails. To see why, consider the process

$$\mathbf{par}\ \{P \parallel (Q_1 + Q_2 + Q_3)\}$$

and assume that $P' \leftarrow P \rightarrow P''$. Then, we can form the $\rightarrow$-reductions

$$\begin{array}{lcll} \mathbf{par}\ \{P \parallel (Q_1 + Q_2 + Q_3)\} & \rightarrow & \mathbf{par}\ \{P' \parallel Q_1\} + \mathbf{par}\ \{P' \parallel Q_2\} + \mathbf{par}\ \{P \parallel Q_3\} & \triangleq\ R \quad (1) \\ \mathbf{par}\ \{P \parallel (Q_1 + Q_2 + Q_3)\} & \rightarrow & \mathbf{par}\ \{P \parallel Q_1\} + \mathbf{par}\ \{P'' \parallel Q_2\} + \mathbf{par}\ \{P'' \parallel Q_3\} & \triangleq\ S \quad (2) \end{array}$$

Reduction (1) is obtained by associating the sum as $(Q_1+Q_2)+Q_3$, distributing the mix over the outer sum, applying the reduction $P \rightarrow P'$ to obtain $\mathbf{par}\ \{P' \parallel (Q_1+Q_2)\}+\mathbf{par}\ \{P' \parallel Q_3\}$, after which we distribute P' over the sum $Q_1 + Q_2$. The derivation of reduction (2) follows a similar pattern but we associate the sum as $Q_1 + (Q_2 + Q_3)$ and we apply the reduction $P \rightarrow P''$ instead.

Assume that P' and P'' reduce to a common form P^* in one single step: $P' \rightarrow P^* \leftarrow P''$. Note, however, that we need two $\rightarrow$-reduction steps to bring processes R and S to a common term

$$R \quad \overset{2}{\rightarrow} \quad \mathbf{par}\ \{P' \parallel Q_1\} + \mathbf{par}\ \{P^* \parallel Q_2\} + \mathbf{par}\ \{P'' \parallel Q_3\} \quad \overset{2}{\leftarrow} \quad S$$

The solution is to allow, in one single step, to perform in parallel $\rightarrow$-reductions of independent sum components, as captured in the following definition.

Definition 19 (Parallel Sum Reduction $\twoheadrightarrow$)**.** *Let $\twoheadrightarrow$ be the least relation that contains $\rightarrow$ and satisfies the following rule*

$$\frac{P \twoheadrightarrow P' \quad Q \twoheadrightarrow Q'}{P + Q \twoheadrightarrow P' + Q'}\ [+par]$$

The relation $\twoheadrightarrow$ allows us to $\rightarrow$-reduce the arguments $P_1, \ldots, P_n$ of a sum expression $P_1 + \ldots + P_n$ in parallel. That is, if $P_i \rightarrow P_i'$, for all $1 \leq i \leq n$, then

$$P_1 + \ldots + P_n \twoheadrightarrow P_1' + \ldots + P_n'$$

Therefore, it allows us to close the $\rightarrow$-reduction fork (1)-(2) in the example above in one single $\twoheadrightarrow$-reduction step. The following lemma relates $\rightarrow$ with $\twoheadrightarrow$.

Lemma 5. *The following (in)equalities hold: (1) $\rightarrow \subseteq \twoheadrightarrow$, (2) $\twoheadrightarrow \subseteq \overset{*}{\rightarrow}$ and (3) $\overset{*}{\twoheadrightarrow} = \overset{*}{\rightarrow}$.*

Proof. (1) follows directly from Definition 19. (2) is by induction on $\twoheadrightarrow$. (1) and (2) imply (3). □

Therefore, going back to the Tait and Martin-Löf technique, we observe that relation $\twoheadrightarrow$ satisfies requirement (2), i.e. that $\overset{*}{\rightarrow} = \overset{*}{\twoheadrightarrow}$, as stated by Lemma 5(3). The rest of the chapter is devoted to prove requirement (1), i.e. to establish the diamond property for relation $\twoheadrightarrow$.

To handle the complexity of the $\equiv$-commuting conversions that manipulate sums, we show first that we can write each process in a $\equiv$-normal form by interleaving all the concurrent take cell usages and distributing the static operators over sums. This essentially computes a normal form for the left-to-right oriented $\equiv$-rules that distribute the static constructors over sum ($\equiv$ rules [MSm], [CSm], [C!Sm] and [ShSm]) as well as for the rule that interleave take usages ($\equiv$ rule [TSh]).

Each $\twoheadrightarrow$-reduction is then obtained by evaluating the summands of this normal form with a restricted form of reduction $\rightarrow_d$ that does not manipulate sums at all and for which the diamond property is straightforward to establish. The relation $\twoheadrightarrow$ then factors $\rightarrow_d$, which allows us to lift the diamond property of the latter to the former, thereby concluding our proof of global confluence.

This chapter is organised as follows. Section 7.2 introduces the relation $\rightarrow_d$ and proves that it satisfies the diamond property (Lemma 8). Section 7.3 proves that relation $\twoheadrightarrow$ factors through $\rightarrow_d$ (Lemma 9(3)). Then, in Section 7.4, we lift the diamond property of $\rightarrow_d$ to $\twoheadrightarrow$ (Lemma 11) and, as consequence, for $\overset{*}{\rightarrow}$ which yields our main result (Theorem 3). Finally, Section 7.5 concludes with further discussion and related work.

7.2 The Reduction Relation $\rightarrow_d$

In this section we introduce the restricted simpler form of reduction $\rightarrow_d$ that essentially does not manipulate sums (Def. 20) and for which a diamond property is straightforward to establish (Lemma 8). We start by defining $\rightarrow_d$.

Definition 20 (Relations $\equiv_d$, $\rightarrow_d$)**.** *Let $\equiv_d$ be defined by all the rules of $\equiv$ except the laws that manipulate sums, namely the distributive laws over sum (rules [MSm], [CSm], [C!Sm] and [ShSm]) and the interleaving law (rule [TSh]), sum commutativity (rule [Sm]), associativity (rule [SmSm]) and idempotency for inaction (rule [0Sm]). Let $\rightarrow_d$ be the least relation that satisfies all the rules of $\rightarrow$ except [$\equiv$] and for which the rule [$\equiv_d$] $P \equiv_d P' \rightarrow_d Q' \equiv_d Q \supset P \rightarrow_d Q$ holds. Furthermore, in rule [cong], the hole in the context C is not guarded by a sum.*

Relation $\equiv_d$ is defined by all the process manipulation laws of $\equiv$, except those that involve sums. Reduction $\rightarrow_d$ contains all the principal cut reductions of $\rightarrow$ but is closed by congruence $\equiv_d$ instead of being closed by congruence $\equiv$. Furthermore, sum $P + Q$ is a $\rightarrow_d$-normal form since it offers no $\rightarrow_d$-reduction.

We will now prove that $\rightarrow_d$ satisfies the diamond property. The proof is as follows: we show that each process P can be decomposed in an $\equiv_d$-equivalent form as $C[R_1, \ldots, R_k]$, i.e., as a static context with k-holes that exhibits all the k predexes of the process P (Lemma 6). All the $\rightarrow_d$ reductions of process P are then obtained by $\rightarrow_d$-reducing one

of the predexes $R_1, \ldots, R_k$ and since the diamond property for $\rightarrow_d$ holds for a predex (Lemma 7), then the diamond property for $\rightarrow_d$ holds for all processes in general (Lemma 8). We start by formally defining what is a predex.

Definition 21 (Predex)**.** *The set of predexes is defined by*

$$\begin{array}{rcl} R & ::= & \mathbf{cut}\ \{\mathcal{A}\ \ |x|\ \ \mathcal{B}\},\ s(\mathcal{A}) = s(\mathcal{B}) = x \\ & | & \mathbf{cut!}\ \{y.P\ |x|\ \mathbf{call}\ x(y); Q\} \\ & | & \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |y|\ \ \mathcal{D}[R_1, \ldots, R_n]\} \\ & | & \mathbf{cut}\ \{C[R_1, \ldots, R_m]\ \ |x|\ \ \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |y|\ \ \mathcal{D}[R'_1, \ldots, R'_n]\}\} \end{array}$$

where $C, \mathcal{D}$ are static contexts with m ad n holes, respectively. Furthermore, the constituent processes of $C, \mathcal{D}$ have no possible $\rightarrow_d$ reduction, we write $C \nrightarrow_d$ and $\mathcal{D} \nrightarrow_d$.

Intuitively, predexes characterise the possible ways in which we can have a $\rightarrow_d$ reduction, namely it can be result of the interaction of two dual actions $\mathcal{A}$ and $\mathcal{B}$, of the interaction between an unrestricted server and a server invocation or it can be result of an interaction with a forwarder. In the latter case there are two possibilities, depending on whether the two forwarder names are cut or not.

The following result proves that the set of predexes is complete in the sense that we can decompose any process P as a context C and k predexes, the predexes exhibiting all the possible $\rightarrow_d$ reductions of P.

Lemma 6 (Predex Decomposition)**.** *For each process $P \vdash_\eta \Delta; \Gamma$ there are predexes $R_1, \ldots, R_k$ and a static context C such that $P \equiv_d C[R_1, \ldots, R_k]$ and $C \nrightarrow$.*

Proof. By induction on the structure of a typing derivation tree for $P \vdash_\eta \Delta; \Gamma$ and by case analysis on the root rule of the tree. We illustrate with cases [Tmix] and [Tcut]. The complete proof is in Appendix C.

Case: [Tmix]

Let $P = \mathbf{par}\ \{P_1 \parallel P_2\}$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

Let $C = \mathbf{par}\ \{C_1 \parallel C_2\}$.

We have $P \equiv_d C[R_1, \ldots R_p, S_1, \ldots, S_q]$, with $k = p + q$.

Furthermore, since $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$, then $C \nrightarrow_d$.

Case: [Tcut]

Let $P = \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

We have the following cases to consider.

Case: Either $C_1 = C_1'[\mathbf{fwd}\ x\ z]$ for some $1 \leq i \leq p$ and static context C_1' or $C_2 = C_2'[\mathbf{fwd}\ x\ z]$ for some $1 \leq j \leq q$ and static context C_2'.
Suppose w.l.o.g. that $C_1 = C_1'[\mathbf{fwd}\ x\ z]$ for some $1 \leq i \leq p$ and static context C_1'
We have two cases to consider, depending on whether (i) $\mathbf{fwd}\ x\ z$ is guarded by either a share or (ii) not.
If (i), then cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex. So simply let $C = \mathbf{cut}\ \{C_1\ \ |x|\ \ C_2\}$.
Suppose (ii). Then

$$\begin{aligned} P &= \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \mathbf{cut}\ \{C_1'[R_1, \ldots, R_p, \mathbf{fwd}\ x\ z]\ \ |x|\ \ C_2[S_1, \ldots, S_q]\} \\ &\equiv_d C_1'[R_1, \ldots, R_p, \mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}] \end{aligned}$$

with $k = p$.
Observe that $\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}$ is a predex.

Case: Either $R_i = \mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ |z|\ \mathcal{D}[M_1, \ldots, M_n]\}$ or $Q_j = \mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ |z|\ \mathcal{D}[M_1, \ldots, M_n]\}$ for some $1 \leq i \leq p, 1 \leq j \leq q$.

Suppose w.l.o.g. that $R_i = \mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}$.
We have two cases to consider, depending on whether(i) $\mathbf{fwd}\ x\ z$ is guarded by either a share in C_1 or (ii) not.
If (i), then cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex. Let simply $C = \mathbf{cut}\ \{C_1\ \ |x|\ \ C_2\}$.
Suppose (ii). Then

$$\begin{aligned} P &= \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \mathbf{cut}\ \{C_1[R_1, \ldots, \mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}, \ldots, R_p]\ \ |x|\ \ C_2[S_1, \ldots, S_q]\} \\ &\equiv_D C_1[R_1, \ldots, \mathbf{cut}\ \{\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}, \ldots, R_p] \end{aligned}$$

with $k = p$.
Observe that $\mathbf{cut}\ \{\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}$ is a predex.

Case: There is no C_1' s.t. $C_1 \equiv_d C_1'[\mathcal{A}]$, for some static context where $s(\mathcal{A}) = x$ and the hole in C_1' is not guarded by a share on x.
Then, the cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex.

Case: There is no C_2' s.t. $C_2 \equiv_d C_2'[\mathcal{A}]$, for some static context where $s(\mathcal{A}) = x$ and the hole in C_2' is not guarded by a share on x.
Then, the cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex.

Case: $C_1 \equiv_d C_1'[\mathcal{A}]$, $C_2 \equiv_d C_2'[\mathcal{B}]$, $s(\mathcal{A}) = s(\mathcal{B}) = x$, C_1', C_2' are static contexts in which the holes are not guarded by share operations on x.
Then

$$\begin{aligned} P &= \textbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \textbf{cut}\ \{C_1'[\mathcal{A}, R_1, \ldots, R_p]\ \ |x|\ \ C_2'[\mathcal{B}, S_1, \ldots, S_q]\} \\ &\equiv_d C_1'[\textbf{cut}\ \{\mathcal{A}\ \ |x|\ \ C_2'[\mathcal{B}, S_1, \ldots, S_q]\}, R_1, \ldots, R_p] \\ &\equiv_d C_1'[C_2'[\textbf{cut}\ \{\mathcal{A}\ \ |x|\ \ \mathcal{B}\}, S_1, \ldots S_q], R_1, \ldots, R_p] \end{aligned}$$

□

Lemma 7 (Diamond Property for $\rightarrow_d$, Restricted to Predexes)**.** *Let P be a predex s.t. $P \rightarrow_d Q$ and $P \rightarrow_d R$. Either $Q \equiv_d R$ or there exists S s.t. both $Q \rightarrow_d S$ and $R \rightarrow_d S$.*

Proof. The proof is by induction on the structure of the predex P. We illustrate for case in which the predex P is of the form $\textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_n]\}$. Complete proof is Appendix C.

Since $C \twoheadrightarrow_d$, the $\rightarrow_d$-reductions of $\textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_n]\}$ are either obtained by $\rightarrow_d$-reducing one of the predexes R_i or by $\rightarrow_d$-reducing the cut on name y with the forwarder $\textbf{fwd}\ x\ y$. Consequently, we have the following cases to consider

Case: Q and R are obtained by $\rightarrow_d$-reducing distinct predexes R_i, R_j.
Suppose w.l.o.g. that $i < j$. We have

$$\begin{aligned} Q &\equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i', \ldots, R_j, \ldots, R_n]\} \text{ and} \\ R &\equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i, \ldots, R_j', \ldots, R_n]\} \end{aligned}$$

where $R_i \rightarrow_d R_i'$ and $R_j \rightarrow_d R_j'$.

Let $S = \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i', \ldots, R_j', \ldots, R_n]\}$.

Then, both $Q \rightarrow_d S$ and $R \rightarrow_d S$.

Case: Q and R are obtained by $\rightarrow_d$-reducing the same predex R_i.
We have

$$Q \equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i', \ldots, R_n]\}$$

and

$$Q \equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i'', \ldots, R_n]\}$$

where $R_i \rightarrow_d R_i'$ and $R_i \rightarrow_d R_i''$.

By applying the i.h. to R_i we conclude that either (1) $R_i \equiv_d R_i''$ or (2) exists S_i s.t. both $R_i' \rightarrow_d S_i$ and $R_i'' \rightarrow_d S_i$.

Suppose (1). Then $Q \equiv_d R$.

Suppose (2). Let $S = \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, S_i, \ldots, R_n]\}$. Then, both $Q \rightarrow_d S$ and $R \rightarrow_d S$.

Case: One of Q, R is obtained by $\rightarrow_d$-reducing the cut on name y, the other by $\rightarrow_d$-reducing one predex R_i.

Suppose w.l.o.g. that Q is obtained by $\rightarrow_d$-reducing the cut on name y and that R is obtained by $\rightarrow_d$-reducing a predex R_i. Then

$$Q \equiv_d (\{x/y\}C)[\{x/y\}R_1, \ldots, \{x/y\}R_i, \ldots, \{x/y\}R_n] \text{ and}$$
$$R \equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i', \ldots, R_n]\}$$

where $R_i \rightarrow_d R_i'$.

We have $\{x/y\}R_i \rightarrow_d \{x/y\}R_i'$.

Let $S = (\{x/y\}C)[\{x/y\}R_1, \ldots, \{x/y\}R_i', \ldots, \{x/y\}R_n]$. Both $Q \rightarrow_d S$ and $R \rightarrow_d S$.

□

Lemma 8 (Diamond Property for $\rightarrow_d$)**.** *Suppose $P \rightarrow_d Q$ and $P \rightarrow_d R$. Either $Q \equiv_d R$ or exists S s.t. $Q \rightarrow_d S$ and $R \rightarrow_d S$.*

Proof. Apply Lemma 6 to obtain predexes $R_1, \ldots, R_n$ and a static n-hole context C such that $P \equiv C[R_1] \ldots [R_n]$ and $C \not\twoheadrightarrow$. The $\rightarrow_d$-reductions $P \rightarrow_d Q$ and $P \rightarrow_d R$ are either obtained by $\rightarrow_d$-reducing the same redex R_i, in which we case we apply Lemma 7, or they are obtained by $\rightarrow_d$-reducing distinct redexes R_i, R_j, in which case they commute. □

7.3 Factorisation of $\twoheadrightarrow$ through $\rightarrow_d$

In the previous section we have introduced the restricted form of reduction $\rightarrow_d$, which does not manipulate sums, and for which a diamond property was straightforward to establish. In this section we prove that the parallel-sum reduction $\twoheadrightarrow$, which does manipulate sums and, furthermore, allows summands to be reduced in parallel, factors through $\rightarrow_d$ (Lemma 9(3)). This factorisation result will then be used in the next section to lift the diamond property of $\rightarrow_d$ to $\twoheadrightarrow$ and $\overset{*}{\rightarrow}$.

The result works by essentially factorising each $\twoheadrightarrow$ reduction of a process P in a series of $\equiv$ manipulations that involve sums and then a series of $\rightarrow_d$ manipulations. More specifically, given a process P we compute an $\equiv$-equivalent sum expansion $P_1 + \ldots P_n$ in which all the summands are exposed. This computation is done essentially by interleaving all the concurrent take usages, applying $\equiv$ law [Tsh], and by pushing the sums outwards using $\equiv$ laws - [MSm], [CSm], [C!Sm] and [ShSm] - that distribute the static constructors mix, linear and unrestricted cut and share over sum. This canonical sum expansion exposes all the possible sum manipulations of P and, as a consequence, all the $\twoheadrightarrow$-reductions of P can be produced by $\rightarrow_d$-reducing one or more of the summands $P_1, \ldots, P_n$.

$$
\begin{array}{lcll}
\mathcal{S}(\mathcal{X}) & \triangleq & \{\mathcal{X}\} & [\mathcal{S}\mathcal{X}] \\
\mathcal{S}(\textbf{par}\ \{P \parallel Q\}) & \triangleq & \{\textbf{par}\ \{P_i \parallel Q_j\}\}_{i\in\mathcal{I},j\in\mathcal{J}} & [\mathcal{S}\mathrm{M}] \\
S(\textbf{cut}\ \{P\ \ |x|\ \ Q\}) & \triangleq & \{\textbf{cut}\ \{P_i\ \ |x|\ \ Q_j\}\}_{i\in\mathcal{I},j\in\mathcal{J}} & [\mathcal{S}\mathrm{C}] \\
\mathcal{S}(\textbf{cut!}\ \{y.P\ |x|\ Q\}) & \triangleq & \{\textbf{cut!}\ \{y.P\ |x|\ Q_j\}\}_{j\in\mathcal{J}} & [\mathcal{S}\mathrm{C!}] \\
S(\textbf{share}\ x\ \{P \parallel Q\}) & \triangleq & \bigcup_{i\in\mathcal{I},j\in\mathcal{J}} \mathcal{I}_x(P_i, Q_j) & [\mathcal{S}\mathrm{Sh}] \\
\mathcal{S}(P + Q) & \triangleq & \mathcal{S}(P) \cup \mathcal{S}(Q) & [\mathcal{S}\mathrm{Sm}]
\end{array}
$$

$$
\text{where } \mathcal{I}_x(P,Q) \triangleq \begin{cases} \{C \circ \mathcal{D}[\textbf{take}\ x(y); \textbf{share}\ x\ \{P' \parallel Q\}],\ C \circ \mathcal{D}[\textbf{take}\ x(z); \textbf{share}\ x\ \{P \parallel Q'\}]\}, \\ \text{if } P \equiv_d C[\textbf{take}\ x(y); P'] \text{ and } Q \equiv_d \mathcal{D}[\textbf{take}\ x(z); Q']. \\ \\ \{\textbf{share}\ x\ \{P \parallel Q\}\}, \text{otherwise.} \end{cases}
$$

Figure 7.1: Sum expansion map $\mathcal{S}(P)$.

In this section we will find it useful to work with finite nonempty multisets of processes $\{P_1, \ldots, P_n\}$ that represent sums $P_1 + \ldots + P_n$, as expressed by the following definition

Definition 22 ($\sum \mathcal{S}$)**.** *Given a nonempty multiset $\mathcal{S}$ of processes we define $\sum \mathcal{S}$ inductively by*

$$\sum\{P\} \triangleq P \quad \sum(\mathcal{S}, P) \triangleq (\sum \mathcal{S}) + P$$

Since sum + is associative and commutative modulo structural congruence $\equiv$ (laws [Sm] and [SmSm]), the order in which the processes are listed in the multiset $\mathcal{S}$ when computing $\sum \mathcal{S}$ is irrelevant. Now, we will define our sum expansion map that allows us to factorise $\twoheadrightarrow$ through $\rightarrow_d$.

Definition 23 (Sum Expansion Map $\mathcal{S}(P)$)**.** *For each process P the sum expansion map $\mathcal{S}(P)$ is defined by induction according to Fig. 7.1. In $[\mathcal{S}\mathcal{X}]$, we let $\mathcal{X}$ stand for* **0**, **fwd** *x y or any action. In $[\mathcal{S}\mathrm{M}]$, $[\mathcal{S}\mathrm{C}]$, $[\mathcal{S}\mathrm{C!}]$ and $[\mathcal{S}\mathrm{Sh}]$ we consider that*

$$\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}} \text{ and } \mathcal{S}(Q) = \{Q_j\}_{j\in\mathcal{J}}$$

The map $\mathcal{S}(R)$ takes as input a process R and computes its sum expansion $\{R_1, \ldots, R_n\}$. In the base case, where R is either the inaction process **0**, a forwarder or an action, then its sum expansion is simply the singleton $\{R\}$ (case $[S\mathcal{X}]$]). Since structural congruence $\equiv$ operates *on the nose*, sums guarded by actions (for example **wait** x; $(R_1 + R_2)$) are simply ignored.

The sum expansion of a mix **par** $\{P \parallel Q\}$ (case $[\mathcal{S}\mathrm{M}]$) is obtained by first computing the sum expansions of each argument P and Q

$$\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}} \text{ and } \mathcal{S}(Q) = \{Q_j\}_{j\in\mathcal{J}}$$

Then, $\mathcal{S}(\textbf{par}\ \{P \parallel Q\})$ is the defined as the set of all elements **par** $\{P_i \parallel Q_j\}$, where $i \in \mathcal{I}$ and $j \in \mathcal{J}$. Similarly for linear and unrestricted cut (cases $[\mathcal{S}\mathrm{C}]$ and $[\mathcal{S}\mathrm{C!}]$).

A similar process applies to share **share** x $\{P \mid\mid Q\}$ (case [$\mathcal{S}$Sh]) but instead of simply aggregating each component P_i of the sum expansion of P and each component Q_j of the sum expansion of Q in a share **share** x $\{P_i \mid\mid Q_j\}$, we need first to expose the possible interleavings resultant the shared usages of P_i and Q_j, as defined by the auxiliary interleaving map $\mathcal{I}_x(P_i, Q_j)$.

Map $\mathcal{I}_x(P, Q)$ computes the sum expansion resultant by interleaving two concurrent take actions, one coming from P and another from Q, as essentially expressed by $\equiv$ commuting law [TSh]. In the definition of $\mathcal{I}_x(P, Q)$ we assume that $\mathcal{C}, \mathcal{D}$ are static contexts where the holes are not guarded by share or sum constructs on x. Each process P and Q to which the interleaving map $\mathcal{I}_x(P, Q)$ applies is a component of some sum expansion, hence there are no unguarded sums and all concurrent share operations were previously interleaved. As a consequence, each process P and Q in $\mathcal{I}_x(P, Q)$ can offer at most one take action. If some of them cannot offer a take action, then there is no interleaving to carry out and we simply aggregate the components P and Q in a share.

The sum expansion of a sum is simply $P + Q$ is simply obtained by taking the multiset union of the sum expansion of P and the sum expansion of Q (case [$\mathcal{S}$Sm]). The following result states some properties about the sum expansion map.

Lemma 9. *The following properties hold*

(1) $\sum \mathcal{S}(P) \equiv P$, *for all* P.

(2) *Suppose* $P \equiv Q$. *Then*

(i) For all $P \in \mathcal{S}(P)$ *there exists* $Q' \in \mathcal{S}(Q)$ *s.t.* $P' \equiv_d Q'$.

(ii) For all $Q' \in \mathcal{S}(Q)$ *there exists* $P' \in \mathcal{S}(P)$ *s.t.* $P' \equiv_d Q'$.

(3) *Let* $\mathcal{S}(P) = \{P_i\}_{i \in \mathcal{I}}$. *If* $P \twoheadrightarrow Q$, *then exists* $\{Q_{ij}\}_{i \in \mathcal{I}, j \in \mathcal{J}_i}$ *s.t.* $Q \equiv \sum_{i \in \mathcal{I}, j \in \mathcal{J}_i} Q_{ij}$ *and for all* $i \in \mathcal{I}$, $j \in \mathcal{J}_i$, *either* $P_i \equiv_d Q_{ij}$ *or* $P_i \rightarrow_d Q_{ij}$.

Proof. We give a proof sketch. Details can be found in Appendix C.

(1) By structural induction in P and case analysis on its principal form. The equivalence $\sum \mathcal{S}(P) \equiv P$ is shown essentially by applying the structural congruence $\equiv$ laws that [MSm], [CSm], [C!Sm], [ShSm] that distribute the static constructors over sum and structural congruence law $\equiv$ [TSh] that computes the interleaving of two concurrent take actions.

(2) By induction on the structure of a derivation tree for $P \equiv Q$ and by case analysis on the root rule. Some cases of $\equiv$ immediately follow because they are also contained in $\equiv_d$ and therefore we can apply the conversion pointwise to the multiset $\mathcal{S}(P)$ (and symmetrically to $\mathcal{S}(Q)$), e.g. $\equiv$ law [M] that expresses the commutativity of the mix construct. Other cases follow because the two multisets $\mathcal{S}(P)$ and $\mathcal{S}(Q)$ are the same, e.g. $\equiv$ law [Sm].

(3) By induction on the structure of a derivation tree for $P \twoheadrightarrow Q$ and case analysis on the root rule. The principal cut reductions are handled straightforwardly since they do not involve sums, hence in those cases $\mathcal{S}(P)$ is simply the singleton $\{P\}$. Furthermore, all the principal cut reductions are also valid rules of $\rightarrow_d$. Case [cong] is handled by applying pointwise the congruence rule. Case [+par] is straightforward by invoking the inductive hypothesis on both summands P_1 and P_2 of $P = P_1 + P_2$. Case [≡] follows by (2).

□

Each process P can be written in an ≡-equivalent sum of basic processes given by its sum expansion $\mathcal{S}(P)$ (Lemma 9(1)). Property Lemma 9(2) relates the sum expansions $\mathcal{S}(P)$ and $\mathcal{S}(Q)$ of ≡-equivalent processes P and Q. It states that $\mathcal{S}(P)$ and $\mathcal{S}(Q)$ are essentially the same up to congruence $\equiv_d$. Furthermore, this notion of *sameness* ignores the multiplicity of $\equiv_d$-equivalent terms, which accounts for idempotency of sum ($P + P \equiv P$). That is, if two multisets $\mathcal{S}$ and $\mathcal{T}$ are the *same*, then so they are the multisets $\mathcal{S}, P$ and $\mathcal{T}, P$, where $P \equiv_d P'$ for some process P'. In other words, the quotient sets $\mathcal{S}(P)/\equiv_d$ and $\mathcal{S}(Q)/\equiv_d$ of two ≡-equivalent processes P and Q are the same.

Lemma 9(3) states that a $\twoheadrightarrow$-reduction of a process P can be factored through its sum expansion $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$ by $\rightarrow_d$-reducing many components P_i as many times as necessary. This is the crucial factorisation property which will allow us to lift the diamond property of $\rightarrow_d$ to $\twoheadrightarrow$ and hence to $\overset{*}{\rightarrow}$. We will show how in the next section.

7.4 Diamond Property for $\overset{*}{\rightarrow}$

In this section we conclude the technical development of this chapter with the proof our main result of global confluence (Theorem 3). We have proved before that the restricted form of reduction $\rightarrow_d$, which does not manipulate sums, satisfies the diamond property (Lemma 8). We have also showed that we can factor the parallel-sum $\twoheadrightarrow$ reduction through $\rightarrow_d$ (Lemma 9(3)). This factorisation result allows us to lift the diamond property from relation $\rightarrow_d$ to relation $\twoheadrightarrow$ (Lemma 11). Since $\overset{*}{\rightarrow} = \twoheadrightarrow^*$, this establishes our main result (Theorem 3).

Before presenting the proof of the diamond property for $\twoheadrightarrow$ we need the following technical auxiliary result.

Lemma 10. *Suppose $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. There are multisets $\{P_i\}_{i\in\mathcal{I}}$, $\{Q_i\}_{i\in\mathcal{I}}$ and $\{R_i\}_{i\in\mathcal{I}}$ s.t.*

(1) *$\sum_{i\in\mathcal{I}} P_i \equiv P$, $\sum_{i\in\mathcal{I}} Q_i \equiv Q$, $\sum_{i\in\mathcal{I}} R_i \equiv R$; and*

(2) *for all $i \in \mathcal{I}$, $P_i\ (\rightarrow_d \cup \equiv_d)\ Q_i$ and $P_i\ (\rightarrow_d \cup \equiv_d)\ R_i$.*

Proof. Let $\mathcal{S}(P) = \{P'_i\}_{1\le i\le n}$. By applying Lemma 9(3) to $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$ we conclude that exists $\{Q'_{ij}\}_{1\le i\le n, 1\le j\le n_i}$ and $\{R'_{ik}\}_{1\le i\le n, 1\le k\le m_i}$ s.t.

(a) $Q \equiv \sum_{1\le i\le n, 1\le j\le n_i} Q'_{ij}$, $R \equiv \sum_{1\le i\le n, 1\le k\le m_i} R'_{ik}$;

(b) $P_i \; (\to_d \cup \equiv_d) \; Q'_{ij}$, for all $1 \le i \le n, 1 \le j \le n_i$; and

(c) $P_i \; (\to_d \cup \equiv_d) \; R'_{ij}$, for all $1 \le i \le n, 1 \le k \le m_i$.

Define the indexed multisets

$$\{P_{ix}\}_{1\le i\le n, 1\le x\le \max(n_i,m_i)},\ \{Q_{ix}\}_{1\le i\le n, 1\le x\le \max(n_i,m_i)} \text{ and } \{R_{ix}\}_{1\le i\le n, 1\le x\le \max(n_i,m_i)}$$

by

$$P_{ix} \triangleq P'_i \quad Q_{ix} \triangleq \begin{cases} Q'_{ix}, x \le n_i \\ Q'_{in_i}, \text{ otherwise} \end{cases} \qquad R_{ix} \triangleq \begin{cases} R'_{ix}, x \le m_i \\ R'_{im_i}, \text{ otherwise} \end{cases}$$

The defined multisets have the same number of elements, given by $\sum_{1\le i\le n} \max(n_i, m_i)$, they correspond to the multisets $\{P_i\}_{i\in\mathcal{I}}, \{Q_i\}_{i\in\mathcal{I}}$ and $\{R_i\}_{i\in\mathcal{I}}$ of the statement of this lemma, where $\mathcal{I} = \{(i,x) \mid 1 \le i \le n,\ 1 \le x \le \max(n_i, m_i)\}$, and for which we will now prove properties (1) and (2).

We have

$$\begin{aligned} \sum_{1\le i\le n, 1\le x\le \max(n_i,m_i)} P_{ix} &= \sum_{1\le i\le n} \Big(\sum_{1\le x\le \max(n_i,m_i)} P_{ix} \Big) \\ &\equiv \sum_{1\le i\le n} P'_i && \text{(idempotency of sum)} \\ &\equiv P \end{aligned}$$

Similarly, one may derive

$$\textstyle\sum_{1\le i\le n, 1\le x\le \max(n_i,m_i)} Q_{ix} \equiv Q \quad \text{and} \quad \sum_{1\le i\le n, 1\le x\le \max(n_i,m_i)} \equiv R$$

This proves (1).

Let $1 \le i \le n$ and $1 \le x \le \max(n_i, m_i)$. Properties (b) and (c) imply (2). □

Lemma 10 is a consequence of the factorisation property Lemma 9(3), which we instantiate in the $\twoheadrightarrow$-reduction fork $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. We organised the factorisation a bit, taking advantage of idempotency of sum, so that we can map a $\twoheadrightarrow$-reduction fork to many $\to_d$-reduction forks. These $\to_d$-reduction forks can then be closed since $\to_d$ satisfies the diamond property, this then implies the diamond property for $\twoheadrightarrow$, as stated by the following lemma.

Lemma 11 (Diamond Property for $\twoheadrightarrow$)**.** *Suppose $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$. Either $Q \equiv R$ or there exists S s.t. $Q \twoheadrightarrow S$ and $R \twoheadrightarrow S$.*

Proof. Apply Lemma 10 to $P \twoheadrightarrow Q$ and $P \twoheadrightarrow R$ in order to infer the existence of multisets $\{P_i\}_{i\in\mathcal{I}}, \{Q_i\}_{i\in\mathcal{I}}$ and $\{R_i\}_{i\in\mathcal{I}}$ for which Lemma 10(1) and Lemma 10(2) hold.

We will now define, by case analysis $\{S_i\}_{i\in\mathcal{I}}$:

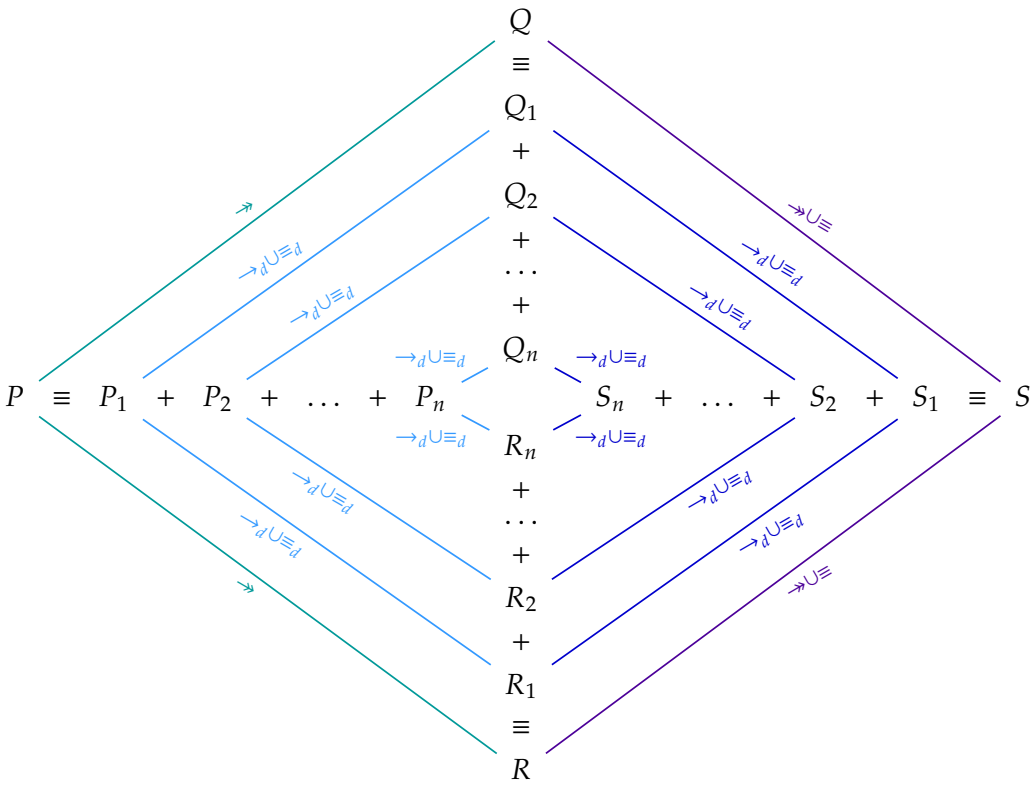


Figure 7.2: Diagram illustrating proof of confluence.

- If $P_i \equiv_d Q_i$ and $P_i \equiv_d R_i$, let $S_i \triangleq P_i$.
- If $P_i \equiv_d Q_i$ and $P_i \to_d R_i$, let $S_i \triangleq R_i$.
- If $P_i \to_d Q_i$ and $P_i \equiv_d R_i$, let $P_i \triangleq Q_i$.
- If $P_i \to_d Q_i$ and $P_i \to_d R_i$, let S_i be s.t. that either $S_i \equiv_d Q_i \equiv_d R_i$ or $Q_i \to_d S_i$ and $R_i \to_d S_i$. Such an S_i is guaranteed to exist because $\to_d$ satisfies the diamond property. (Lemma 8).

By construction, both Q_i $(\to_d \cup \equiv_d)$ S_i and R_i $(\to_d \cup \equiv_d)$ S_i, for all $i \in \mathcal{I}$. Let $S = \sum_{i \in \mathcal{I}} S_i$. By iterative application of rule [+par] (Def. 19) we obtain both Q $(\equiv \cup \twoheadrightarrow)$ S and R $(\equiv \cup \twoheadrightarrow)$ S. The proof is illustrated in the diagram of Figure 7.2. The process sum expansions and light-blue edges are inferred by applying Lemma 10, which essentially allow us to factor the $\twoheadrightarrow$-reduction fork through many $\to_d$-reduction forks. Dark-blue edges are inferred because $\to_d$ satisfies the diamond property (Lemma 8). Finally, since we are allowed to $\twoheadrightarrow$-reduce parallel independent summands simultaneously, by applying rule [+par] (Def. 19), this justifies the purple edges. □

We conclude this section with the proof our main result.

Theorem 3 (Diamond Property for $\overset{*}{\longrightarrow}$)**.** *Suppose $P \overset{*}{\longrightarrow} Q$ and $P \overset{*}{\longrightarrow} R$. Either $Q \equiv R$ or there exists S s.t. $Q \overset{*}{\longrightarrow} S$ and $R \overset{*}{\longrightarrow} S$.*

Proof. Follows from Lemma 11 and Lemma 5(3). □

7.5 Further Discussion and Related Work

In this chapter we have proved that the reduction relation $\longrightarrow$ of our our language CLASS, with first-class reference cells, is confluent (Theorem 3) modulo $\equiv$.

CLASS is the first language that accommodates shared state into the propositions-as-types correspondence between linear logic and session-based concurrency [23, 157], without sacrificing the confluence property. For example, the session-typed calculi presented in [10, 128, 7, 91] do not capture the nondeterminism that naturally emerges from racy concurrent processes with non-collapsing sums, as we do in CLASS, and, hence, fail to be confluent. As we have already mentioned, confluence is a necessary requirement for calculi based on proposition-as-types, where reduction is to be interpreted as a proof equivalence.

The idea of using sums to internalise nondeterministic computations and obtain a confluent notion of reduction is not new, being explored, for example, in the context of variants of the Lambda Calculus such as the Differential Lambda Calculus [55] and the Resource Lambda Calculus [119]; in the context of Differential Interaction Nets [54, 151, 152]; and even in the context of process calculi, like in the work of Beffara [13], which presents an extension of the πI-calculus (polyadic π calculus with internal mobility) with sums that enjoys local confluence.

Our proof of confluence (Theorem 3) is based on the the Tait and Martin-Löf proof technique, which was employed by W. Tait and P. Martin-Löf to show the confluence of β-reduction for the Lambda Calculus [12]. It consists of proving the diamond property for a relation $\mathcal{R}$ by finding a relation $\mathcal{S}$ which (1) satisfies the diamond property and (2) whose reflexive-transitive closure $\mathcal{S}^*$ equates $\mathcal{R}$. For the case of the Lambda Calculus, relation $\mathcal{S}$ is instantiated by a *parallel* version of β-reduction which allows to reduce a number of redexes in a λ-term simultaneously [142].

Interestingly, in our proof, the candidate relation $\mathcal{S}$ is just a more permissible relation than reduction $\longrightarrow$ which allows independent summands to be reduced simultaneously (Def. 19). However, it is not necessary to allow the simultaneous reduction of the arguments P and Q in a cut $\mathbf{cut}\ \{P \ \ |x| \ \ Q\}$, for example.

In [121], the authors prove confluence for session-typed process calculus based on the correspondence with intuitionistic linear logic developed in [23], by applying the technique of the linear logical relations. The session-typed calculus introduced in [23] is essentially functional, hence their proof of confluence does not need to handle sums as we have to in our proof. It remains to explore if the technique of linear logical relations developed in [121] can be adapted to yield confluence for CLASS.

8

Cut Normalisation

Cut elimination states that any provable judgment has a proof that does not make use of the cut rule and was originally proved by Gentzen for intuitionistic and classical logic (Gentzen's *Hauptsatz*) [59]. It is a result with many deep consequences, among which the consistency of a logical system, the subformula property and various resolution techniques, which are important tools in the development of automated theorem provers. In fact, the central role that cut elimination plays in logic lead Girard once to affirm that *a sequent calculus without cut-elimination is like a car without engine* [64].

In this chapter, we study a cut normalisation result for $\mathsf{CLASS}\backslash\exists\mu$, which stands for the session-typed process calculus CLASS that extends classical linear logic with first-class reference cells, as defined in Chapter 3, but without the second-order quantifiers $\exists X.A, \forall X.A$ and inductive/coinductive $\mu X.\ A, \nu X.\ A$ session types.

Namely, we establish the Cut-Normalisation Theorem 4 which states that every typed process of $\mathsf{CLASS}\backslash\exists\mu$ can be rewritten into an equivalent normal form on which there are no cuts, except on open identity axioms on shared usages: we call open cells to such cuts. This motivates the following two definitions

Definition 24 (Open Cell)**.** *An open cell is a process of the form*

$$\mathbf{cut}\ \{C(x)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z\ ||\ Q\}\}$$

where either $C(x) = \mathbf{cell}\ x(y.P)$ *or* $C(x) = \mathbf{empty}\ x$.

Definition 25 (Normal form)**.** *A process is a normal form if it contains no cuts except open cells.*

Open cells have a simple form $\mathbf{cut}\ \{C(x)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z\ ||\ Q\}\}$, where the usage x on the left argument of the share construct is forwarded to a free name z. The share in the open cell cannot be converted to a sum of $\mathbf{U}_f\ \overline{A}$ introduction forms or to an $\mathbf{U}_e\ \overline{A}$ introduction form, since $\mathbf{fwd}\ x\ z$ is not an introduction form, and offers no structure at z. There is no real redex in an open cell: the share is suspended on the availability of cell usages at open name z from the environment. This justifies open cells as normal forms.

The necessary transformations to obtain cut normalisation are captured by a congruence relation $\approx$ on type derivations, that contains structural congruence $\equiv$ and reduction

→, and adds a complete set of commuting conversions along standard lines [23, 24, 157]. The new conversions of ≈ essentially allow actions to be commuted with share, linear and unrestricted cut constructs as well as unrestricted cuts to be discarded from the leaves of type derivation trees.

The Cut Normalisation Theorem follows by the the Cut Normalisation Lemma 15, which constructs an ≈-equivalent normal form for a cut $\mathbf{cut}\ \{P\ |x|\ Q\}$, when given normal forms for P and Q. The proof of the Cut Normalisation Lemma is established by adapting Frank Pfenning's structural cut elimination technique for classical linear logic [122, 123].

The proof relies on an auxiliary Share Expansion Lemma 14, that is applied when normalising cuts between a cell and share on the same session c. This works by expanding the share of usage c into an ≈-equivalent sum of sequential cell usages on c, after which the cut distributes over the sum and normalisation proceeds as usual in each independent summand. Of course, ff the share is suspended on a forwarder $\mathbf{fwd}\ c\ c'$, expansion is not possible, in which case we obtain an open cell.

The Cut Normalisation Theorem yields some corollaries, namely a Subformula Property Corollary 1 and a Cut Elimination for Pure Sequents Corollary 2, the latter states that it is always possible to eliminate cuts from pure sequents $P \vdash \Delta; \Gamma$ in which the cell state and usage modalities are not present in the typing context $\Delta; \Gamma$.

Interestingly, Corollary 2 exposes a strong conservativeness and expressiveness result about the language $\mathsf{CLASS}\backslash\exists\mu$. It implies that any process $P \vdash \Delta; \Gamma$ in $\mathsf{CLASS}\backslash\exists\mu$ that potentially uses shared state internally but that implements an interface $\Delta; \Gamma$ only manifesting standard propositional pure session types is ≈-equivalent to a (possibly nondeterministic and larger) process $Q \vdash \Delta; \Gamma$ that *does not use imperative constructs at all*. This normal form Q expresses the externally observable behaviour of the original stateful open process P. Interestingly, this normal form can be computed by doing simple algebraic manipulations as expressed by ≈ laws, we illustrate with a comprehensive example in 16.

The rest of the chapter is organised as follows. Section 8.1 defines relation ≈, introducing a complete set of commuting conversions for cut-normalisation and share-expansion. Section 8.2 defines the #-measure on processes, proves the Share Expansion Lemma 14 and the Cut Normalisation Lemma 15, from which we derive the Cut Normalisation Theorem 4. Section 8.3 presents some corollaries of cut normalisation, namely the Subformula Property Corollary 1 and Cut Elimination for Pure Sequents Corollary 2. Finally, Section 8.4 concludes with further discussion and related work.

8.1 The Relation ≈ : A Complete Set of Commuting Conversions

In this section we define relation ≈ (Def. 26), which extends structural congruence ≡ and → with a sufficient set of proof transformations, that allows us to establish the cut normalisation result.

To illustrate why further conversions are necessary, consider the following process

$$\mathbf{cut}\ \{\mathbf{wait}\ y; \mathbf{close}\ x\ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\}$$

The cut on session x cannot be eliminated solely by applying the commuting conversions of $\equiv$ and the principal cut conversions of $\rightarrow$. It is necessary to commute the cut with the wait action on name y in order to expose the redex, after which the principal cut conversion $\rightarrow$ [$\mathbf{1}\bot$] can then be applied

$$\begin{array}{rcl} \mathbf{cut}\ \{\mathbf{wait}\ y; \mathbf{close}\ x\ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\} & \approx & \mathbf{wait}\ y; (\mathbf{cut}\ \{\mathbf{close}\ x\ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\}) \\ & \rightarrow & \mathbf{wait}\ y; P \end{array}$$

The relation $\approx$ is defined by the following

Definition 26. *$\approx$ is a congruence relation on type derivations of* CLASS$\backslash\exists\mu$ *that contains structural congruence $\equiv$ (Def. 13) and reduction $\rightarrow$ (Def. 14) and includes the commuting conversions listed in figs. 8.1, 8.2,8.3 as well as the unrestricted cut discarding principles in fig. 8.4. In all the rules of figs. 8.1, 8.2 and 8.3 we consider the subject of the prefixed action y to be distinct of x. Furthermore, in all the rules of figs.. 8.1 and 8.2 we consider that $y \notin \mathsf{fn}(P)$.*

Relation $\approx$ is defined on type derivation trees. However, to be concise, the conversion rules of $\approx$ are written on processes. Nevertheless, from each rule on processes we can always obtain the corresponding rule on type derivations, for example rule [C$\bot$] $\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{wait}\ y; Q\} \approx \mathbf{wait}\ y; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ corresponds to the conversion on type derivations

$$\dfrac{P_1 \vdash \Delta_1, x : \overline{A}; \Gamma \qquad \dfrac{P_2 \vdash \Delta_2, x : A; \Gamma}{\mathbf{wait}\ y; P_1 \vdash \Delta_2, x : A, y : \bot; \Gamma}}{\mathbf{cut}\ \{P_1\ \ |x|\ \ \mathbf{wait}\ y; P_2\} \vdash \Delta_1, \Delta_2, y : \bot; \Gamma} \approx \dfrac{\dfrac{P_1 \vdash \Delta_1, x : \overline{A}; \Gamma \quad P_2 \vdash \Delta_2, x : A; \Gamma}{\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \vdash \Delta_1, \Delta_2; \Gamma}}{\mathbf{wait}\ y; (\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}) \vdash \Delta_1, \Delta_2, y : \bot; \Gamma}$$

The rules listed in figures 8.1, 8.2 8.3 commute a share, a linear cut or an unrestricted cut on name x with an action of subject y distinct of x and all are essential to establish the Cut Normalisation Lemma 15. The role of the cut-action commutation principles was already made clear with the previous example. To illustrate the importance of the share-action commutation rules, which are necessary to establish the Share Expansion Lemma 14, consider a process of the form

$$\mathbf{cut}\ \{\mathbf{cell}\ x(v.P)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathbf{wait}\ y; \mathbf{release}\ x\ ||\ \mathbf{put}\ x(v.Q); \mathbf{release}\ x\}\}$$

We cannot commute a cut with a share construct on the same name x, so the only possibility is to first expand the share by applying $\approx$ rule [Sh$\bot$], followed by $\equiv$ rule [RSh]

$$\mathbf{share}\ x\ \{\mathbf{wait}\ y; \mathbf{release}\ x\ ||\ \mathbf{put}\ x(v.Q); \mathbf{release}\ x\} \approx \mathbf{wait}\ y; \mathbf{put}\ x(v.Q); \mathbf{release}\ x$$

$\mathbf{share}\ x\ \{\mathbf{wait}\ y; P \parallel Q\} \approx \mathbf{wait}\ y; \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\bot$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{send}\ y(z.Q_1); Q_2\} \approx \mathbf{send}\ y(z.(\mathbf{share}\ x\ \{P \parallel Q_1\})); Q_2,\ x \in \mathsf{fn}(Q_1)$ [Sh$\otimes_1$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{send}\ y(z.Q_1); Q_2\} \approx \mathbf{send}\ y(z.Q_1); \mathbf{share}\ x\ \{P \parallel Q_2\},\ x \in \mathsf{fn}(Q_2)$ [Sh$\otimes_2$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{recv}\ y(z); Q\} \approx \mathbf{recv}\ y(z); \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\parr$]

$\mathbf{share}\ x\ \{P \parallel y.\mathbf{inl}; Q\} \approx y.\mathbf{inl}; \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\oplus_l$]

$\mathbf{share}\ x\ \{P \parallel y.\mathbf{inr}; Q\} \approx y.\mathbf{inr}; \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\oplus_r$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{case}\ y\ \{|\mathsf{inl} : Q_1 \mid \mathsf{inr} : Q_2\}\}$
$\quad \approx \mathbf{case}\ y\ \{|\mathsf{inl} : \mathbf{share}\ x\ \{P \parallel Q_1\} \mid \mathsf{inr} : \mathbf{share}\ x\ \{P \parallel Q_2\}\}$ [Sh&]

$\mathbf{share}\ x\ \{P \parallel ?y; Q\} \approx ?y; \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh?]

$\mathbf{share}\ x\ \{P \parallel \mathbf{call}\ y(z); Q\} \approx \mathbf{call}\ y(z); \mathbf{share}\ x\ \{P \parallel Q\}$ [ShCall]

$\mathbf{share}\ x\ \{P \parallel \mathbf{sendty}\ y\ A; Q\} \equiv_{\mathsf{c}} \mathbf{sendty}\ y\ A; \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\exists$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{recvty}\ y(X); Q\} \equiv_{\mathsf{c}} \mathbf{recvty}\ y(X); \mathbf{share}\ x\ \{P \parallel Q\}$ [Sh$\forall$]

$\mathbf{share}\ x\ \{P \parallel \mathbf{unfold}_{\mu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\mu}\ y; \mathbf{share}\ x\ \{P \parallel Q\}$ [Shμ]

$\mathbf{share}\ x\ \{P \parallel \mathbf{unfold}_{\nu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\nu}\ y; \mathbf{share}\ x\ \{P \parallel Q\}$ [Shν]

$\mathbf{share}\ x\ \{P \parallel \mathbf{affine}\ y; Q\} \approx \mathbf{affine}\ y; \mathbf{share}\ x\ \{P \parallel Q\}$ [ShAffine]

$\mathbf{share}\ x\ \{P \parallel \mathbf{use}\ y; Q\} \approx \mathbf{use}\ y; \mathbf{share}\ x\ \{P \parallel Q\}$ [ShUse]

$\mathbf{share}\ x\ \{P \parallel \mathbf{cell}\ y(z.Q)\} \approx \mathbf{cell}\ y(z.\mathbf{share}\ x\ \{P \parallel Q\})$ [ShCell]

$\mathbf{share}\ x\ \{P \parallel \mathbf{take}\ y(z); Q\} \approx \mathbf{take}\ y(z); \mathbf{share}\ x\ \{P \parallel Q\}$ [ShTake]

$\mathbf{share}\ x\ \{P \parallel \mathbf{put}\ y(z.Q_1); Q_2\} \equiv_{\mathsf{c}} \mathbf{put}\ y(z.\mathbf{share}\ x\ \{P \parallel Q_1\}); Q_2,\ x \in \mathsf{fn}(Q_1)$[ShPut1]

$\mathbf{share}\ x\ \{P \parallel \mathbf{put}\ y(z.Q_1); Q_2\} \approx \mathbf{put}\ y(z.Q_1); \mathbf{share}\ x\ \{P \parallel Q_2\},\ x \in \mathsf{fn}(Q_2)$ [ShPut2]

Figure 8.1: Share-action commuting conversions ≈.

after which the cut can then be normalised

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{cell}\ c(v.P)\ \ |x|\ \ \mathbf{wait}\ y; \mathbf{put}\ x(v.Q); \mathbf{release}\ x\} \\
&\approx \mathbf{wait}\ y; (\mathbf{cut}\ \{\mathbf{cell}\ c(v.P)\ \ |x|\ \ \mathbf{put}\ x(v.Q); \mathbf{release}\ x\}) \\
&\xrightarrow{+} \mathbf{wait}\ y; \mathbf{0}
\end{aligned}$$

The discarding principles [C!$\mathbf{0}$], [C!$\mathbf{1}$], [C!fwd] and [C!free], when read from left to

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{wait}\ y; Q\} \approx \mathbf{wait}\ y; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [C$\bot$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{send}\ y(z.Q_1); Q_2\} \approx \mathbf{send}\ y(z.(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q_1\})); Q_2,\ x \in \mathsf{fn}(Q_1)$ [C$\otimes_1$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{send}\ y(z.Q_1); Q_2\} \approx \mathbf{send}\ y(z.Q_1); (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q_2\}),\ x \in \mathsf{fn}(Q_2)$ [C$\otimes_2$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{recv}\ y(z); Q\} \approx \mathbf{recv}\ y(z); (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [C$\parr$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ y.\mathbf{inl}; Q\} \approx y.\mathbf{inl}; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [C$\oplus_l$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ y.\mathbf{inr}; Q\} \approx y.\mathbf{inr}; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [C$\oplus_r$]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{case}\ y\ \{|\mathsf{inl} : Q_1 \mid \mathsf{inr} : Q_2\}\}$
$\quad \approx \mathbf{case}\ y\ \{|\mathsf{inl} : (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q_1\}) \mid \mathsf{inr} : (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q_2\})\}$ [C&]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ ?y; Q\} \approx ?y; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [C?]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{call}\ y(z); Q\} \approx \mathbf{call}\ y(z); (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [CCall]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{sendty}\ y\ A; Q\} \equiv_{\mathsf{c}} \mathbf{sendty}\ y\ A; \mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ [C$\exists$]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{recvty}\ y(X); Q\} \equiv_{\mathsf{c}} \mathbf{recvty}\ y(X); \mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ [C$\forall$]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{unfold}_{\mu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\mu}\ y; \mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ [Cμ]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{unfold}_{\nu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\nu}\ y; \mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ [Cν]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{affine}\ y; Q\} \approx \mathbf{affine}\ y; \mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}$ [CAffine]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{use}\ y; Q\} \approx \mathbf{use}\ y; \mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}$ [CUse]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{cell}\ y(z.Q)\} \approx \mathbf{cell}\ y(z.(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}))$ [CCell]

$\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{take}\ y(z); Q\} \approx \mathbf{take}\ y(z); (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})$ [CTake]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{put}\ y(z.Q_1); Q_2\} \equiv_{\mathsf{c}} \mathbf{put}\ y(z.\mathbf{cut}\ \{P\ \ |x|\ \ Q_1\}); Q_2,\ x \in \mathsf{fn}(Q_1)$[CPut1]

$\mathbf{cut}\ \{P\ \ |x|\ \ \mathbf{put}\ y(z.Q_1); Q_2\} \approx \mathbf{put}\ y(z.Q_1); \mathbf{cut}\ \{P\ \ |x|\ \ Q_2\},\ x \in \mathsf{fn}(Q_2)$ [CPut2]

Figure 8.2: Cut-action commuting conversions $\approx$.

right, allow us to delete an unrestricted cut with server body P composed with an axiom. The server will never be invoked and hence can be safely discarded. The distributive $\equiv$ conversions associated with the unrestricted cut allows us to lift the discarding principles to type derivations in general, as stated by the following result.

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{wait}\ y; Q\} \approx \mathbf{wait}\ y; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!⊥]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{send}\ y(z.Q_1); Q_2\}$
$\quad \approx \mathbf{send}\ y(z.(\mathbf{cut!}\ \{w.P\ |x : A|\ Q_1\})); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q_2\})$ [C!⊗]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{recv}\ y(z); Q\} \approx \mathbf{recv}\ y(z); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!⅋]

$\mathbf{cut!}\ \{w.P\ |x : A|\ y.\mathbf{inl}; Q\} \approx y.\mathbf{inl}; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!$\oplus_l$]

$\mathbf{cut!}\ \{w.P\ |x : A|\ y.\mathbf{inr}; Q\} \approx y.\mathbf{inr}; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!$\oplus_r$]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{case}\ y\ \{|\mathsf{inl} : Q_1 \mid \mathsf{inr} : Q_2\}\}$
$\quad \approx \mathbf{case}\ y\ \{|\mathsf{inl} : (\mathbf{cut!}\ \{w.P\ |x : A|\ Q_1\}) \mid \mathsf{inr} : (\mathbf{cut!}\ \{w.P\ |x : A|\ Q_2\})\}$ [C!&]

$\mathbf{cut!}\ \{w.P\ |x : A|\ ?y; Q\} \approx ?y; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!?]

$\mathbf{cut!}\ \{w.P\ |x : A|\ !y(z); Q\} \approx !y(z); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!!]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{call}\ y(z); Q\} \approx \mathbf{call}\ y(z); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!Call]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{sendty}\ y\ A; Q\} \equiv_{\mathsf{c}} \mathbf{sendty}\ y\ A; \mathbf{cut!}\ \{w.P\ |x : A|\ Q\}$ [C!∃]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{recvty}\ y(X); Q\} \equiv_{\mathsf{c}} \mathbf{recvty}\ y(X); \mathbf{cut!}\ \{w.P\ |x : A|\ Q\}$ [C!∀]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{unfold}_{\mu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\mu}\ y; \mathbf{cut!}\ \{w.P\ |x : A|\ Q\}$ [C!μ]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{unfold}_{\nu}\ y; Q\} \equiv_{\mathsf{c}} \mathbf{unfold}_{\nu}\ y; \mathbf{cut!}\ \{w.P\ |x : A|\ Q\}$ [C!ν]
$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{affine}\ y; Q\} \approx \mathbf{affine}\ y; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!Affine]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{use}\ y; Q\} \approx \mathbf{use}\ y; (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!Use]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{cell}\ y(z.Q)\} \approx \mathbf{cell}\ y(z.(\mathbf{cut!}\ \{w.P\ |x : A|\ Q\}))$ [C!Cell]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{take}\ y(z); Q\} \approx \mathbf{take}\ y(z); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q\})$ [C!Take]

$\mathbf{cut!}\ \{w.P\ |x : A|\ \mathbf{put}\ y(z.Q_1); Q_2\}$
$\quad \approx \mathbf{put}\ y(z.(\mathbf{cut!}\ \{w.P\ |x : A|\ Q_1\})); (\mathbf{cut!}\ \{w.P\ |x : A|\ Q_2\})$ [C!Put]

Figure 8.3: Cut!-action commuting conversions ≈.

Lemma 12. *Let* $P \vdash y : \overline{A}; \Gamma$ *and* $Q \vdash \Delta; \Gamma, x : A$. *Then, the rule*

$$\mathbf{cut!}\ \{y.P\ |x|\ Q\} \approx Q,\ x \notin \mathsf{fn}(Q) \quad \textit{[C!Discard]}$$

is derivable.

Proof. By structural induction on a type derivation tree for $Q \vdash \Delta; \Gamma$. We perform case analysis on the root rule of the tree. We will illustrate the proof with some representative

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{0} \vdash \emptyset; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{0}\} \vdash \emptyset; \Gamma} \approx \mathbf{0} \vdash \emptyset; \Gamma \quad [\text{C!}\mathbf{0}]$$

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{close}\ z \vdash z : \mathbf{1}; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{close}\ z\} \vdash z : \mathbf{1}; \Gamma} \approx \mathbf{close}\ z \vdash z : \mathbf{1}; \Gamma \quad [\text{C!}\mathbf{1}]$$

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{fwd}\ z\ w \vdash z : B, w : \overline{B}; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{fwd}\ z\ w\} \vdash z : B, w : \overline{B}; \Gamma} \approx \mathbf{fwd}\ z\ w \vdash z : B, w : \overline{B}; \Gamma \quad [\text{C!fwd}]$$

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{discard}\ z \vdash z : \vee B; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{discard}\ z\} \vdash z : \vee B; \Gamma} \approx \mathbf{discard}\ z \vdash z : \vee B; \Gamma \quad [\text{C!discard}]$$

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{release}\ z \vdash z : \mathbf{U}_f\ B; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{release}\ z\} \vdash z : \mathbf{U}_f\ B; \Gamma} \approx \mathbf{release}\ z \vdash z : \mathbf{U}_f\ B; \Gamma \quad [\text{C!release}]$$

$$\frac{P \vdash y : \overline{A}; \Gamma \qquad \mathbf{empty}\ z \vdash z : \mathbf{U}_e\ B; \Gamma, x : A}{\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{empty}\ z\} \vdash z : \mathbf{U}_e\ B; \Gamma} \approx \mathbf{empty}\ z \vdash z : \mathbf{U}_e\ B; \Gamma \quad [\text{C!empty}]$$

Figure 8.4: Cut! discarding conversions $\approx$.

cases. The remaining cases are handled similarly.

Case: [T**0**]. Applying [C!**0**] yields $\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{0}\} \approx \mathbf{0}$. Similarly for [T**1**], [Tfwd], [Tdiscard], [Trelease] and [Tempty].

Case: [T$\bot$]. We have

$$\begin{aligned} \mathbf{cut!}\ \{y.P\ |x|\ \mathbf{wait}\ z; Q\} &\approx \mathbf{wait}\ z; (\mathbf{cut!}\ \{y.P\ |x|\ Q\}) && ([\text{C!}\bot]) \\ &\approx \mathbf{wait}\ z; Q && (\text{i.h. applied to } Q) \end{aligned}$$

Case: [Tmix]. We have

$$\begin{aligned} \mathbf{cut!}\ \{y.P\ |x|\ (\mathbf{par}\ \{Q_1 \mid\mid Q_2\})\} &\approx \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x|\ Q_1\}) \mid\mid (\mathbf{cut!}\ \{y.P\ |x|\ Q_2\})\} \\ & && ([\text{D-C!M}]) \\ &\approx \mathbf{par}\ \{Q_1 \mid\mid Q_2\} && (\text{i.h. applied to } Q_1 \text{ and } Q_2) \end{aligned}$$

Case: [Tcall]. Since $x \notin \mathrm{fn}(\mathbf{call}\ z(w); Q)$, we conclude that the subject of the introduced action is $z \neq x$. We have

$$\begin{aligned} \mathbf{cut!}\ \{y.P\ |x|\ \mathbf{call}\ z(w); Q\} &\approx \mathbf{call}\ z(w); (\mathbf{cut!}\ \{y.P\ |x|\ Q\}) && ([\text{C!Call}]) \\ &\approx \mathbf{call}\ z(w); Q && (\text{i.h. applied to } Q) \end{aligned}$$

□

Interestingly, our axiomatisation is now redundant since we can derive commuting conversions of the form [C!X], where X ranges through the static operators mix (M), cut (C), unrestricted cut (C!) and share (Sh), from the commuting conversions [C!Discard] and [D-C!X]. For example, the commuting conversion [C!M]

$$\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{par}\ \{Q\ ||\ R\})\} \equiv \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ R\}, x \notin \mathsf{fn}(R)$$

is derivable by

$$\begin{aligned}\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{par}\ \{Q\ ||\ R\})\} &\approx \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \\ &\qquad\qquad \text{([D-C!M])} \\ &\approx \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ R\} \\ &\qquad\qquad \text{([C!Discard], since } x \notin \mathsf{fn}(R)\text{)}\end{aligned}$$

8.2 Share Expansion and Cut Normalisation

The goal of this section is to establish the Cut Normalisation Theorem 4 which follows from the Share Expansion Lemma 14 and the Cut Normalisation Lemma 15. Lemmas 14 and 15 apply the conversions of $\approx$ in a given direction so as to expand a share operation into a sum of processes or so as to push a cut down the derivation tree. Crucially, some of these transformations decrease a measure # on processes, as introduced by the following definition. In this section we consider all processes P to be from the subcalculus $\mathsf{CLASS}\backslash\exists\mu$ without quantifiers and inductive/coinductive types.

Definition 27 (The Measure $\#P$)**.** *We define $\#P$ by induction and pattern matching on P*

$$\#(\mathbf{0}) \triangleq 1 \quad \#(\mathbf{fwd}\ x\ y) \triangleq 2 \quad \#(\mathbf{close}\ x) \triangleq 2 \quad \#(\mathbf{discard}\ x) \triangleq 2 \quad \#(\mathbf{release}\ x) \triangleq 2 \quad \#(\mathbf{empty}\ x) \triangleq 2$$
$$\#(\mathbf{wait}\ x; P) \triangleq 1 + \#P \quad \#(\mathbf{send}\ x(y.P); Q) \triangleq 1 + \#P \times \#Q \quad \#(\mathbf{recv}\ x(y); P) \triangleq 1 + \#P$$
$$\#(x.\mathbf{inl}; P) \triangleq 1 + \#P \quad \#(x.\mathbf{inr}; P) \triangleq 1 + \#P \quad \#(\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}) \triangleq 1 + \#P + \#Q$$
$$\#(!x(y); P) \triangleq 1 + \#P \quad \#(?x; P) \triangleq 1 + \#P \quad \#(\mathbf{call}\ x(y); P) \triangleq 1 + \#P$$
$$\#(\mathbf{affine}\ x; P) \triangleq 1 + \#P \quad \#(\mathbf{use}\ x; P) \triangleq 1 + \#P$$
$$\#\mathbf{cell}\ x(y.P) \triangleq 1 + \#P \quad \#(\mathbf{take}\ x(y); P) \triangleq 1 + \#P \quad \#(\mathbf{put}\ x(y.P); Q) \triangleq 1 + \#P \times \#Q$$
$$\#(\mathbf{par}\ \{P\ ||\ Q\}) \triangleq \#P \times \#Q \quad \#(\mathbf{cut}\ \{P\ |x|\ Q\}) \triangleq \#P \times \#Q \quad \#(\mathbf{cut!}\ \{y.P\ |x|\ Q\}) \triangleq \#P \times \#Q$$
$$\#(\mathbf{share}\ x\ \{P\ ||\ Q\}) \triangleq \#P \times \#Q \quad \#(P + Q) \triangleq \#P + \#Q$$

It is easy to check that # is invariant under name substitution, i.e. $\#P = \#(\{x/y\}P)$. Furthermore, the following principles hold

Lemma 13. *Let $P \vdash \Delta; \Gamma$. Then, either*

(i) $\#P > 1$ *or*

(ii) $\#P = 1$ *and* $\Delta = \emptyset$.

Proof. Follows straightforwardly by induction on a type derivation tree for $P \vdash \Delta; \Gamma$. We illustrate with some cases.

Case: [T**0**]. We have $P = \mathbf{0}$ and $\Delta = \emptyset$. Since $\#\mathbf{0} = 1$, (ii) holds.

Case: [Tfwd]. We have $P = \mathbf{fwd}\ x\ y$. Since $\#\mathbf{fwd}\ x\ y = 2$, (i) holds. Similar for cases [T**1**], [Tdiscard], [Tempty] and [Trelease].

Case: [T$\bot$]. We have $P = \mathbf{wait}\ x; P'$, for some $P' \vdash \Delta'; \Gamma$.

I.h. applied to $P' \vdash \Delta'; \Gamma$ yields $\#P' > 0$.

Then

$$\#(\mathbf{wait}\ x; P') = 1 + \#P' > 1 + 0 = 1$$

which implies (i).

Similarly for cases [T$\parr$], [T$\otimes$], [T$\oplus_l$], [T$\oplus_r$], [T$\&$], [T!], [T?],[Tcall], [Taffine], [Tuse], [Tcell], [Ttake] and [Tput].

Case: [Tmix]. We have $P = \mathbf{par}\ \{P_1 \mid\mid P_2\}$, where $P_1 \vdash \Delta_1; \Gamma$ and $P_2 \vdash \Delta_2; \Gamma$ for some $\Delta_1, \Delta_2 = \Delta$.

By applying i.h. to $P_1 \vdash \Delta_1; \Gamma$ and $P_2 \vdash \Delta_2; \Gamma$, we conclude that both $\#P_1 \geq 1$ and $\#P_2 \geq 1$ and one of the following cases hold

Case: Either $\#P_1 > 1$ or $\#P_2 > 1$.

Suppose w.l.o.g. that $\#P_1 > 1$.

Then

$$\#(\mathbf{par}\ \{P_1 \mid\mid P_2\}) = \#P_1 \times \#P_2 \geq \#P_1 > 1$$

which implies (i).

Case: Both $\#P_1 = \#P_2 = 1$, in which case $\Delta_1 = \Delta_2 = \emptyset$.

Then $\Delta = \emptyset$ and $\#(\mathbf{par}\ \{P_1 \mid\mid P_2\}) = \#P_1 \times \#P_2 = 1 \times 1 = 1$.

Case: [Tcut]. We have $P = \mathbf{cut}\ \{P_1\ \ |x : A|\ \ P_2\}$, where $P_1 \vdash \Delta_1, x : A; \Gamma$ and $P_2 \vdash \Delta_2, x : \overline{A}; \Gamma$ for some $\Delta_1, \Delta_2 = \Delta$.

Applying i.h. to $P_1 \vdash \Delta_1, x : A; \Gamma$ and $P_2 \vdash \Delta_2, x : \overline{A}; \Gamma$ yields $\#P_1 > 1$ and $\#P_2 > 1$.

Then $\#(\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}) = \#P_1 \times \#P_2 > 1 \times 1 = 1$, which implies (ii).

Similarly for cases [Tsh], [TshL], [TshR].

Case [Tcut!]. We have $P = \mathbf{cut!}\ \{y.P_1\ |x : A|\ P_2\}$ where $P_1 \vdash y : A; \Gamma$ and $P_2 \vdash \Delta; \Gamma, x : \overline{A}$.

I.h applied to $P_1 \vdash y : A; \Gamma$ and $P_2 \vdash \Delta; \Gamma, x : \overline{A}$ yields $\#P_1 > 1$ and $\#P_2 \geq 1$.

Then

$$\#(\textbf{cut!}\ \{y.P_1\ |x|\ P_2\}) = \#P_1 \times \#P_2 \geq \#P_1 > 1$$

which implies (i).

Case: [Tsum]. We have $P = P_1 + P_2$ where $P_1 \vdash \Delta; \Gamma$ and $P_2 \vdash \Delta; \Gamma$.

Applying i.h. to $P_1 \vdash \Delta; \Gamma$ and $P_2 \vdash \Delta; \Gamma$ yields $\#P_1, \#P_2 \geq 1$.

Then $\#(P_1 + P_2) = \#P_1 + \#P_2 \geq 1 + 1 = 2 > 1$, which implies (i).

□

Observe that the measure # remains invariant under some structural congruence rules of $\approx$, namely the conversions that commute static figures, such as [CM]

$$\textbf{cut}\ \{P\ \ |x|\ \ (\textbf{par}\ \{Q\ ||\ R\})\} \equiv \textbf{par}\ \{(\textbf{cut}\ \{P\ \ |x|\ \ Q\})\ ||\ R\}$$

and remains also invariant by the conversions that distribute the static constructs over sum, such as [CSm]

$$\textbf{cut}\ \{P\ \ |x|\ \ (Q + R)\} \equiv (\textbf{cut}\ \{P\ \ |x|\ \ Q\}) + (\textbf{cut}\ \{P\ \ |x|\ \ R\})$$

Importantly, though, the $\equiv$ share commuting conversions [RSh], [TSh] and [PSh] and the conversions of Fig. 8.1 that commute a share with an action all yield a #-smaller process when applied from left to right. And all play an essential role in establishing the following lemma

Lemma 14 (Share Expansion)**.** *Let P and Q be processes of* CLASS$\backslash\exists\mu$ *s.t.* $Q \vdash \Delta', x : \boldsymbol{U}_f\ A; \Gamma$ *and either* $P \vdash \Delta, \boldsymbol{U}_e\ A; \Gamma$ *or* $P \vdash \Delta, x : \boldsymbol{U}_f\ A; \Gamma$*. Then, one of the following hypothesis hold*

(i) *There is a process R s.t.* $\textbf{share}\ x\ \{P\ ||\ Q\}\ \approx R$ *and* $\#(\textbf{share}\ x\ \{P\ ||\ Q\}) > \#R$.

(ii) *There are processes R, S s.t* $\textbf{share}\ x\ \{P\ ||\ Q\} \approx R + S$ *and* $\#(\textbf{share}\ x\ \{P\ ||\ Q\}) > \#R, \#S$.

(iii) *There is a process R s.t.* $\textbf{share}\ x\ \{P\ ||\ Q\}\ \approx \textbf{share}\ x\ \{\textbf{fwd}\ x\ y\ ||\ R\}$.

Proof. The proof is by induction on the sum of depths of the type derivation trees of P and Q. If P and Q are both take actions with subject x, then we apply [TSh] and (ii) holds. If either P or Q is either a put or release with subject x, when we apply either [PSh] or [RSh] and (i) holds. If either P and Q is a sum, then we apply [ShSm] and (ii) holds. If either P or Q is a forwarder, then (iii) holds. The remaining cases follow by commuting the share with the principal form of either P or Q and by applying the inductive hypothesis. Check Appendix D for details. □

Lemma 14 states that either we can expand a share into a #-smaller process (i) or a sum of #-smaller process (ii) by applying a series of $\approx$-conversions. This is possible if the usage is not being forwarded, in which case expansion is not possible (iii). This latter case gives, as we shall see in the proof of the Cut Normalisation Lemma 15, origin to open cells.

The proof of Cut Normalisation Lemma 15 is by lexicographical induction, the first component of this order being the type of the cut as ordered by the relation $\leq$, introduced in the following definition

Definition 28 ($A < B$)**.** *Let $\approx$ be the least congruence relation on types s.t.*

$$A \approx \overline{A} \quad \mathbf{S}_f\ A \approx \mathbf{S}_e\ A$$

$<$ is the least partial order on types that satisfies the following

$$A < A \otimes B \quad B < A \otimes B \quad A < A \oplus B \quad B < A \oplus B \quad A <!A \quad A < \wedge A \quad A < \mathbf{S}_f\ A \quad A < \mathbf{S}_e\ A$$
$$\wedge A < \mathbf{S}_f\ A \quad \frac{A \approx C \quad C < D \quad D \approx B}{A < B}$$

It is easy to check that $<$ has no infinite descending chain, since every descending chain starting with a type A has length at most $h(A) + 1$, where $h(A)$ is the height of the construction tree for type A. Crucially, we have $\wedge A < \mathbf{S}_f\ A$, since a cut on $\mathbf{S}_f\ A$ between a full reference cell and release or a take usage reduces to an expression that in which there is a cut on $\wedge A$, as expressed by the principal cut reductions $[\mathbf{S}_f\ \mathbf{U}_f\ r]$ and $[\mathbf{S}_f\ \mathbf{U}_f\ t]$.

The relation $<$ is preserved by $\approx$-equality both on the left and on the right. Dual types have the same $<$-order, as well as the state full and empty modalities. The latter because a cut on $\mathbf{S}_f\ A$ reduces to a cut on $\mathbf{S}_e\ A$ and vice-versa, as expressed by the principal cut reductions $[\mathbf{S}_f\ \mathbf{U}_f\ t]$ and $[\mathbf{S}_e\ \mathbf{U}_e\ ]$.

Normalisation of linear cuts depends on normalisation of unrestricted cuts, since a linear cut reduces to an unrestricted cut, as expressed by $\rightarrow$ rule [!?]. At the same time, normalisation of unrestricted cuts depends on normalisation of linear cuts, since the principal conversion [call] reduces an unrestricted cut by spawning a new linear cut. This knot is untied by establishing linear and unrestricted cut normalisation by mutual lexicographical induction as described in the proof of the following lemma

Lemma 15 (Cut Normalisation)**.** *The following two hypothesis hold*

$H_1(A, P, Q)$**:** *Suppose $P \vdash \Delta', x : A; \Gamma$, $Q \vdash \Delta, x : \overline{A}; \Gamma$ are normal processes of* $\mathsf{CLASS}\backslash\exists\mu$*. There exists a normal process R s.t.* **cut** $\{P \ |x : A|\ Q\} \approx R$*.*

$H_2(A, P, Q)$ *Suppose $P \vdash y : A; \Gamma$ and $Q \vdash \Delta; \Gamma, x : \overline{A}$ are normal processes of* $\mathsf{CLASS}\backslash\exists\mu$*. There exists a normal process R s.t.* **cut!** $\{y.P \ |x : A|\ Q\} \approx R$*.*

Proof. The proof is by mutual induction. More precisely, by lexicographical induction: first on A (ordered by $<$, Def. 28), then on the hypothesis being proved - we consider $H_1 < H_2$ - and finally on the #-measure of the cut.

Since $H_1 < H_2$, we can appeal to $H_1(A, P, Q)$ when proving $H_2(A, P, Q)$. To justify the inductive call we use the following notation

$$(A, H_i, P, Q) \rightarrow (B, H_j, R, S)$$

which means that we are proving hypothesis $H_i(A, P, Q)$ and that somewhere during the proof we appeal to hypothesis $H_j(B, R, S)$. The inductive call must respect the lexicographical order.

The overall proof strategy is to work the linear or the unrestricted cut by applying one of the rules $\approx$, so as to either expose a cut where we can make an inductive call, or to obtain a process in which there are not cuts except open cells. Lemma 14 is crucially used to normalise cuts between cells and shared usages on the same session.

We will now illustrate the proof of both hypothesis. Check Appendix D for details.

Proof of $H_1(A, P, Q)$**:** We perform case analysis on the root rules of derivation trees for $P \vdash \Delta', x : A; \Gamma$ and $Q \vdash \Delta, x : \overline{A}; \Gamma$. We illustrate with some cases.

Case: The root rule of P is [T!], the root rule of Q is [T?], both introduce an action with the same subject x.

We have

$$\frac{P' \vdash y : B; \Gamma}{!x(y); P' \vdash x :!B; \Gamma} \qquad \frac{Q' \vdash \Delta; \Gamma, x : \overline{B}}{?x; Q' \vdash \Delta, x :?\overline{B}; \Gamma}$$

where $P = !x(y); P'$, $\Delta' = \emptyset$ and $Q = ?x; Q'$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{!x(y); P' \ \ |x :!B|\ \ ?x; Q'\} \\
&\to \mathbf{cut!}\ \{y.P' \ |x : B|\ Q'\} && (\to [!?]) \\
&\approx R, \text{ for some normal } R && (\text{induction } (!B, H_1, -, -) \to (B, H_2, -, -))
\end{aligned}$$

Case The root rule of P is [Tcell], the root rule of Q is [Trelease], both introduce an action with the same subject x.

We have

$$\frac{P' \vdash \Delta', y : \wedge B; \Gamma}{\mathbf{cell}\ x(y.P') \ \vdash \Delta', x : \mathbf{S}_f\ B; \Gamma} \qquad \mathbf{release}\ x \vdash x : \mathbf{U}_f\ \overline{B}; \Gamma$$

where $P = \mathbf{cell}\ x(y.P')$, $Q = \mathbf{release}\ x$ and $\Delta = \emptyset$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{cell}\ x(y.P') \ \ |x : \mathbf{S}_f\ B|\ \ \mathbf{release}\ x\} \\
&\to \mathbf{cut}\ \{P \ \ |y : \wedge B|\ \ \mathbf{discard}\ y\} && (\to [\mathbf{S}_f\ \mathbf{U}_f\ \mathrm{r}]) \\
&\approx R, \text{ for some normal } R && (\text{induction } (\mathbf{S}_f\ B, -, -, -) \to (\wedge B, -, -, -))
\end{aligned}$$

Case: The root rule of P [Tcell] with subject x, the root rule of Q is [Tsh] on the usage x.

We have

$$\frac{P' \vdash \Delta, y : \wedge B; \Gamma}{\textbf{cell}\ x(y.P') \vdash x : \mathbf{S}_f\ B; \Gamma} \qquad \frac{Q_1 \vdash \Delta_1', x : \mathbf{U}_f\ \overline{B}; \Gamma \quad Q_2 \vdash \Delta_2', x : \mathbf{U}_f\ \overline{B}; \Gamma}{\textbf{share}\ x\ \{Q_1 \mid\mid Q_2\} \vdash \Delta_1', \Delta_2', x : \mathbf{U}_f\ \overline{B}; \Gamma}$$

where $P = \textbf{cell}\ x(y.P')$, $Q = \textbf{share}\ x\ \{Q_1 \mid\mid Q_2\}$, $\Delta' = \Delta_1', \Delta_2'$ and $A = \mathbf{S}_f\ B$.

By a applying Lemma 14 to hypothesis $Q_1 \vdash \Delta_1', x : \mathbf{U}_f\ B; \Gamma$ and $Q_2 \vdash \Delta_2', x : \mathbf{U}_f\ B; \Gamma$ we conclude that one of the following cases hold

Case (i): There is Q' s.t. $Q \approx Q'$ and $\#Q > \#Q'$.

Then

$$\begin{aligned} \textbf{cut}\ \{P\ \ |x : A|\ \ Q\} &\approx \textbf{cut}\ \{P\ \ |x : A|\ \ Q'\} \\ &\approx R, \text{ for some normal } R \\ &\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P, Q')) \end{aligned}$$

Case (ii): Exists R_1, R_2 s.t. $Q \approx R_1 + R_2$ and $\#Q > \#R_1, \#R_2$.

Then

$$\begin{aligned} \textbf{cut}\ \{P\ \ |x : A|\ \ Q\} &\approx \textbf{cut}\ \{P\ \ |x : A|\ \ (R_1 + R_2)\} \\ &\approx (\textbf{cut}\ \{P\ \ |x : A|\ \ R_1\}) + (\textbf{cut}\ \{P\ \ |x : A|\ \ R_2\}) \\ &\approx S_1 + (\textbf{cut}\ \{P\ \ |x : A|\ \ R_2\}), \text{ for some normal } S_2 \\ &\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P, R_1)) \\ &\approx S_1 + S_2, \text{ for some normal } S_2 \\ &\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P, R_2)) \end{aligned}$$

Case (iii): Exists Q' s.t. $Q \approx \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \mid\mid Q'\}$

Then

$$\textbf{cut}\ \{P\ \ |x : A|\ \ Q\} \approx \textbf{cut}\ \{\textbf{cell}\ x(y.P')\ \ |x : A|\ \ \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \mid\mid Q'\}\}$$

and $\textbf{cut}\ \{\textbf{cell}\ x(y.P')\ \ |x : A|\ \ \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \mid\mid Q'\}\}$ is an open cell.

Proof of H_2

We perform case analysis on the root rule of the tree for $Q \vdash \Delta; \Gamma, x : \overline{A}$. We illustrate with the principal unrestricted cut reduction.

Case The root rule if [Tcall] with subject x.

We have

$$\frac{Q' \vdash \Delta, z : \overline{A}; \Gamma, x : \overline{A}}{\textbf{call}\ x(z); Q' \vdash \Delta; \Gamma, x : \overline{A}}$$

where $Q = \textbf{call}\ x(z); Q'$.

Then

$$\textbf{cut!}\ \{y.P\ |x : A|\ \textbf{call}\ x(z); Q'\} \to \textbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ (\textbf{cut!}\ \{y.P\ |x : A|\ Q'\})\} \quad (\to [\text{call}])$$

$$\approx \textbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ R'\}, \text{ for some normal } R' \quad (\text{i.h. } (A, H_2, P, Q) \to (A, H_2, P, Q'))$$

$$\approx R, \text{ for some normal } R \quad (\text{i.h. } (A, H_2, -, -) \to (A, H_1, -, -))$$

□

Our main result for this chapter, the Cut Normalisation Theorem 4, follows immediately by the Cut Normalisation Lemma 15. In the next section we will present some corollaries of this main result.

Theorem 4 (Cut Normalisation)**.** *Fore every process $P \vdash \Delta; \Gamma$ there is a normal form Q s.t. $P \approx Q$.*

Proof. By straightforward induction on a type derivation tree for $P \vdash \Delta; \Gamma$ and by case analysis on the root rule. Cases [Tcut] and [Tcut!] follow by Lemma 15. □

8.3 Some Corollaries

In this section we collect some corollaries of the Cut Normalisation Theorem 4. First, we establish that every typed process has a type derivation that satisfies the subformula property (Corollary 1). In this section we consider all processes P to be from the subcalculus $\mathsf{CLASS}\backslash\exists\mu$ without quantifiers and inductive/coinductive types.

Then, we show that cuts can be eliminated from processes that type with a *pure* typing context (Corollary 2), i.e. with a typing context formed with only the types of Propositional Linear Logic, without the imperative state and usage modalities. Furthermore, we show that this target cut-free simpler form is *pure* itself, in the sense that it does not use imperative constructs, even if the source process manipulates shared state internally. Interestingly, this allows us to express the behaviour of a potentially complex imperative concurrent program as a sum of considerable simpler pure processes, as illustrated in Example 16

Before presenting our first corollary 2 we show first that every normal type derivation in $\mathsf{CLASS}\backslash\exists\mu$ satisfies the subformula property. But first let us introduce some definitions. We say that a type A is in a typing context $\Delta; \Gamma$ iff $x : A \in \Delta$ or $x : A \in \Gamma$, for some x. A type A is in a type derivation iff it is in a typing context of one of the sequents composing the derivation. In the following definition the order $\leq$ on types refers to Def. 28.

Definition 29 (Subformula Property)**.** *A type derivation for $P \vdash \Delta; \Gamma$ has the subformula property iff for all types A in the derivation there exists a type B in $\Delta; \Gamma$ s.t. $A \leq B$.*

Lemma 16 (Normal Forms Satisfy the Subformula Property)**.** *If P is normal then every type derivation for $P \vdash \Delta; \Gamma$ satisfies the subformula property.*

Proof. By induction on a derivation for $P \vdash \Delta; \Gamma$ and by case analysis on the root rule. The interesting case is [Tcut]. Since P is normal, the cut must be an open cell.

Suppose w.l.o.g that it is a full open cell. Then,

$$P = \mathbf{cut}\ \{\mathbf{cell}\ x(y.Q)\ \ |x : \mathbf{S}_f\ A|\ \ \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z\ ||\ R\}\}$$

and by inverting we conclude that any type derivation for $P \vdash \Delta; \Gamma$ has the following form

$$\dfrac{\dfrac{\dfrac{\vdots}{Q \vdash y : !A; \Gamma}}{\mathbf{cell}\ x(y.Q) \vdash x : \mathbf{S}_f\ A; \Gamma} \qquad \dfrac{\dfrac{}{\mathbf{fwd}\ x\ z\ \vdash x : \mathbf{U}_f\ A, z : \mathbf{S}_f\ A; \Gamma} \qquad \dfrac{\dfrac{\vdots}{R \vdash \Delta', x : \mathbf{U}_f\ \overline{A}; \Gamma}}{\mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z\ ||\ R\}\ \vdash \Delta', x : \mathbf{U}_f\ \overline{A}, z : \mathbf{S}_f\ A; \Gamma}}{\mathbf{cut}\ \{\mathbf{cell}\ x(y.Q)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ z\ ||\ R\}\} \vdash \Delta', z : \mathbf{S}_f\ A; \Gamma}$$

where $\Delta = \Delta', z : \mathbf{S}_f\ \overline{A}$. By induction on $Q \vdash y : \overline{A}; \Gamma$ and $R \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that derivations for $Q \vdash y : \overline{A}; \Gamma$ and $R \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ enjoy the subformula property, from which we can easily derive subformula property for the type derivation of $P \vdash \Delta; \Gamma$. Recall that $\leq$ is preserved by duality, hence $\mathbf{U}_f\ \overline{A} \leq \mathbf{S}_f\ A$ and that the inequality $\mathbf{affine}\ A; \leq \mathbf{S}_f\ A$ holds (Def. 28). □

We now present our first corollary.

Corollary 1 (Subformula Property)**.** *Suppose $P \vdash \Delta; \Gamma$. There is a type derivation $Q \vdash \Delta; \Gamma$ that satisfies the subformula property and $P \approx Q$.*

Proof. Follows by Cut Normalisation Theorem 4 and Lemma 16. □

We will now present the second corollary 2. But first, some definitions. We say that a process P is cut-free iff there is no process context C nor processes Q, R s.t. $P = C[\mathbf{cut}\ \{Q\ \ |x|\ \ R\}]$ or $P = C[\mathbf{cut!}\ \{y.Q\ |x|\ R\}]$, in other words if the process does not have either linear nor unrestricted cuts as subprocesses. We say that a type A is pure iff there is no type B s.t. either $\mathbf{S}_f\ B \leq A$ or $\mathbf{S}_e\ B \leq A$. A typing context is pure iff all the types in $\Delta; \Gamma$ are pure. A process P is pure iff there is no process context nor processes Q, R s.t. either $P = C[\mathbf{cell}\ x(y.Q)]$, $P = C[\mathbf{release}\ x]$, $P = C[\mathbf{take}\ x(y); Q]$, $P = C[\mathbf{put}\ x(y.Q); R]$ or $P = C[\mathbf{share}\ x\ \{Q\ ||\ R\}]$.

Now, we present our second corollary.

Corollary 2 (Cut Elimination for Pure Sequents)**.** *Suppose $P \vdash \Delta; \Gamma$, where $\Delta; \Gamma$ is a pure typing context. There exists a pure cut-free process $Q \vdash \Delta; \Gamma$ s.t. $P \approx Q$.*

Proof. By the Normalisation Theorem 4, there exists a normal process $Q \vdash \Delta; \Gamma$ s.t $P \approx Q$.

We show first that $Q \vdash \Delta; \Gamma$ is pure.

$\mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b; B(b))\ \ |c|\ \ \mathsf{toggle}(c, c')\}$
$\approx \mathbf{cell}\ c'(b'.\mathbf{affine}\ b'; (\mathbf{cut}\ \{B(b)\ \ |b|\ \ \mathsf{not}(b, b')\}))$ [cell-toggle]
$\mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b; B(b))\ \ |c|\ \ \mathsf{obs}(c, x, c')\}$
$\approx \mathbf{par}\ \{\mathbf{send}\ x(b.B(b)); \mathbf{close}\ x\ ||\ \mathbf{cell}\ c'(b'.\mathbf{affine}\ b'; \mathsf{false}(b'))\}$ [cell-obs]

$\mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b; B(b))\ \ |c|\ \ \mathsf{toggle}(c, c')\}$
$= \mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b; B(b))\ \ |c|\ \ \mathbf{take}\ c(b); \mathbf{put}\ c(b'.\mathbf{affine}\ b'; \mathbf{use}\ b; \mathsf{not}(b, b')); \mathbf{fwd}\ c\ c'\}$
(by def. of $\mathsf{toggle}(c, c')$)
$\approx \mathbf{cut}\ \{\mathbf{empty}\ c\ |c|\ \ \mathbf{affine}\ b; B(b)\ |b|\ \mathbf{put}\ c(b'.\mathbf{affine}\ b'; \mathbf{use}\ b; \mathsf{not}(b, b')); \mathbf{fwd}\ c\ c'\}$
(by $\rightarrow$ rule $[\mathbf{S}_f\ \mathbf{U}_f\ t]$)
$\approx \mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c|\ \ \mathbf{put}\ c(b'.\mathbf{affine}\ b'; \mathbf{cut}\ \{\mathbf{affine}\ b; B(b)\ \ |b|\ \ \mathbf{use}\ b; \mathsf{not}(b, b')\}); \mathbf{fwd}\ c\ c'\}$
(by $\approx$ rules [CPut] and [CAffine])
$\approx \mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c|\ \ \mathbf{put}\ c(b'.\mathbf{affine}\ b'; \mathbf{cut}\ \{B(b)\ \ |b'|\ \ \mathsf{not}(b, b')\}); \mathbf{fwd}\ c\ c'\}$
(by $\rightarrow$ $[\wedge\ \vee\ u]$)
$\approx \mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b'; \mathbf{cut}\ \{B(b)\ \ |b'|\ \ \mathsf{not}(b, b')\})\ \ |c|\ \ \mathbf{fwd}\ c\ c'\}$
(by $\rightarrow$ rule $[\mathbf{S}_e\ \mathbf{U}_e\ ]$)
$\approx \mathbf{cell}\ c'(b.\mathbf{affine}\ b'; \mathbf{cut}\ \{B(b)\ \ |b'|\ \ \mathsf{not}(b, b')\})$
(by $\rightarrow$ [fwd])

$\mathbf{cut}\ \{\mathbf{cell}\ c_0(b.\mathbf{affine}\ b; \mathsf{true}(b))\ \ |c_0|\ \ P\}$
(by def. of P)
$\approx \mathbf{cut}\ \{\ \mathbf{cell}\ c_0(b.\mathbf{affine}\ b; \mathsf{true}(b))\ |c_0|\ \ \mathsf{toggle}(c_0, c_1)\ |c_1|\ \mathsf{obs}(c_1, x, c_2)\ |c_2|\ \mathbf{release}\ c_2\}$
(by [cell-toggle])
$\approx \mathbf{cut}\ \{\mathbf{affine}\ b'; \mathbf{cut}\ \{\mathsf{true}(b)\ \ |b|\ \ \mathsf{not}(b, b')\})\ |c_1|\ \ \mathsf{obs}(c_1, x, c_2)\ |c_2|\ \mathbf{release}\ c_2\}$
(since $\mathbf{cut}\ \{\mathsf{true}(b)\ \ |b|\ \ \mathsf{not}(b, b')\} \xrightarrow{+} \mathsf{false}(b')$, see Example 4)
$\approx \mathbf{cut}\ \{\ \mathbf{cell}\ c_1(b'.\mathbf{affine}\ b'; \mathsf{false}(b'))\ \mathbf{cut}c_1\ \ \mathsf{obs}(c_1, x, c_2)\ \mathbf{cut}c_2\ \mathbf{release}\ c_2\}$
(by [cell-obs])
$\approx \mathbf{par}\ \{\mathbf{send}\ x(b.\mathsf{false}(b)); \mathbf{close}\ x\ ||\ \mathbf{cut}\ \{\mathbf{cell}\ c_2(b.\mathbf{affine}\ b; \mathsf{false}(b))\ \ |c_2|\ \ \mathbf{release}\ c_2\}\}$
(by $\rightarrow$ rules $[\mathbf{S}_f\ \mathbf{U}_f\ r]$ and $[\wedge\ \vee\ d]$)
$\approx \mathbf{par}\ \{\mathbf{send}\ x(b.\mathsf{false}(b)); \mathbf{close}\ x\ ||\ \mathbf{0}\}$
(by $\equiv$ unit rule [0M])
$\approx \mathbf{send}\ x(b.\mathsf{false}(b)); \mathbf{close}\ x$

Figure 8.5: Cut elimination: example.

For suppose, in order to derive a contradiction, that Q has an imperative construct, then it uses one of the typing rules [Tcell], [Tempty] [Tfree], [Ttake], [Tput], [Tsh], [TshL] or [TshR], which necessarily introduce either a $\mathbf{S}_f\ A$ or a $\mathbf{U}_f\ A$ modality.

But then, by the Subformula Property 1, we conclude that there should exist a type B in $\Delta; \Gamma$ for which $\mathbf{S}_f\ A \leq B$, which contradicts the fact that $\Delta; \Gamma$ is a pure typing context.

Now, we show that Q is cut-free. For suppose that Q has a cut, since it is a normal process the cut must be an open cell $\mathbf{cut}\ \{\mathbf{cell}\ y(-.-)\ |y|\ \mathbf{share}\ y\ \{-\ ||\ -\}\}$, but then it implies that Q has imperative constructs and, hence is not pure, which is a contradiction. □

Example 16. *In this example we apply Corollary 2 and show to derive, by doing simple algebraic-like $\approx$-manipulations, a sum of pure processes that summarises the behaviour of a stateful program.*

Consider a stateful process

$$\mathsf{system}(x) \vdash x : \mathsf{Bool} \otimes \mathbf{1}$$

defined by

$$\begin{array}{l}\mathsf{system}(x) \triangleq \\ \textbf{cut}\ \{\textbf{cell}\ c_0(b.\textbf{affine}\ b; \mathsf{true}(b))\ \ |c_0| \\ \textbf{share}\ c_0\ \{\textbf{cut}\ \{\mathsf{toggle}(c_0, c_1)\ \ |c_1|\ \ \textbf{release}\ c_1\}\ ||\ \textbf{cut}\ \{\mathsf{obs}(c_0, x, c_1)\ \ |c_1|\ \ \textbf{release}\ c_1\}\}\}\end{array}$$

where

$$\begin{array}{rcl}\mathsf{toggle}(c, c') & \triangleq & \textbf{take}\ c(b); \textbf{put}\ c(b.\textbf{affine}\ b'; \textbf{use}\ b; \mathsf{not}(b, b')); \textbf{fwd}\ c\ c' \\ \mathsf{obs}(c, x, c') & \triangleq & \textbf{take}\ c(b); \textbf{par}\ \{\textbf{use}\ b; \textbf{send}\ x(b); \textbf{close}\ x\ || \\ & & \quad \textbf{put}\ c(b'.\textbf{affine}\ b'; \mathsf{false}(b')); \textbf{fwd}\ c\ c'\}\end{array}$$

Process $\mathsf{system}(x)$ *composes a reference boolean cell* $c_0 : \boldsymbol{S}_f\ \mathsf{Bool}$*, initially storing the boolean* true*, with two atomic actions: one that toggles the cell state and another that observes. The booleans and their basic operations were previously defined in Example 4.*

The toggle action is defined by composing, via a cut on c_1*, process* $\mathsf{toggle}(c_0, c_1)$ *with* **release** c_1*. Process*

$$\mathsf{toggle}(c, c') \vdash c : \boldsymbol{U}_f\ \overline{\mathsf{Bool}}, \boldsymbol{S}_f\ \mathsf{Bool}$$

updates the reference cell with its negated boolean and forwards its updated state to c'*.*

Likewise, the observation operation is defined by composing $\mathsf{obs}(c_0, x, c_1)$ *with* **release** c_1*. Process*

$$\mathsf{obs}(c, x, c') \vdash c : \boldsymbol{U}_f\ \overline{\mathsf{Bool}}, x : \mathsf{Bool} \otimes \mathbf{1}, c' : \boldsymbol{S}_f\ \mathsf{Bool}$$

observes cell c *by sending the stored boolean on a session* x*, after which* x *is closed, then it resets the cell* c *to* false *and forwards the updated cell to* c'*.*

Process $\mathsf{system}(x)$ *internally manipulates a reference cell, but since it types with a pure typing context, by Corollary 2, there must exist a cut-free pure process* $\mathsf{system}'(x) \vdash x : \mathsf{Bool} \otimes \mathbf{1}$ *such that* $\mathsf{system}(x) \approx \mathsf{system}'(x)$*.*

Indeed, let

$$\mathsf{system}'(x) \triangleq \textbf{send}\ x(b.\textbf{affine}\ b; \mathsf{false}(b)); \textbf{close}\ x + \textbf{send}\ x(b.\textbf{affine}\ b; \mathsf{true}(b)); \textbf{close}\ x$$

The cut-free pure process $\mathsf{system}'(x) \vdash x : \mathsf{Bool} \otimes \mathbf{1}$ *summarises the behaviour of* $\mathsf{system}(x)$ *as a sum of pure process that send either the boolean* false *or* true*, depending on the nondeterministic scheduling of the two concurrent atomic actions toggle and observe.*

We will now show how to compute $\mathsf{system}'(x)$ *by doing simple algebraic manipulations, as expressed by the complete set of commuting conversions* $\approx$ *(Def. 26).*

First, we will start by expanding the share of usage c_0 *of* $\mathsf{system}(x)$ *into a sum of sequential usages*

$$\textbf{share}\ c_0\ \{\textbf{cut}\ \{\mathsf{toggle}(c_0, c_1)\ \ |c_1|\ \ \textbf{release}\ c_1\}\ ||\ \textbf{cut}\ \{\mathsf{obs}(c_0, x, c_1)\ \ |c_1|\ \ \textbf{release}\ c_1\}\} \approx P + Q$$

where

$$\begin{array}{lcl} P & \triangleq & \mathbf{cut}\ \{(\mathbf{cut}\ \{\mathsf{toggle}(c_0,c_1) \quad |c_1|\ \ \mathsf{obs}(c_1,x,c_2)\}) \quad |c_2|\ \ \mathbf{release}\ c_2\} \\ Q & \triangleq & \mathbf{cut}\ \{(\mathbf{cut}\ \{\mathsf{obs}(c_0,x,c_1) \quad |c_1|\ \ \mathsf{toggle}(c_1,c_2)\}) \quad |c_2|\ \ \mathbf{release}\ c_2\} \end{array}$$

The sum is exhibited by first applying $\approx$ law [TSh] that interleaves the two concurrent take actions. Then, we apply further laws of $\approx$ to each summand that allows us to push the structure inside the share constructs outside, like for example $\equiv$ rule [PSh] and $\approx$ rule [ShUse]. Finally, by applying the identity $\equiv$ law [RSh] $\mathbf{share}\ x\ \{\mathbf{release}\ x\ ||\ R\}\ \equiv R$ we get rid of the share construct, obtaining the sequential cell usages defined by P and Q.

Process P corresponds to the cell usage scheduling where we first toggle the cell and only then do the observation, whereas Q corresponds to the scheduling in which the observation is done before the toggle.

Applying $\equiv$ law [CSm] allows us to distribute the cell over each summand

$$\begin{array}{lcl} \mathsf{system}(x) & \approx & \mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b;\mathsf{true}(b))\ \ |c|\ \ (P+Q)\} \\ & \approx & (\mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b;\mathsf{true}(b))\ \ |c|\ \ P\}) + (\mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b;\mathsf{true}(b))\ \ |c|\ \ Q\}) \end{array}$$

and then cut elimination proceeds independently for each summand.

The following auxiliary $\approx$-equivalences

$$\begin{array}{ll} \mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b;B(b))\ \ |c|\ \ \mathsf{toggle}(c,c')\} & \\ \approx \mathbf{cell}\ c'(b'.\mathbf{affine}\ b';(\mathbf{cut}\ \{B(b)\ \ |b|\ \ \mathsf{not}(b,b')\})) & \textit{[cell-toggle]} \\ \mathbf{cut}\ \{\mathbf{cell}\ c(b.\mathbf{affine}\ b;B(b))\ \ |c|\ \ \mathsf{obs}(c,x,c')\} & \\ \approx \mathbf{par}\ \{\mathbf{send}\ x(b.B(b));\mathbf{close}\ x\ ||\ \mathbf{cell}\ c'(b'.\mathbf{affine}\ b';\mathsf{false}(b'))\} & \textit{[cell-obs]} \end{array}$$

allows us to compute the result of the interaction of a cell c storing an arbitrary boolean $B(b)$ with processes $\mathsf{toggle}(c,c')$ and $\mathsf{obs}(c,x,c')$, respectively.

Fig. 8.5 shows how to derive step-by-step law [cell-inc], law [cell-obs] can be derived in a similar way. It shows $\approx$-equalities [cell-toggle] and [cell-obs] for interaction between a natural reference cell and the imperative atomic toggle and observe operations. The equalities are presented on top. Then, we show how to derive step-by-step [cell-toggle]. We also show how the $\approx$-equalities [cell-toggle] and [cell-obs] are then used to simplify a stateful process, obtaining a pure one.

Then, we obtain

$$\mathbf{cut}\ \{\mathbf{cell}\ c_0(b.\mathbf{affine}\ b;\mathsf{true}(b))\ \ |c_0|\ \ P\} \approx \mathbf{send}\ x(b.\mathsf{false}(b));\mathbf{close}\ x$$

the derivation of which is also displayed in Fig. 8.5. Similarly one may derive

$$\mathbf{cut}\ \{\mathbf{cell}\ c_0(b.\mathbf{affine}\ b;\mathsf{true}(b))\ \ |c_0|\ \ Q\} \approx \mathbf{send}\ x(b.\mathsf{true}(b));\mathbf{close}\ x$$

and this concludes the derivation of the simplified cut-free pure process $\mathsf{system}'(x)$.

8.4 Further Discussion and Related Work

Our proof of the Cut Normalisation Lemma 15 is inspired by [122, 123], where Pfenning presents a proof of cut elimination for linear logic, based on the work of Hodas [70]. The structural proof of Pfenning is given for a dyadic sequent calculus formulation of linear logic, called LV, that works by isolating the non-linear reasoning into an unrestricted typing context (like in CLASS), and thereby circumvents some difficulties of establishing cut elimination for monadic sequent formulations of linear logic in the presence of contraction. Pfenning proves linear and unrestricted cut admissibility for LV by a mutual lexicographical induction. The novelty in the proof of Lemma 15 lies in the additional step for expanding share operations by applying a series of $\approx$-transformations (Share Expansion Lemma).

Another result that is usually proved alongside cut elimination is identity elimination, i.e. that each provable type judgment has a proof that does not use the identity axiom [Tfwd]. The result holds for classical linear logic and the collection of necessary proof transformations for identity elimination are usually referred to as *identity expansions* (see, for example, [24]).

For example, if we consider the following type-annotated version of the forwarding construct $\mathbf{fwd}_A\ x\ y \vdash x : A, y : \overline{A}; \Gamma$, then the identity expansions for the unit $\mathbf{1}$ and the tensor $\otimes$ are written as

$$\begin{array}{lcl} \mathbf{fwd}_{\mathbf{1}}\ x\ y & \approx & \mathbf{wait}\ y; \mathbf{close}\ x \\ \mathbf{fwd}_{A \otimes B}\ x\ y & \approx & \mathbf{recv}\ y(w); \mathbf{send}\ x(z.\mathbf{fwd}_A\ z\ w); \mathbf{fwd}_B\ x\ y \end{array}$$

Given the infinite possible set of behaviours of a cell usage and the complexity layer added by cell sharing, it is not clear at the moment if it is possible to express identity expansions for the state modalities.

The technique used in this chapter to prove cut normalisation relies on a well-founded inductive measure on the cut formula and, therefore, does not scale for more complex type constructs such as polymorphic and inductive types. In the following chapter we introduce a new technique, based on linear logical relations, that allows us to prove a normalisation result for the complete language CLASS.

9

Strong Normalisation

9.1 Introduction

In this chapter we prove a strong normalisation result. Let $\mathcal{R}$ be a binary relation on processes. A process P_0 is strongly normalising (SN) w.r.t. $\mathcal{R}$ iff there is no infinite sequence $P_0, P_1, P_2, \ldots$ starting with P_0 s.t. $(P_i, P_{i+1}) \in \mathcal{R}$, for all $0 \leq i$. The relation $\mathcal{R}$ is said to be strongly normalising if all the processes P are strongly normalising w.r.t $\mathcal{R}$.

Strong normalisation of reduction is key for critical imperative concurrent code since it guarantees that processes are responsive participants which always react within a finite amount of time. For example: when a client requests a service, the computation on the server side is always guaranteed to terminate and to return an answer. The client will not be left hanging out forever. And even if at the global level we may want to write programs that do not terminate (such as operating systems) we still want to guarantee that this non-terminating behaviour does not result from internal activity. Furthermore, basic operations of concurrent datatypes must always terminate.

Strong normalisation is a result quite challenging to obtain, particularly in the the context of concurrent process calculi [46]. Furthermore, it is a property which can be easily compromised in the presence of recursion and shared mutable state. For example, with higher-order state and without further restrictions, we can express non-terminating programs such as the famous Landin's Knot. However, our linear logic based type system rules out cyclic dependencies in memory, additionally our recursion is well-founded since the type system excludes infinite chains of nested corecursive calls.

Our proof of strong normalisation relies on the technique of linear logical relations for session-based concurrent processes [121, 32, 146], here adapted to classical linear logic where we explore the benefits of having a duality relation for session types [63, 9, 1]. We extend the technique with auxiliary constructs **cell** $c(a.S)$ and **empty** $c(a.S)$, which are cells subject to interference over a set S. They crucially allows us to reason about state sharing in a compositional way and the argument to go through. The intuition is that a take on an interference-sensitive cell might obtain a session distinct from the session previously put, even if the cell is not being shared.

In CLASS, reduction $\longrightarrow$ is not SN, essentially due to the idempotency of the sum constructor ($P \equiv P + P$). For suppose that $P \longrightarrow P'$, then we can form the infinite reduction sequence

$$P \equiv P + P \longrightarrow P' + P \equiv P' + (P + P) \longrightarrow P' + (P' + P) \equiv (P' + P') + P \equiv P' + P \longrightarrow \ldots$$

Nevertheless, there is a sum-free subcalculus of CLASS for which strong normalisation holds. More precisely, we say that a process P in CLASS is said to be sum-free iff there are no processes Q, R in CLASS s.t. $Q + R$ is a subprocess of P. In this chapter we consider all processes to be sum-free.

In this chapter, we work with a congruence relation $\equiv_c$ that extends structural congruence $\equiv$, restricted to sum-free processes, with a complete set of commuting conversions ,along standard lines [23, 24, 157], that allows us commute actions with static constructs, for example:

$$\begin{array}{lcl}
\textbf{par}\ \{\textbf{wait}\ x; P \parallel Q\} & \approx & \textbf{wait}\ x; \textbf{par}\ \{P \parallel Q\} \\
\textbf{cut}\ \{\textbf{wait}\ x; P \ \ |y|\ \ Q\} & \approx & \textbf{wait}\ x; \textbf{cut}\ \{P \ \ |y|\ \ Q\},\ y \neq x \\
\textbf{share}\ y\ \{\textbf{wait}\ x; P \parallel Q\} & \approx & \textbf{wait}\ x; \textbf{share}\ y\ \{P \parallel Q\}
\end{array}$$

and that satisfies rule [TShC]

$$\textbf{share}\ x\ \{\textbf{take}\ x(y_1); P_1 \parallel \textbf{take}\ x(y_2); P_2\} \equiv_c \textbf{take}\ x(y_1); \textbf{share}\ x\ \{P_1 \parallel \textbf{take}\ x(y_2); P_2\}$$

which allows us to nondeterministically choose a take operation to have precedence and interact with a reference cell. In rule [TShC] we pick the take operation that occurs in the left argument of the share construct, however since share is commutative we can pick the take operation that occurs in the right argument instead. The collapsing reduction relation $\longrightarrow_c$ is a static congruence that operates on sum-free processes, it satisfies the same principal cut reductions as reduction $\longrightarrow$ in CLASS but it is closed by the congruence relation $\equiv_c$ instead. Complete definitions are given in Appendix E. Hereafter, strong normalisation is always with respect to $\longrightarrow_c$.

The chapter is organised as follows. In Section 9.2 we extend CLASS with the interference-sensitive reference cells and proves some simulation properties, among which is the key simulation given by Lemma 20 which allows us to reason about state sharing compositionally. Then, in Section 9.3 we introduce the linear logical predicates and conclude with the proof of the Fundamental Lemma 27, which states that every well-typed process is in the corresponding logical predicate. This lemma immediately implies our strong normalisation result (Theorem 5). Finally, we conclude this chapter in Section 9.4 with further discussion and related work.

9.2 Interference-Sensitive Cells

In this section we equip CLASS with interference-sensitive cells, reference cells which internalise state interference, resultant from shared usage manipulation, in their operational model. These auxiliary process constructs play a crucial technical role in the proof of the strong normalisation result, essentially because they allow us to reason about state interference compositionally, as expressed by Lemma 20. We start with the definition of interference-sensitive cells.

Definition 30 (Interference-Sensitive Cells)**.** *Let $S \subseteq \{R \mid R \vdash_\eta a : \wedge A\}$. We extend the process calculus* CLASS *with the interference-sensitive full* **cell** $c(a.S)$ *and empty* **empty** $c(a.S)$ *cells, which have following associated principal reduction rules*

$$\textbf{cut}\ \{\textbf{cell}\ c(a.S)\ \ |c|\ \ \textbf{release}\ c\} \rightarrow_{\mathsf{c}} \textbf{cut}\ \{P\ \ |a|\ \ \textbf{discard}\ a\},\ P \in S\ (1)$$
$$\textbf{cut}\ \{\textbf{cell}\ c(a.S)\ \ |c|\ \ \textbf{take}\ c(a'); Q\} \rightarrow_{\mathsf{c}} \textbf{cut}\ \{\textbf{empty}\ c(a.S)\ \ |c|\ \ (\textbf{cut}\ \{P\ \ |a|\ \ \{a/a'\}Q\})\},\ P \in S\ (2)$$
$$\textbf{cut}\ \{\textbf{empty}\ c(a.S)\ \ |c|\ \ \textbf{put}\ c(a.Q_1); Q_2\} \rightarrow_{\mathsf{c}} \textbf{cut}\ \{\textbf{cell}\ c(a.S)\ \ |c|\ \ Q_2\}\ (3)$$

Rules (1) and (2) apply to usage processes $P \vdash c : \mathbf{U}_f\ A$, whereas rule (3) applies to a usage process $P \vdash c : \mathbf{U}_e\ A$. When a take or a release action interacts with an interference-sensitive full cell **cell** $c(a.S)$ we pick an arbitrary element P from the set S (rules (1) and (2)). On the other hand, when a put action **put** $c(a.Q_1); Q_2$ interacts with an interference-sensitive empty cell **empty** $c(a.S)$ it evolves to **cell** $c(a.S)$ (3).

The process constructs **cell** $c(a.S)$ and **empty** $c(a.S)$ can be though of as reference cells subject to interference over the set S. They contrast wit the the basic empty and full reference cells **cell** $c(a.P)$ and **empty** c of CLASS which are, so to speak, blind to the interference that results from concurrency, since from a local point of view they obey a sequential protocol: if a cell is not being shared by any other thread then every take acquires the session that was put before or that was present in the cell initially. On the other hand, a take on an interference-sensitive cell might obtain a session distinct from the session previously put, even if the interference-sensitive cell is not being explicitly shared. So, interference resulting from cell sharing is baked in the operational semantics of the interference-sensitive cells as expressed by rules (1)-(3) of Def. 30.

Provided the usages are well-behaved according to to the set over which the interference-sensitive cells are defined, as formalised by coinductive Def. 31, it is possible to simulate the basic full and empty cells of CLASS with interference-sensitive cells, as described by Lemma 18.

Definition 31. *Let $S \subseteq \{R \mid R \vdash y : \wedge\overline{A}\}$. A process P, where either $P \vdash x : \mathbf{U}_f\ A$ or $P \vdash x : \mathbf{U}_e\ A$, is S-preserving on x iff the following hold*

(a) *If $P \xrightarrow{*}_{\mathsf{c}} Q$, $Q \equiv_{\mathsf{c}} \textbf{take}\ x(y'); Q'$ and $R \in S$, then* **cut** $\{\{y'/y\}R\ \ |y'|\ \ Q'\}$ *is S-preserving on x.*

(b) *If $P \xrightarrow{*}_{\mathsf{c}} Q$ and $Q \equiv_{\mathsf{c}} \textbf{put}\ x(y'.Q_1); Q_2$, then $\{y/y'\}Q_1 \in S$ and Q_2 is S-preserving on x.*

If a process P is S-preserving on x and after some internal reductions it offers a take action, then the continuation of the take action composed with an element from S is also

S-preserving on x (Def. 31(a)). Symmetrically, if P offers a put action then the element put is on the set S and the continuation is still S-preserving (Def. 31(b)). A set of processes T is S-preserving on x if and only for all $P \in T$, P is S-preserving on x.

Intuitively a process P that uses a cell x is S-preserving on x if it only puts values from S on cell x. The notion of S-preservation, parametric on any S, brings explicit the conditions needed for safe interaction with a memory cell, subject to interference, while ensuring a state invariant S on the cell contents.

The notion of S-preserving is preserved by reduction $\overset{*}{\rightarrow}_c$, as expressed by the following lemma.

Lemma 17. *If P is S-preserving on x and $P \overset{*}{\rightarrow}_c Q$, then Q is S-preserving on x.*

Proof. Immediate from Def. 31. □

The following result sufficient conditions for simulating be basic reference cells using the interference-sensitive cells. But before we need to introduce the notion of simulation. A simulation $\mathcal{S}$ is a binary relation on processes s.t. whenever $(P, Q) \in \mathcal{S}$ and $P \rightarrow_{\mathsf{c}} P'$ then there exists Q' s.t. $Q \overset{+}{\rightarrow}_{\mathsf{c}} Q'$ and $(P', Q') \in \mathcal{S}$. We say that P simulates Q iff there exists a simulation $\mathcal{S}$ s.t. $(Q, P) \in \mathcal{S}$.

Lemma 18. *The following properties hold*

(1) *Let $S \subseteq \{R \mid R \vdash_\eta y : \wedge A\}$, $P \in S$, $Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}$ and suppose Q is S-preserving on x. Then, $\mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\}$.*

(2) *Let $S \subseteq \{R \mid R \vdash_\eta y : \wedge A\}$, $Q \vdash_\eta x : \mathbf{U}_e\ \overline{A}$ and Q suppose Q is S-preserving on x. Then, $\mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ Q\}$.*

Proof. Define

$$\mathcal{S} \triangleq \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$$

where

$$\begin{array}{rcl}
\mathcal{S}_1 & \triangleq & \{(M,N) \mid \exists P \in S, \exists Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}.\ Q \text{ is } S\text{-preserving on } x \text{ and} \\
 & & M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q\} \text{ and } N \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\}\} \\
\mathcal{S}_2 & \triangleq & \{(M,N) \mid \exists Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}.\ Q \text{ is } S\text{-preserving on } x \text{ and} \\
 & & M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ Q\} \text{ and } N \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ Q\}\} \\
\mathcal{S}_3 & \triangleq & \{(M,N) \mid M \equiv_{\mathsf{c}} N\}
\end{array}$$

We prove that $\mathcal{S}$ is a simulation, by performing case analysis first on $(M, N) \in \mathcal{S}$ and then on $M \rightarrow_{\mathsf{c}} M'$. Complete proof in Appendix E. □

Crucially, the notion of S-preserving is preserved by concurrent share composition as described by the following lemma

Lemma 19. *If P and Q are S-preserving on x, then $\mathbf{share}\ x\ \{P \parallel Q\}$ is S-preserving on x.*

Proof. By coinduction. We need to prove that **share** $x\ \{P \mid\mid Q\}$ satisfies (a)-(b) of Def. 31.

(a) Let $R \in S$ and suppose **share** $x\ \{P \mid\mid Q\} \xrightarrow{*}_c \equiv_{\mathsf{c}}$ **take** $x(y); M$.

The take on x comes either from P or Q. Suppose w.l.o.g. that it comes from P. Then

$$P \xrightarrow{*}_c \equiv_{\mathsf{c}} \textbf{take}\ x(y); P' \text{ and } M \equiv_{\mathsf{c}} \textbf{share}\ x\ \{P' \mid\mid Q'\}$$

where $Q \xrightarrow{*}_c Q'$.

We need to prove that **cut** $\{R\ \ |y|\ \ M\}$ is S-preserving on x.

But

$$\textbf{cut}\ \{R\ \ |y|\ \ M\} \equiv_{\mathsf{c}} \textbf{cut}\ \{R\ \ |y|\ \ \textbf{share}\ x\ \{P' \mid\mid Q'\}\} \equiv_{\mathsf{c}} \textbf{share}\ x\ \{\textbf{cut}\ \{R\ \ |y|\ \ P'\} \mid\mid Q'\}$$

Since P is S-preserving on x and $R \in S$, then Def. 31(a) implies that **cut** $\{R\ \ |y|\ \ P'\}$ is S-preserving on x.

Since Q is S-preserving on x and $Q \xrightarrow{*}_c Q'$, then Q' is S-preserving on x (by Lemma 17).

By coinductive hypothesis we conclude that **share** $x\ \{$**cut** $\{R\ \ |y|\ \ P'\} \mid\mid Q'\}$ is S-preserving on x.

(b) If $P \xrightarrow{*}_c Q$ and $Q \equiv_{\mathsf{c}}$ **put** $x(y.Q_1); Q_2$, then $Q_1 \in S$ and Q_2 is S-preserving on x.

Suppose **share** $x\ \{P \mid\mid Q\} \xrightarrow{*}_c \equiv_{\mathsf{c}}$ **put** $x(y.M_1); M_2$.

Suppose w.l.o.g. that $P \vdash x : \mathbf{U}_e\ A$, then the put comes from P.

Hence

$$P \xrightarrow{*}_c \equiv_{\mathsf{c}} \textbf{put}\ x(y.M_1); P' \text{ and } M \equiv_{\mathsf{c}} \textbf{share}\ x\ \{P' \mid\mid Q'\}$$

where $Q \xrightarrow{*}_c Q'$.

We need to prove that (i) $M_1 \in S$ and that (ii) **share** $x\ \{P' \mid\mid Q'\}$ is S-preserving on x.

(i) follows since P is S-preserving on x (Def. 31(b)).

Since P is S-preserving on x (Def. 31(b)), then P' is S-preserving.

Since Q is S-preserving on x and $Q \xrightarrow{*}_c Q'$, then Q' is S-preserving on x (by Lemma 17).

By coinductive hypothesis, **share** $x\ \{P' \mid\mid Q'\}$ is S-preserving on x, hence (ii).

□

Since the potential interference resulting from cell sharing is absorbed by the operational semantics that characterises the interference-sensitive cells (Def. 30), we have the following simulation property which allows us to reason compositionally about state sharing, and with which we conclude this section.

Lemma 20. *The following pair of simulations hold*

(1) *Let* $P \vdash_\eta x : \mathbf{U}_f\ A$, $Q \vdash_\eta x : \mathbf{U}_f\ A$ *and* $S \subseteq \{R \mid R \vdash_\eta y : \wedge\overline{A}\}$. *Then,*

$$\begin{array}{c}\textbf{par}\ \{(\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\} \\ \textit{simulates} \\ \textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ \textbf{share}\ x\ \{P\ ||\ Q\}\}\end{array}$$

(2) *Let* $P \vdash_\eta x : \mathbf{U}_e\ A$, $Q \vdash_\eta x : \mathbf{U}_f\ A$ *and* $S \subseteq \{R \mid R \vdash_\eta y : \wedge\overline{A}\}$. *Then,*

$$\begin{array}{c}\textbf{par}\ \{(\textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\} \\ \textit{simulates} \\ \textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ \textbf{share}\ x\ \{P\ ||\ Q\}\}\end{array}$$

Proof. Define

$$\mathcal{S} \triangleq \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$$

where

$$\begin{array}{rcl}\mathcal{S}_1 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta x : \mathbf{U}_f\ A, \exists Q \vdash_\eta x : \mathbf{U}_f\ A.\ M \equiv_{\mathsf{c}} \textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ \textbf{share}\ x\ \{P\ ||\ Q\}\} \\ & & \text{and } N \equiv_{\mathsf{c}} \textbf{par}\ \{(\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}\} \\ \mathcal{S}_2 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta x : \mathbf{U}_e\ A, \exists Q \vdash_\eta x : \mathbf{U}_f\ A.\ M \equiv_{\mathsf{c}} \textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ \textbf{share}\ x\ \{P\ ||\ Q\}\} \\ & & \text{and } N \equiv_{\mathsf{c}} \textbf{par}\ \{(\textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}\} \\ \mathcal{S}_3 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta \emptyset;\emptyset, \exists \mathcal{C} \exists \mathcal{D}.\ M \equiv_{\mathsf{c}} \mathcal{C} \circ \mathcal{D}[P] \text{ and } N \equiv_{\mathsf{c}} \textbf{par}\ \{\mathcal{C}[P]\ ||\ \mathcal{D}[P]\}\}\end{array}$$

We prove that $\mathcal{S}$ is a simulation, by performing case analysis first on $(M,N) \in \mathcal{S}$ and then on $M \to_{\mathsf{c}} M'$. Complete proof in Appendix E. □

9.3 Linear Logical Predicates for Strong Normalisation

The goal of this section is to introduce the linear logical predicates, used to establish our strong normalisation result. In 9.3, we start by presenting some basic properties about SN processes and then we introduce the orthogonal operation. This operation is then used to define, later in 9.3, our basic logical predicates $[\![x : A]\!]_\sigma$, we then prove some properties. We conclude in 9.3 with the proof of the Fundamental Lemma 27, from which our strong normalisation result follows immediately (Theorem 5).

Orthogonal and Basic Properties

We start by stating some basic properties (Lemma 21) but first let us introduce a measure on SN processes, which will be often used to prove properties about strong normalisation by induction. For every process P there is a finite (up to $\equiv_{\mathsf{c}}$) number of processes Q for which $P \to_{\mathsf{c}} Q$. Hence, By König's Lemma, for each SN process P there is a longest $\to_{\mathsf{c}}$-reduction sequence starting with P, we denote the length of this sequence by $N(P)$.

Lemma 21 (SN: Basic Properties)**.** *The following properties hold*

(1) *If P is SN and $P \equiv_c Q$, then Q is SN.*

(2) *If P is SN and $P \to_c Q$, then Q is SN.*

(3) *Suppose Q is SN whenever $P \to_c Q$. Then, P is SN.*

(4) *If P and Q are SN, then* **par** $\{P \mid\mid Q\}$ *is SN.*

(5) *If Q is SN and Q simulates P, then P is SN.*

Proof. All properties are easy to establish, in particular we have the following: in (1) $N(P) = N(Q)$, in (2) $N(Q) = N(P) - 1$, in (3) $N(P) = (\mathsf{max}\ \{Q \mid P \to_c Q\}) + 1$ and in (4) $N(\textbf{par}\ \{P \mid\mid Q\}) = N(P) + N(Q)$. □

We will now introduce the orthogonal, which will play a key role when defining logical predicates for strong normalisation. As we will see, each logical predicate is defined by taking the orthogonal of some set. In the following, we write P_x to emphasise that x is the only free name of P.

Definition 32 (Orthogonal $(-)^\perp$)**.** *Let S be a subset of processes Q_x with a single free name x. We define the orthogonal of S, written $S^\perp$, by*

$$S^\perp \triangleq \{P_x \mid \forall Q_x \in S.\ \textbf{cut}\ \{P_x \ \ |x|\ \ Q_x\}\ \textit{is SN}\}$$

The orthogonal satisfies some well-known properties, as stated by the following lemma.

Lemma 22 (Orthogonal: Basic Properties)**.** *The following properties hold*

(1) *If $P \in S^\perp$ and $P \equiv_c Q$, then $Q \in S^\perp$.*

(2) *If $P \in S^\perp$ and $P \to_c Q$,then $Q \in S^\perp$.*

(3) *If $S_1 \subseteq S_2$, then $S_2^\perp \subseteq S_1^\perp$*

(4) *$S \subseteq S^{\perp\perp}$.*

(5) *$S^{\perp\perp\perp} = S^\perp$*

(6) *Let $\mathcal{S}$ be a collection of sets. Then, $(\bigcup \mathcal{S})^\perp = \bigcap_{S \in \mathcal{S}} S^\perp$.*

(7) *Let $\mathcal{S}$ be a collection of sets S s.t. $S = S^{\perp\perp}$, whenever $S \in \mathcal{S}$. Then, $(\bigcap \mathcal{S})^{\perp\perp} = \bigcap \mathcal{S}$.*

Proof. **(1)** Follows by Lemma 21(1).

(2) Follows by Lemma 21(2).

$$
\begin{array}{lcl}
[\![x : X]\!]_\sigma & \triangleq & \sigma(X)[x] \\
[\![x : \mathbf{1}]\!]_\sigma & \triangleq & \{P \mid P \equiv_{\mathsf{c}} \textbf{close}\ x \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : A \otimes B]\!]_\sigma & \triangleq & \{P \mid \exists P_1, P_2.\ P \equiv_{\mathsf{c}} \textbf{send}\ x(y.P_1); P_2 \text{ and } P_1 \in [\![y : A]\!]_\sigma \text{ and } P_2 \in [\![x : B]\!]_\sigma\}^{\perp\perp} \\
[\![x : A \oplus B]\!]_\sigma & \triangleq & \{P \mid \exists Q.\ (P \equiv_{\mathsf{c}} x.\textbf{inl}; Q \text{ and } Q \in [\![x : A]\!]_\sigma) \text{ or } (P \equiv_{\mathsf{c}} x.\textbf{inr}; Q \text{ and } Q \in [\![x : B]\!]_\sigma)\}^{\perp\perp} \\
[\![x : !A]\!]_\sigma & \triangleq & \{P \mid \exists Q.\ P \equiv_{\mathsf{c}} !x(y); Q \text{ and } Q \in [\![y : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \exists X.A]\!] & \triangleq & \{P \mid \exists Q, S \in \mathcal{R}[- : B].\ P \equiv_{\mathsf{c}} \textbf{sendty}\ x\ B; Q \text{ and } Q \in [\![x : A]\!]_{\sigma[X \mapsto S]}\}^{\perp\perp} \\
[\![x : \mu X.\ A]\!]_\sigma & \triangleq & (\bigcap\{S \in \mathcal{R}[- : \mu X.A] \mid \textbf{unfold}_\mu\ x; [\![x : A]\!]_{\sigma[X \mapsto S]} \subseteq S\})^{\perp\perp} \\
[\![x : \wedge A]\!]_\sigma & \triangleq & \{P \mid \exists Q.\ P \equiv_{\mathsf{c}} \textbf{affine}\ x; Q \text{ and } Q \in [\![x : A]\!]_\sigma\}^{\perp\perp} \\
[\![x : \mathbf{S}_f\ A]\!]_\sigma & \triangleq & \{P \mid P \equiv_{\mathsf{c}} \textbf{cell}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : \mathbf{S}_e\ A]\!]_\sigma & \triangleq & \{P \mid P \equiv_{\mathsf{c}} \textbf{empty}\ x(y.[\![y : \wedge A]\!]_\sigma) \text{ and } P \text{ is SN}\}^{\perp\perp} \\
[\![x : A]\!]_\sigma & \triangleq & [\![x : \overline{A}]\!]^\perp_\sigma
\end{array}
$$

Figure 9.1: Logical predicate $[\![x : A]\!]_\sigma$.

(3) Suppose $P \in S_2^\perp$.

So let $Q \in S_1$. Since $S_1 \subseteq S_2$, then $Q \in S_2$. Since $P \in S_2^\perp$, then **cut** $\{P \ \ |x|\ \ Q\}$ is SN.

Thus, $P \in S_1^\perp$.

(4) Let $P \in S$. We want $P \in S^{\perp\perp}$. Take $Q \in S^\perp$. It suffices to show that **cut** $\{P \ \ |x|\ \ Q\}$ is SN. It follows from $Q \in S^\perp$ and $P \in S$.

(5) From (2) and (3) follows $S^{\perp\perp\perp} \subseteq S^\perp$. From (3) follows $S^\perp \subseteq (S^\perp)^{\perp\perp} = S^{\perp\perp\perp}$.

(6) We prove that (i) $(\bigcup \mathcal{S})^\perp \subseteq \bigcap_{S \in \mathcal{S}} S^\perp$ and (ii) $\bigcap_{S \in \mathcal{S}} S^\perp \subseteq (\bigcup \mathcal{S})^\perp$.

(ii) follows immediately by Def. 32.

So let us consider (i).

Let $S \in \mathcal{S}$. Applying (3) to $S \subseteq \bigcup \mathcal{S}$ yields $(\bigcup \mathcal{S})^\perp \subseteq S^\perp$.

Then, $(\bigcup \mathcal{S})^\perp \subseteq \bigcap_{S \in \mathcal{S}} S^\perp$.

(7) We have

$$
\begin{array}{lcll}
(\bigcap \mathcal{S})^{\perp\perp} & = & (\bigcap_{S \in \mathcal{S}} S)^{\perp\perp} & \\
& = & (\bigcap_{S \in \mathcal{S}} S^{\perp\perp})^{\perp\perp} & (S = S^{\perp\perp}, \text{whenever } S \in \mathcal{S}) \\
& = & (\bigcup_{S \in \mathcal{S}} S^\perp)^{\perp\perp\perp} & (\text{from (6)}) \\
& = & (\bigcup_{S \in \mathcal{S}} S^\perp)^\perp & (\text{from (5)}) \\
& = & \bigcap_{S \in \mathcal{S}} S^{\perp\perp} & (\text{from (6)}) \\
& = & \bigcap_{S \in \mathcal{S}} S & (S = S^{\perp\perp}, \text{whenever } S \in \mathcal{S})
\end{array}
$$

□

Logical Predicates $[\![x : A]\!]_\sigma$

We will now introduce the logical predicates $[\![x : A]\!]_\sigma$ for strong normalisation. Since we are working with polymorphic and inductive types, the definition is parametric on a map σ from type variables to reducibility candidates. So let us define reducibility candidates first.

Definition 33 (Reducibility Candidates $R[x : A]$)**.** *Given a type A and a name x we define a reducibility candidate at $x : A$, denoted by $R[x : A]$ as a set of SN processes $P \vdash x : A$ which is equal to its biorthogonal, i.e. $R[x : A] = R[x : A]^{\perp\perp}$.*

We let $\mathcal{R}[- : A]$ be the set of all reducibility candidates $R[x : A]$ for some name x. Reducibility candidates are ordered by set-inclusion $\subseteq$, the least candidate being $\emptyset^{\perp\perp}$.

Definition 34 (Logical Predicate $[\![x : A]\!]_\sigma$)**.** *By induction on the type A, we define the sets $[\![x : A]\!]_\sigma$ an shown in Fig. 9.1, such that $[\![x : \mathbf{U}_f\ A]\!]_\sigma$ and $[\![x : \mathbf{U}_e\ A]\!]_\sigma$ are $[\![- : \wedge\overline{A}]\!]$-preserving on x.The definition is direct for the positive types, the negative types are defined by the last clause, by orthogonality.*

For the positive types A, the predicate $[\![x : A]\!]_\sigma$ takes the biorthogonal of some base set S of processes P that offer an action, further conditions then characterise the process constituents of the actions. In the base cases **close** x, **cell** $x(y.[\![y : \wedge A]\!]_\sigma)$ and **empty** $x(y.[\![y : \wedge A]\!]_\sigma)$, where the action does not have any further process constituents, we simply require the action offering process to be SN.

The presence of duality give us some succinctness in the presentation of the logical predicates, since, for the negative types A, the predicate $[\![x : A]\!]_\sigma$ is simply defined as the biorthogonal of the logical predicate for its dual $\overline{A}$ type. In fact, we can also establish this property for the positive types (Lemma 23(4)), thereby lifting duality to the logical level using the orthogonal operation. As a pleasant consequence we conclude immediately that if $P[\![x : A]\!]_\sigma$ and $Q \in [\![x : \overline{A}]\!]_\sigma$, then the resulting cut composition **cut** $\{P \ |x|\ Q\}$ is SN.

By exploiting the properties satisfied by the orthogonal (Lemma 22) we obtain a strategy to establish the membership $P \in [\![x : A]\!]_\sigma$. For the positive types we have $[\![x : A]\!]_\sigma = S^{\perp\perp}$, for some set S. Since $S \subseteq S^{\perp\perp}$ (Lemma 22(4)), we can conclude that $P \in [\![x : A]\!]_\sigma$, provided we prove $P \in S$. On the other hand, for the negative types we have $[\![x : A]\!]_\sigma = S^{\perp\perp\perp}$. But since $S^{\perp\perp\perp} = S^{\perp}$ (Lemma 22(5)), it is equivalent to prove that for all $Q \in S$, **cut** $\{P \ |x|\ Q\}$ is SN. These strategies will be applied throughout the proof of the Fundamental Lemma 27.

In all cases, with some exceptions, when defining $[\![x : A]\!]_\sigma$ we simply propagate map σ without modifications. The exceptions are the defining clauses corresponding to the existential $\exists X.A$ and the inductive types $\mu X.\ A$, in which we extend the map σ with an assignment for the type variable X. Furthermore, the definition of the predicate for a type variable $[\![x : X]\!]_\sigma$ picks the corresponding reducibility candidate $\sigma(X) = R[y : B]$, instantiated at name x: $\{x/y\}R[y : B]$.

The definition of $[\![x : \mu X.\, A]\!]_\sigma$ relies on the construction $\mathbf{unfold}_\mu\ x; S$, that for any set S, is defined according to

$$\mathbf{unfold}_\mu\ x; S \triangleq \{P \mid \exists Q.\, P \equiv_{\mathsf{c}} \mathbf{unfold}_\mu\ x; Q \text{ and } Q \in S\}$$

Similarly, given a set S, we define $\mathbf{unfold}_\nu\ x; A$ by

$$\mathbf{unfold}_\nu\ x; S \triangleq \{P \mid \exists Q.\, P \equiv_{\mathsf{c}} \mathbf{unfold}_\nu\ x; Q \text{ and } Q \in S\}$$

The following lemma states some basic properties about the logical predicates.

Lemma 23 (Logical Predicates: Basic Properties)**.** *The following properties hold*

(1) *If* $P \in [\![x : A]\!]_\sigma$*, then* $\{y/x\}P \in [\![y : A]\!]_\sigma$.

(2) *If* $P \in [\![x : A]\!]_\sigma$ *and* $P \equiv_{\mathsf{c}} Q$*, then* $Q \in [\![x : A]\!]_\sigma$.

(3) *If* $P \in [\![x : A]\!]_\sigma$ *and* $P \rightarrow_{\mathsf{c}} Q$*, then* $Q \in [\![x : A]\!]_\sigma$.

(4) $[\![x : \overline{A}]\!]_\sigma = [\![x : A]\!]^{\perp}_{\sigma}$.

(5) $[\![x : \{B/X\}A]\!]_\sigma = [\![x : A]\!]_{\sigma[X \mapsto [\![x:B]\!]_\sigma]}$.

(6) $[\![x : A]\!]_{\sigma[X \mapsto S^{\perp}]} = [\![x : \{\overline{X}/X\}A]\!]_{\sigma[X \mapsto S]}$.

Proof. Property (1) is trivial. Properties (2) and (3) follows by Lemma 22(1) and Lemma 22(2), respectively. Property (4) follows directly by Def. 34 for half of the types. The remaining half follows by Lemma 22(5). Properties (5) and (6) are straightforward by induction on A. □

The logical predicates are preserved by name substitution, the congruence relation $\equiv_{\mathsf{c}}$ and the reduction relation $\rightarrow_{\mathsf{c}}$ (Lemma 23(1)-(3)). Property Lemma 23(4) relates the logical predicates of duality related types, using the orthogonal. Lemma 23(5)-(6) relate type variable substitution with the parametric map σ.

We use the interference-sensitive reference cells (Def. 30) to define the logical predicates $[\![x : \mathbf{S}_f\ A]\!]_\sigma$ and $[\![c : \mathbf{S}_e\ A]\!]_\sigma$, for the state full and the state empty modalities, respectively. This allows us to internalise state interference in the definition of the logical predicate itself and, as consequence, we can reason compositionally about state sharing as witnessed by the following lemma

Lemma 24. *The following properties hold*

(1) *If* $P_1 \in [\![c : \mathbf{U}_f\ A]\!]_\sigma$ *and* $P_2 \in [\![c : \mathbf{U}_f\ A]\!]_\sigma$*, then* $\mathbf{share}\ c\ \{P_1 \mid\mid P_2\} \in [\![c : \mathbf{U}_f\ A]\!]_\sigma$.

(2) *If* $P_1 \in [\![c : \mathbf{U}_e\ A]\!]_\sigma$ *and* $P_2 \in [\![c : \mathbf{U}_e\ A]\!]_\sigma$*, then* $\mathbf{share}\ c\ \{P_1 \mid\mid P_2\} \in [\![c : \mathbf{U}_e\ A]\!]_\sigma$.

Proof. **(1)** By Def. 32 and Lemma 22(5) we have $[\![c : \mathbf{U}_f\ A]\!] = S^{\perp}$, where

$$S = \{Q \mid Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])_{\sigma}\}.$$

Let $Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])_{\sigma}$.

We need to prove that $\mathbf{cut}\ \{Q\ \ |c|\ \ \mathbf{share}\ c\ \{P_1 \mid\mid P_2\}\}$ is SN.

By Lemma 20(1) we conclude that $\mathbf{cut}\ \{Q\ \ |c|\ \ \mathbf{share}\ c\ \{P_1 \mid\mid P_2\}\}$ is simulated by

$$\mathbf{par}\ \{(\mathbf{cut}\ \{Q\ \ |c|\ \ P_1\}) \mid\mid (\mathbf{cut}\ \{Q\ \ |c|\ \ P_2\})\}$$

By hypothesis, $P_1 \in [\![c : \mathbf{U}_f\ A]\!]_{\sigma}$, hence $\mathbf{cut}\ \{Q\ \ |c|\ \ P_1\}$ is SN.

By hypothesis, $P_2 \in [\![c : \mathbf{U}_f\ A]\!]_{\sigma}$, hence $\mathbf{cut}\ \{Q\ \ |c|\ \ P_2\}$ is SN.

Then, $\mathbf{par}\ \{(\mathbf{cut}\ \{Q\ \ |c|\ \ P_1\}) \mid\mid (\mathbf{cut}\ \{Q\ \ |c|\ \ P_2\})\}$ is SN (Lemma 21(4)).

Therefore, $\mathbf{cut}\ \{Q\ \ |c|\ \ \mathbf{share}\ c\ \{P_1 \mid\mid P_2\}\}$ is SN (Lemma 21(5)).

By hypothesis, for any y, both P_1 and P_2 are $[\![y : \wedge\overline{A}]\!]$-preserving on c. Applying Lemma 19, we conclude that $\mathbf{share}\ c\ \{P_1 \mid\mid P_2\}$ is also $[\![y : \wedge\overline{A}]\!]$-preserving on c.

(2) Similarly to (1), by applying the simulation Lemma 20(2).

□

We will now state some properties concerning the logical predicate for inductive types. But first, let us introduce the following definition.

Definition 35 ($\phi_A(S)$)**.** *Suppose that X occurs positively on A. Define*

$$\phi_A(S) \triangleq \mathbf{unfold}_{\mu}\ x; [\![x : A]\!]_{\sigma[X \mapsto S]}$$

$[\![x : \mu X.\ A]\!]_{\sigma}$ is defined as the biorthogonal of the intersection of all ϕ_A-closed sets S, i.e. sets S s.t. $\phi_A(S) \subseteq S$. Since ϕ_A is monotonic (Lemma 25(1)), Knaster-Tarski theorem implies that $[\![x : \mu X.\ A]\!]_{\sigma}$ is the least fixed point of ϕ_A (Lemma 25(2)). Symmetrically, we can obtain a greatest fixed point characterisation for $[\![x : \nu X.\ A]\!]_{\sigma}$ (Lemma 25(3)). Applying Kleene's fixed point theorem we explicitly construct the fixed point of ϕ_A (Lemma 25(4)).

Lemma 25. *The following properties hold*

(1) *The map ϕ_A is monotonic, i.e. $\phi_A(S_1) \subseteq \phi_A(S_2)$, whenever $S_1 \subseteq S_2$.*

(2) *$[\![x : \mu X.\ A]\!]_{\sigma}$ is the least fixed point of ϕ_A.*

(3) *Let $\psi_A(S) \triangleq \phi_{\{\overline{X}/X\}\overline{A}}(S^{\perp})^{\perp}$. Then, $[\![x : \nu X.\ A]\!]_{\sigma}$ is the greatest fixed point of ψ_A.*

(4) $[\![x : \mu X.\ A]\!]_{\sigma} = \bigcup_{n \in \mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$.

(5) $\mathbf{unfold}_{\nu}\ x; [\![x : \{\nu X.\ A/X\}A]\!]_{\sigma} \subseteq [\![x : \nu X.\ A]\!]_{\sigma}$.

Proof. **(1)** We prove hypothesis (H1) if $S_1 \subseteq S_2$, then $[\![x : A]\!]_{\sigma[X\mapsto S_1]} \subseteq [\![x : A]\!]_{\sigma[X\mapsto S_2]}$, which implies (1).

The proof of (H1) is by induction on A, we handle some representative cases.

Case: $A = Y$.

There are two cases to consider, depending on whether (i) $Y \neq X$ or (ii) $Y = X$.

If (i), then $[\![x : Y]\!]_{\sigma[X\mapsto S_1]} = \sigma(Y) = [\![x : Y]\!]_{\sigma[X\mapsto S_2]}$.

If (ii), then $[\![x : X]\!]_{\sigma[X\mapsto S_1]} = S_1 \subseteq S_2 = [\![x : X]\!]_{\sigma[X\mapsto S_2]}$.

In either case (i)-(ii), $[\![x : Y]\!]_{\sigma[X\mapsto S_1]} \subseteq [\![x : Y]\!]_{\sigma[X\mapsto S_2]}$.

Case: $A = \mathbf{1}$.

We have $[\![x : \mathbf{1}]\!]_{\sigma[X\mapsto S_1]} = [\![x : \mathbf{1}]\!]_{\sigma[X\mapsto S_2]}$.

Case: $A = A_1 \otimes A_2$.

By Def. 34,

$$[\![x : A_1 \otimes A_2]\!]_{\sigma[X\mapsto S]} = f(S)^{\perp\perp}$$

where

$$f(S) \triangleq \{P \mid \exists P_1, P_2.\ P \approx \mathbf{send}\ x(y.P_1); P_2 \text{ and } P_1 \in [\![y : A_1]\!]_{\sigma[X\mapsto S]} \text{ and } P_2 \in [\![x : A_2]\!]_{\sigma[X\mapsto S]}\}$$

Suppose that $S_1 \subseteq S_2$. I.h. applied to A_1 and A_2 yields $f(S_1) \subseteq f(S_2)$.

Lemma 22(3) applied twice to $f(S_1) \subseteq f(S_2)$ yields

$$[\![x : A_1 \otimes A_2]\!]_{\sigma[X\mapsto S_1]} = f(S_1)^{\perp\perp} \subseteq f(S_2)^{\perp\perp} = [\![x : A_1 \otimes A_2]\!]_{\sigma[X\mapsto S_2]}$$

Case: $A = \mu Y.\ B$.

By Def. 34

$$[\![x : \mu Y.\ B]\!]_{\sigma[X\mapsto S]} = (\bigcap f(S))^{\perp\perp}$$

where

$$f(S) \triangleq \{T \in \mathcal{R}[- : \mu Y.B] \mid \mathbf{unfold}_\mu\ x; [\![x : B]\!]_{\sigma[X\mapsto S, Y\mapsto T]} \subseteq T\}$$

Suppose $S_1 \subseteq S_2$. Let $T \in f(S_2)$. Then, $\mathbf{unfold}_\mu\ x; [\![x : B]\!]_{\sigma[X\mapsto S_2, Y\mapsto T]} \subseteq T$.

I.h. applied to B yields $\mathbf{unfold}_\mu\ x; [\![x : B]\!]_{\sigma[X\mapsto S_1, Y\mapsto T]} \subseteq \mathbf{unfold}_\mu\ x; [\![x : B]\!]_{\sigma[X\mapsto S_2, Y\mapsto T]}$.

By transitivity of $\subseteq$, $\mathbf{unfold}_\mu\ x; [\![x : B]\!]_{\sigma[X\mapsto S_1, Y\mapsto T]} \subseteq T$.

Hence, $T \in f(S_1)$.

This establishes $f(S_2) \subseteq f(S_1)$.

Then, $\bigcap f(S_1) \subseteq \bigcap f(S_2)$.

Lemma 22(3) applied twice to $\bigcap f(S_1) \subseteq \bigcap f(S_2)$ yields

$$[\![x : \mu Y.\ B]\!]_{\sigma[X\mapsto S_1]} = (\bigcap f(S_1))^{\perp\perp} \subseteq (\bigcap f(S_2))^{\perp\perp} = [\![x : \mu Y.\ B]\!]_{\sigma[X\mapsto S_2]}$$

Case: $A = \mathbf{S}_f\ B$.

By Def. 34

$$[\![x : \mathbf{S}_f\ B]\!]_{\sigma[X\mapsto S]} = f(S)^{\perp\perp}$$

where

$$f(S) \triangleq \{P \mid P \equiv_{\mathsf{c}} \mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S]})\}$$

Suppose $S_1 \subseteq S_2$. We prove that $f(S_2)^{\perp} \subseteq f(S_1)^{\perp}$.

Let $Q \in f(S_2)^{\perp}$. In order to show that $Q \in f(S_1)^{\perp}$ we must show that $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ is SN, when $P \in f(S_1)$.

We prove by induction on $N(P)+N(Q)$ that all the reductions $\mathbf{cut}\ \{P\ \ |x|\ \ Q\} \to R$ are SN.

We handle only the interesting reduction, which corresponds to a cell-take interaction on session x. Then

$$\begin{aligned}\mathbf{cut}\ \{P\ \ |x|\ \ Q\} \quad &\equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_1]})\ \ |x|\ \ \mathbf{take}\ x(y); Q'\} \\ &\to_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_1]})\ \ |x|\ \ (\mathbf{cut}\ \{P'\ \ |y|\ \ Q'\})\} = R\end{aligned}$$

where $P \equiv_{\mathsf{c}} \mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_1]})$, $Q \equiv_{\mathsf{c}} \mathbf{take}\ x(y); Q'$ and P' is some element in $[\![y : \wedge A]\!]_{\sigma[X\mapsto S_1]}$. By hypothesis, $Q \in f(S_2)^{\perp}$, hence

$$\mathbf{cut}\ \{\mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_2]})\ \ |x|\ \ \mathbf{take}\ x(y); Q'\}$$

is SN.

Then, all the reductions of $\mathbf{cut}\ \{\mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_2]})\ \ |x|\ \ \mathbf{take}\ x(y); Q'\}$ are SN, in particular the following reduction can be obtained, since $P' \in S_1 \subseteq S_2$:

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{cell}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_2]})\ \ |x|\ \ \mathbf{take}\ x(y); Q'\} \\ &\to \mathbf{cut}\ \{\mathbf{empty}\ x(y.[\![y : \wedge A]\!]_{\sigma[X\mapsto S_2]})\ \ |x|\ \ (\mathbf{cut}\ \{P'\ \ |y|\ \ Q'\})\}\end{aligned}$$

(2) By Def. 34

$$[\![x : \mu X.\ A]\!]_{\sigma} \quad = \quad (\bigcap\{S \in \mathcal{R}[- : \mu X.A] \mid \phi_A(S) \subseteq S\})^{\perp\perp}$$

Since a reducibility candidate is equal to its biorthogonal (Def. 33), we can write $[\![x : \mu X.\ A]\!]_{\sigma}$ in the alternative form (Lemma 22(7))

$$[\![x : \mu X.\ A]\!]_{\sigma} = \bigcap\{S \in \mathcal{R}[- : \mu X.A] \mid \phi_A(S) \subseteq S\}$$

i.e. $[\![x : \mu X.\ A]\!]_{\sigma}$ is the intersection of all ϕ_A-closed sets in $\mathcal{R}[- : \mu X.A]$.

We now prove the following propositions

(i) $[\![x : \mu X.\ A]\!]_{\sigma}$ is ϕ_A-closed, i.e. $\phi_A([\![x : \mu X.\ A]\!]_{\sigma}) \subseteq [\![x : \mu X.\ A]\!]_{\sigma}$.

Let $S \in \mathcal{R}[- : \mu X.A]$ be a ϕ_A-closed set.

By definition, we have (a) $\phi_A(S) \subseteq S$ and (b) $[\![x : \mu X.\ A]\!]_{\sigma} \subseteq S$.

Monotonicity of ϕ_A (1) applied to (b) yields $\phi_A([\![x : \mu X.\ A]\!]_\sigma \subseteq \phi_A(S)$.

Hence, transitivity and (a) implies $\phi_A([\![x : \mu X.\ A]\!]_\sigma) \subseteq S$.

Since $[\![x : \mu X.\ A]\!]_\sigma$ is the intersection of all ϕ_A-closed sets in $\mathcal{R}[- : \mu X.A]$, then $\phi_A([\![x : \mu X.\ A]\!]_\sigma) \subseteq [\![x : \mu X.\ A]\!]_\sigma$.

(ii) $[\![x : \mu X.\ A]\!]_\sigma \subseteq \phi_A([\![x : \mu X.\ A]\!]_\sigma)$.
Monotonicity of ϕ_A (1) applied to (i) yields $\phi_A(\phi_A([\![x : \mu X.\ A]\!]_\sigma) \subseteq \phi_A([\![x : \mu X.\ A]\!]_\sigma)$, i.e. $\phi_A([\![x : \mu X.\ A]\!]_\sigma)$ is ϕ_A-closed.
Since $[\![x : \mu X.\ A]\!]_\sigma$ is the intersection of all ϕ_A-closed sets in $\mathcal{R}[- : \mu X.A]$, then $[\![x : \mu X.\ A]\!]_\sigma \subseteq \phi_A([\![x : \mu X.\ A]\!]_\sigma)$.

Propositions (i) and (ii) imply that $[\![x : \mu X.\ A]\!]_\sigma$ is a fixed point of ϕ_A.

Let $S \in \mathcal{R}[- : \mu X.A]$ be any fixed point of ϕ_A. Then, in particular, S is ϕ_A-closed, hence $[\![x : \mu X.\ A]\!]_\sigma \subseteq \phi_A$.

Therefore, $[\![x : \mu X.\ A]\!]_\sigma$ is the least fixed point of ϕ_A.

(3) We need to prove the following propositions

(i) $[\![x : \nu X.\ XA]\!]_\sigma$ is a fixed point of ψ_A.
By (b), $[\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma$ is a fixed point of $\phi_{\{\overline{X}/X\}\overline{A}}$

$$\phi_{\{\overline{X}/X\}\overline{A}}([\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma) = [\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma$$

hence, applying the orthogonal to both sides of the equation yields

$$\phi_{\{\overline{X}/X\}\overline{A}}([\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma)^\perp = [\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma^\perp$$

Since $[\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_\sigma^\perp = [\![x : \nu X.\ XA]\!]_\sigma$ (Lemma 23(4)) we can rewrite the equation in the equivalent form

$$\phi_{\{\overline{X}/X\}\overline{A}}([\![x : \nu X.\ XA]\!]_\sigma^\perp)^\perp = [\![x : \nu X.\ XA]\!]_\sigma$$

Then, $[\![x : \nu X.\ XA]\!]_\sigma$ is a fixed point of ψ_A.

(ii) If S is a fixed point of ψ_A, then $S \subseteq [\![x : \nu X.\ XA]\!]_\sigma$.
Suppose that S is a fixed point of ψ_A, i.e.

$$\psi_A(S) = \phi_{\{\overline{X}/X\}\overline{A}}(S^\perp)^\perp = S$$

Applying the orthogonal to both sides of the equation yields

$$\phi_{\{\overline{X}/X\}\overline{A}}(S^\perp)^{\perp\perp} = S^\perp$$

Since $\phi_{\{\overline{X}/X\}\overline{A}}(S^\perp) \subseteq \phi_{\{\overline{X}/X\}\overline{A}}(S^\perp)^{\perp\perp}$ (Lemma 22(4)), then

$$\phi_{\{\overline{X}/X\}\overline{A}}(S^\perp) \subseteq S^\perp$$

i.e. $S^{\perp}$ is a $\phi_{\{\overline{X}/X\}\overline{A}}$-closed set.

Then, by Def. 34

$$[\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]_{\sigma} \subseteq S^{\perp}$$

Applying the orthogonal to the inequation (Lemma 22(3)) yields

$$S^{\perp\perp} \subseteq [\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]^{\perp}_{\sigma}$$

Since $S \subseteq S^{\perp\perp}$ (Lemma 22(2)), we obtain

$$S \subseteq [\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]^{\perp}_{\sigma}$$

Finally, since $[\![x : \mu X.\ \{\overline{X}/X\}\overline{A}]\!]^{\perp}_{\sigma} = [\![x : \nu X.\ A]\!]_{\sigma}$ (Lemma 23(4)) we have

$$S \subseteq [\![x : \nu X.\ A]\!]_{\sigma}$$

(4) We prove that $\bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$ is the least fixed point of ϕ_A.

By (b) it follows that $[\![x : \mu X.\ A]\!]_{\sigma} = \bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$.

We need to prove the following propositions

(i) $\bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$ is a fixed point of ϕ_A.

We have

$$\phi_A(\bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})) = \bigcup_{n>0} \phi_A^n(\emptyset^{\perp\perp})$$

Since $\phi_A^0(\emptyset^{\perp\perp}) = \emptyset^{\perp\perp}$ is the least reducibility candidate, we have $\phi_A^0(\emptyset^{\perp\perp}) \subseteq \phi_A^n(\emptyset^{\perp\perp})$, for any $n > 0$.

Then

$$\bigcup_{n>0} \phi_A^n(\emptyset^{\perp\perp}) = \bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$$

Therefore

$$\phi_A(\bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})) = \bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp})$$

(ii) If S is fixed point of ϕ_A, then $\bigcup_{n\in\mathbb{N}} \phi_A^n(\emptyset^{\perp\perp}) \subseteq S$.

We that $\phi_A^n(\emptyset^{\perp\perp}) \subseteq S$, for all $n \in \mathbb{N}$. The proof is by induction on n.

Case: $n = 0$.

Since $\phi_A^0(\emptyset^{\perp\perp}) = \emptyset^{\perp\perp}$ is the least reducibility candidate, $\phi_A^0(\emptyset^{\perp\perp}) \subseteq S$.

Case: $n = m + 1$.

By i.h. we have

$$\phi_A^m(\emptyset^{\perp\perp}) \subseteq S$$

Monotonicity of ϕ_A (1) implies

$$\phi_A^{m+1}(\emptyset^{\perp\perp})) \subseteq \phi(S)$$

Since $\phi(S) = S$, then

$$\phi_A^{m+1}(\emptyset^{\perp\perp})) \subseteq S$$

(5) Let $P \equiv_{\mathsf{c}} \mathbf{unfold}_{\nu}\ x; P'$, where $P' \in [\![x : \{\nu X.\ A/X\}A]\!]_\sigma$.

Let $B \triangleq \{\overline{X}/X\}\overline{A}$, hence $\overline{\nu X.\ A} = \mu X.\ B$.

We prove that $\mathbf{cut}\ \{P \ \ |x|\ \ Q\}$ is SN, for all $Q \in [\![x : \mu X.\ B]\!]_\sigma$, by analysing all the possible reductions of $\mathbf{cut}\ \{P \ \ |x|\ \ Q\}$ and concluding that all of them are SN.

The critical reduction is the unfold-unfold interaction on session x, in which case $Q \equiv_{\mathsf{c}} \mathbf{unfold}_{\nu}\ x; Q'$, and $\mathbf{cut}\ \{P \ \ |x|\ \ Q\} \to_{\mathsf{c}} \mathbf{cut}\ \{P' \ \ |x|\ \ Q'\}$.

By (2) we conclude that $Q' \in [\![x : \{\mu X.\ B/X\}B]\!]_\sigma$.

Since $\overline{\{\nu X.\ A/X\}A} = \{\mu X.\ B/X\}B$, we conclude that $\mathbf{cut}\ \{P' \ \ |x|\ \ Q'\}$ is SN.

□

Extended Logical Predicate and Fundamental Lemma

The logical predicates $[\![x : A]\!]_\sigma$ introduced previously apply to processes that have a single free name x. Now, we will extend the logical predicate to general typed processes $P \in [\![\vdash_\eta \Delta; \Gamma]\!]$ by composing it along Δ and Γ with processes from the basic logical predicates (Def. 34) and by replacing the free process variables by elements of the appropriate reducibility candidate, according to the following definition.

Definition 36 (Extended Logical Predicate $[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$)**.** *We define* $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$ *inductively on* Δ, Γ *and* η *as the set of processes* $P \vdash_\eta \Delta; \Gamma$ *s.t.*

$$
\begin{array}{l}
P \in \mathcal{L}[\![\vdash_\eta \emptyset; \emptyset]\!]_\sigma \textit{ iff } P \textit{ is SN.} \\
P \in \mathcal{L}[\![\vdash_\eta \Delta, x : A; \Gamma]\!]_\sigma \textit{ iff } \forall Q \in [\![x : \overline{A}]\!]_\sigma.\ \mathbf{cut}\ \{Q \ \ |x : \overline{A}|\ \ P\} \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma. \\
P \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma, x : A]\!]_\sigma \textit{ iff } \forall Q \in [\![y : \overline{A}]\!]_\sigma.\ \mathbf{cut!}\ \{y.Q \ |x : \overline{A}|\ P\} \in \mathcal{L}[\![\vdash_\emptyset \Delta; \Gamma]\!]_\sigma. \\
P \in \mathcal{L}[\![\vdash_{\eta, X(x,\vec{y}) \mapsto \Delta', x:Y; \Gamma} \Delta; \Gamma]\!]_\sigma \textit{ iff } \forall Q \in \sigma(Y).\ \{Q/X\}P \ \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma.
\end{array}
$$

The base case $\mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$ corresponds to the set of closed well-typed SN processes.

We will now introduce some auxiliary definitions which allows us to give a more succinct definition of $\mathcal{L}[\![\vdash_\emptyset \emptyset; \emptyset]\!]_\sigma$. We define the set $[\![\Delta]\!]_\sigma$ of linear logical contexts at Δ is inductively by

$$
[\![\emptyset]\!]_\sigma \triangleq \{-\} \quad [\![\Delta, x : A]\!]_\sigma \triangleq \{\mathbf{cut}\ \{P \ \ |x : \overline{A}|\ \ C\} \mid P \in [\![x : \overline{A}]\!]_\sigma \text{ and } C \in [\![\Delta]\!]_\sigma\}
$$

Similarly, we define the set $[\![\Gamma]\!]^!_\sigma$ of unrestricted logical contexts at Γ inductively by

$$
[\![\emptyset]\!]^!_\sigma \triangleq \{-\} \quad [\![\Gamma, y : A]\!]^!_\sigma \triangleq \{\mathbf{cut!}\ \{y.P \ |x : \overline{A}|\ C\} \mid P \in [\![y : \overline{A}]\!]_\sigma \text{ and } C \in [\![\Gamma]\!]^!_\sigma\}
$$

We extend N from processes to contexts $C \in [\![\Delta]\!]_\sigma$ by $N(-) = 0$ and $N(\mathbf{cut}\ \{P \ \ |x|\ \ C'\}) = N(P) + N(C')$.

Given a map

$$
\eta = X_1(\vec{x_1}) \mapsto \Delta_1; \Gamma_1, \ldots, X_n(\vec{x_n}) \mapsto \Delta_n; \Gamma_n
$$

we define $[\![\eta]\!]_\sigma$ as the set of all substitution maps η' s.t.

$$
\eta' = X_1(\vec{x_1}) \mapsto Q_1, \ldots, X_n(\vec{x_n}) \mapsto Q_n
$$

where $Q_1 \in \mathcal{L}[\![\vdash_\emptyset \Delta_1; \Gamma_1]\!]_\sigma, \ldots, Q_n \in \mathcal{L}[\![\vdash_\emptyset \Delta_n; \Gamma_n]\!]_\sigma$.

Then, Def. 36 is equivalent to the following

$$P \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma \quad \text{iff} \quad \forall \eta' \in [\![\eta]\!]_\sigma \; \forall \mathcal{C} \in [\![\Delta]\!]_\sigma \; \forall \mathcal{D} \in [\![\Gamma]\!]^!_\sigma . \; \eta'(\mathcal{C} \circ \mathcal{D}[P]) \text{ is SN.}$$

where we denote by $\eta'(P)$ the process obtained by substituting the variables in P by processes according to η'.

The following property establishes an equivalence between the extended logical predicate and the basic logical predicates of Def. 34. In one direction it establishes that if $P \in \mathcal{L}[\![\vdash_\emptyset \Delta, x : A; \Gamma]\!]_\sigma$, then we can cut the process along Δ and Γ and prove that the resulting cut composition is an element of $[\![x : A]\!]_\sigma$.

Lemma 26. *The following two propositions*

(1) $P \in \mathcal{L}[\![\vdash_\eta \Delta, x : A; \Gamma]\!]_\sigma$.

(2) *For all* $\mathcal{C} \in [\![\Delta]\!]_\sigma$ *and* $\mathcal{D} \in [\![\Gamma]\!]^!_\sigma$, $\mathcal{C} \circ \mathcal{D}[P] \; \in [\![x : A]\!]_\sigma$.

are equivalent.

Proof. By Lemma 23(4). □

Lemma 26 gives us a degree of freedom in the sense that we can choose an arbitrary typed channel $x : A$ from a nonempty linear typing context Δ of a typed process $P \vdash_\eta \Delta; \Gamma$ and cut the remaining linear context. We conclude this section with the proof of the Fundamental Lemma 27, from which strong normalisation (Theorem 5) follows immediately.

Lemma 27 (Fundamental Lemma)**.** *If* $P \vdash_\eta \Delta; \Gamma$, *then* $P \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma]\!]_\sigma$.

Proof. By induction on the structure of a typing derivation for $P \vdash_\eta \Delta; \Gamma$. Cases [Tcut], [Tfwd], [Tcut!] follow immediately because $[\![x : A]\!] = [\![x : \overline{A}]\!]^\perp$. Case [T**0**] follows because **0** is SN and case [Tmix] follows because **par** $\{P \mid\mid Q\}$ is SN whenever P and Q are SN. For the positive types A, the logical predicate $[\![x : A]\!]_\sigma$ is defined as the biorthogonal of some set S, hence for the typing rules that introduce a positive type A the strategy is to show that the introduced action P lies in $S \subseteq S^{\perp\perp}$. For the negative types $\overline{A}$: $[\![x : \overline{A}]\!]_\sigma = S^{\perp\perp\perp} = S^\perp$, hence the strategy for the typing rules that that introduce an action Q that types with a negative type $x : \overline{A}$ is to show that **cut** $\{P \;\; |x : A| \;\; Q\}$ is SN, for all $P \in S$. Particularly, for rule [Tcorec], where $A = \mu X.\, B$, we proceed by induction on the depth n of unfolding, since $S \bigcup_{n \in \mathbb{N}} \phi_B^n(\emptyset^{\perp\perp})$. Cases [Tcell] and [Tempty] follow by applying the simulations Lemma 18(1)-(2). Cases [Tsh], [TshL], [TshR] follows after applying the *decomposition of the share as a mix* as given by Lemma 20(1)-(2). We illustrate the proof with some cases. In the cases in which the recursive map η that annotates the typing judgments $P \vdash_\eta \Delta; \Gamma$ plays no role and is essentially propagated from the conclusion to the premises of the typing

rule we omit it, working as if the process P did not have any free recursion variable X. Similarly for the map σ which annotates the logical predicates $[\![x : A]\!]_\sigma$. The complete proof can be found in Appendix E.

Case [Tcut]:

$$\frac{P_1 \vdash \Delta_1, x : \overline{A}; \Gamma \quad P_2 \vdash \Delta_2, x : A; \Gamma}{\mathbf{cut}\ \{P_1 \ |x|\ P_2\} \vdash \Delta_1, \Delta_2; \Gamma}$$

Let $C_1 \in [\![\Delta_1]\!]$, $C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$C_1 \circ C_2 \circ \mathcal{D}[\mathbf{cut}\ \{P_1 \ |x|\ P_2\}] \equiv_{\mathsf{c}} \mathbf{cut}\ \{(C_1 \circ \mathcal{D}[P_1]) \ |x|\ (C_2 \circ \mathcal{D}[P_2])\}$$

I.h. and Lemma 26 applied to $P_1 \vdash \Delta_1, x : \overline{A}; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![x : \overline{A}]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash \Delta_2, x : A; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2] \in [\![x : A]\!]$.

By applying Lemma 23(4) we conclude that $\mathbf{cut}\ \{(C_1 \circ \mathcal{D}[P_1]) \ |x|\ (C_2 \circ \mathcal{D}[P_2])\}$ is SN.

Hence, $C \circ \mathcal{D}[\mathbf{cut}\ \{P_1 \ |x|\ P_2\}]$ is SN.

Case: [Tcorec]

$$\frac{\{x/z\}\{\vec{y}/\vec{w}\}P' \vdash_{\eta'} \Delta', x : A; \Gamma \quad \eta' = \eta, Y(x, \vec{y}) \mapsto \Delta', x : X; \Gamma}{\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}] \vdash_\eta \Delta', x : \nu X.\ A; \Gamma}$$

Let $\rho \in [\![\eta]\!]_\sigma$, $C \in [\![\Delta']\!]_\sigma$ and $\mathcal{D} \in [\![\Gamma]\!]^!{}_\sigma$.

We prove that $C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in [\![x : \nu X.\ A]\!]_\sigma$.

By Lemma 26, this implies that $\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}] \in \mathcal{L}[\![\vdash_\eta \Delta', x : \nu X.\ A; \Gamma]\!]_\sigma$.

By Lemma 25(5), we have

$$[\![x : \nu X.\ A]\!]_\sigma = \bigcap_{n \in \mathbb{N}} \phi^n_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp$$

where $\phi_{\{\overline{X}/X\}\overline{A}}(S) \triangleq \mathbf{unfold}_\mu\ x; [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto S]}$.

We prove (H1):

$$\forall n \in \mathbb{N},\ \forall \rho \in [\![\eta]\!]_\sigma,\ \forall C \in [\![\Delta']\!]_\sigma,\ \forall \mathcal{D} \in [\![\Gamma]\!]^!{}_\sigma.$$
$$C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in \phi^n_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp$$

Proof of (H1) is by induction on $n \in \mathbb{N}$:

Case: $n = 0$.

Follows because $C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in \emptyset^\perp$ and since $\phi^0_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp = \emptyset^{\perp\perp\perp} = \emptyset^\perp$ (Lemma 22(5)).

Case: $n = m + 1$.

Let $Q \in \phi^{m+1}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})$.

Then $Q \equiv_{\mathsf{c}} \mathbf{unfold}_{\mu}\ x; Q'$, where $Q' \in [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})]}$.

We prove (H2)

$$\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ \ |x|\ \ Q\} \text{ is SN}$$

by induction on $N(C) + N(\rho) + N(Q)$.

Suppose that $\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ \ |x|\ \ Q\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either C, ρ or Q.

Case: (ii) R is obtained by an interaction on session x.

Case (i) follows by inner inductive hypothesis (H2).

So let us consider case (ii). Then

$$\begin{aligned}
&\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ \ |x|\ \ Q\} \\
&\equiv_{\mathsf{c}} \mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ \ |x|\ \ \mathbf{unfold}_{\mu}\ x; Q'\} \\
&\to_{\mathsf{c}} \mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\{x/z\}\{\vec{y}/\vec{w}\}\{\mathbf{corec}\ Y(z, \vec{w}); P'/Y\}P')]\ \ |x|\ \ Q'\} \\
&= \mathbf{cut}\ \{C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')]\ \ |x|\ \ Q'\} = R
\end{aligned}$$

where $\rho' = \rho, Y(x, \vec{y}) \mapsto \rho(\mathbf{corec}\ Y(z, \vec{w}); P')$.

I.h. (H1) applied to m yields

$$\begin{aligned}
&\forall C \in [\![\Delta']\!],\ \forall \mathcal{D} \in [\![\Gamma]\!]^{!}. \\
&\quad C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in \phi^{m}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}
\end{aligned}$$

Hence, by Lemma 26, we obtain

$$\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}]) \in \mathcal{L}[\![\vdash_{\emptyset} \Delta', x : X; \Gamma]\!]_{\sigma[X \mapsto \psi^{m}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}]}$$

Therefore, $\rho' \in [\![\eta']\!]_{\sigma}$.

Applying i.h. (outer i.h., fundamental lemma) to $\{x/z\}\{\vec{y}/\vec{w}\}P' \vdash_{\eta'} \Delta', x : A; \Gamma$ and Lemma 26 yields $C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')] \in [\![x : A]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})^{\perp}]}$.

Lemma 23(6) implies $C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')] \in [\![x : \{\overline{X}/X\}A]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})]}$.

By hypothesis, $Q' \in [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})]}$, hence by Lemma 23(3) we obtain that R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ |x|\ Q\}$ is SN.

Therefore, $C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])]\ \in \phi^{m+1}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}$.

Case [Tsh]:

$$\frac{P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_f\ A; \Gamma \quad P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ c\ \{P \parallel Q\} \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_f\ A; \Gamma}$$

Let $\mathcal{C}_1 \in [\![\Delta_1]\!]$, $\mathcal{C}_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$\begin{aligned}&\mathcal{C}_1 \circ \mathcal{C}_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1 \parallel P_2\}]\\ &\equiv_{\mathsf{c}} \mathbf{share}\ c\ \{\mathcal{C}_1 \circ \mathcal{D}[P_1] \parallel \mathcal{C}_2 \circ \mathcal{D}[P_2]\}\end{aligned}$$

I.h. and Lemma 26 applied to $P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_f\ A; \Gamma$ yields $\mathcal{C}_1 \circ \mathcal{D}[P_1] \in [\![c : \mathbf{U}_f\ A]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ yields $\mathcal{C}_2 \circ \mathcal{D}[P_2] \in [\![c : \mathbf{U}_f\ A]\!]$.

By applying Lemma 24(1) we conclude that $\mathcal{C}_1 \circ \mathcal{C}_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1 \parallel P_2\}] \in [\![c : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\mathbf{share}\ c\ \{P_1 \parallel P_2\} \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_f\ A; \Gamma]\!]$.

Cases: [TshL], [TshR]. Similarly to case [Tsh], by applying Lemma 24(2).

Case: [Tcell]

$$\frac{P' \vdash_\eta \Delta', a : \wedge A; \Gamma}{\mathbf{cell}\ c(a.P') \vdash_\eta \Delta', c : \mathbf{S}_f\ A; \Gamma}$$

Let $\mathcal{C} \in [\![\Delta']\!]$, $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!]$.

I.h. and Lemma 26 applied to $P' \vdash_\eta \Delta', a : \wedge A; \Gamma$ yields $\mathcal{C} \circ \mathcal{D}[P'] \in [\![a : \wedge A]\!]$.

Since $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!]$, then Q is $[\![a : \wedge A]\!]$-preserving.

Hence, by Lemma 18(1), $\mathbf{cut}\ \{\mathbf{cell}\ c(a.\mathcal{C} \circ \mathcal{D}[P'])\ |c|\ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge A]\!])\ |c|\ Q\}$.

Since $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!] = S^\perp$ where $S = \{R \mid R \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge A]\!])\}$, then $\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge A]\!])\ |c|\ Q\}$ is SN.

Hence, $\mathbf{cut}\ \{\mathcal{C} \circ \mathcal{D}[\mathbf{cell}\ c(a.P')]\ |c|\ Q\}$ is SN.

Then, $\mathbf{cell}\ c(a.P') \in \mathcal{L}[\![\vdash_\eta \Delta', c : \mathbf{S}_f\ A; \Gamma]\!]$.

Case: [Tempty]

$$\frac{}{\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma}$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!]$.

Since $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!]$, then Q is $[\![a : \wedge A]\!]$-preserving.

Hence, by Lemma 18(2), $\mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c|\ \ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ Q\}$.

Since $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!] = S^\perp$ where $S = \{R \mid R \equiv_{\mathsf{c}} \mathbf{empty}\ c([\![a : \wedge A]\!].)\}$, then $\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ Q\}$ is SN.

Hence, $\mathbf{cut}\ \{\mathcal{D}[\mathbf{empty}\ c]\ \ |c|\ \ Q\}$ is SN.

Then, $\mathbf{empty}\ c \in \mathcal{L}[\![\vdash_\eta c : \mathbf{S}_e\ A; \Gamma]\!]$.

Case: [Trelease]

$$\frac{}{\mathbf{release}\ c \vdash_\eta c : \mathbf{U}_f\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \mathbf{U}_f\ A]\!] = S^\perp$, where

$$S = \{Q \mid Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\}.$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\}$ is SN, by induction on $N(Q)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\} \equiv_{\mathsf{c}} \mathcal{D}[\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!]) \ \ |c|\ \ \mathbf{release}\ c\}] \xrightarrow{*}_{\mathsf{c}} \mathcal{D}[\mathbf{0}] = R$$

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\}$ is SN.

Furthermore, $\mathbf{release}\ c$ is vacuously $[\![y : \wedge\overline{A}]\!]$-preserving, for any y.

Therefore, $\mathcal{D}[\mathbf{release}\ c] \in [\![x : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\mathbf{release}\ c \in \mathcal{L}[\![\vdash_\eta a : \mathbf{U}_f\ A; \Gamma]\!]$.

Case: [Ttake]

$$\frac{P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma}{\mathbf{take}\ c(a); P' \vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![c : \mathbf{U}_f\ A]\!] = S^\perp$, where

$$S = \{Q \mid Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\}.$$

Let $\mathcal{C} \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{C} \circ \mathcal{D}[\mathbf{take}\ c(a); P']\}$ is SN, by induction on $N(Q)+N(\mathcal{C})$.

Suppose that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{C} \circ \mathcal{D}[\mathbf{take}\ c(a); P']\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or $\mathcal{C}$.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H). So let us consider case (ii). Then

$$\begin{aligned}&\mathbf{cut}\ \{Q\ \ |c|\ \ C \circ \mathcal{D}[\mathbf{take}\ c(a); P']\}\\ \equiv_{\mathsf{c}}\ &\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ C \circ \mathcal{D}[\mathbf{take}\ c(a); P']\}\\ \rightarrow_{\mathsf{c}}\ &\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ (\mathbf{cut}\ \{Q'\ \ |a|\ \ C \circ \mathcal{D}[P']\})\}\ = R\end{aligned}$$

where $Q' \in [\![a : \wedge\overline{A}]\!]$.

By Def. 32, $[\![c : \mathbf{S}_f\ \overline{A}]\!] = S^{\perp\perp}$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, hence $\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!]) \in [\![c : \mathbf{S}_f\ \overline{A}]\!]$.

Applying i.h. to $P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q\ \ |c|\ \ C \circ \mathcal{D}[\mathbf{take}\ c(a); P']\}$ is SN.

Now, we prove that $C \circ \mathcal{D}[\mathbf{take}\ c(a); P']$ is $[\![a : \wedge\overline{A}]\!]_\sigma$-preserving, for any a. Let $R \in [\![a : \wedge\overline{A}]\!]$. Applying i.h. to $P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma$ we conclude that $\mathbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\} \in [\![c : \mathbf{U}_e\ A]\!]$, which implies that $\mathbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\} \in [\![c : \mathbf{U}_e\ A]\!]$ and hence $\mathbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\}$ is $[\![a : \wedge\overline{A}]\!]$-preserving.

Therefore, $C \circ \mathcal{D}[\mathbf{take}\ c(a); P'] \in [\![c : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\mathbf{take}\ c(a); P' \in \mathcal{L}[\![\vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma]\!]$.

Case: [Tput]

$$\frac{P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma \qquad P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{put}\ c(a.P_1); P_2\ \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![c : \mathbf{U}_e\ A]\!] = S^{\perp}$, where

$$S = \{Q \mid Q \equiv_{\mathsf{c}} \mathbf{empty}\ c([\![a : \wedge\overline{A}]\!].)\}.$$

Let $C_1 \in [\![\Delta_1]\!], C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} \mathbf{empty}\ c([\![a : \wedge\overline{A}]\!].)$.

We prove that (H) $\mathbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\}$ is SN, by induction on $N(Q) + N(C_1) + N(C_2)$.

Suppose that $\mathbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\} \rightarrow_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q, C_1 or C_2.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H).So let us consider case (ii). Then

$$\begin{aligned}&\mathbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\}\\ \equiv_{\mathsf{c}}\ &\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\}\\ \equiv_{\mathsf{c}}\ &\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ \ \mathbf{put}\ c(a.C_1 \circ \mathcal{D}[P_1]); C_2 \circ \mathcal{D}[P_2]\}\\ \rightarrow_{\mathsf{c}}\ &\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ C_2 \circ \mathcal{D}[P_2]\}\ = R\ (*)\end{aligned}$$

I.h. applied to $P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![a : \wedge\overline{A}]\!]$, hence reduction step (*).

By Def. 32, $[\![c : \mathbf{S}_f\ \overline{A}]\!] = S^{\perp\perp}$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, hence **cell** $c(a.[\![a : \wedge\overline{A}]\!]) \in [\![c : \mathbf{S}_f\ \overline{A}]\!]$.

Applying i.h. to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that **cut** $\{Q \ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\}$ is SN.

Now, we prove that $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]$ is $[\![a : \wedge\overline{A}]\!]$-preserving, for any a. Applying i.h. to $P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma$ we conclude that $C_1 \circ \mathcal{D}[P_1] \in [\![a : \wedge\overline{A}]\!]$. Applying i.h. to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ we conclude that $C_2 \circ \mathcal{D}[P_2] \in [\![c : \mathbf{U}_f\ A]\!]$, which implies that $C_2 \circ \mathcal{D}[P_2]$ is $[\![a : \wedge\overline{A}]\!]$-preserving

Therefore, $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2] \in [\![c : \mathbf{U}_e\ A]\!]$.

By Lemma 26, $\mathbf{put}\ c(a.P_1); P_2 \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma]\!]$.

□

Theorem 5 (Strong Normalisation)**.** *If $P \vdash_\emptyset \emptyset; \emptyset$, then P is SN.*

Proof. Immediately by Lemma 27. □

9.4 Further Discussion and Related Work

To the best of our knowledge, our work is the first proposal that integrates affine shared state in the context of session-based interpretations of Linear Logic, and recursion, while still guaranteeing a strong normalisation result. The proof of the result is based in the technique of the linear logical relations, which we have scaled to accommodate shared mutable state, handling state interference compositionally.

Several works exploit the technique of logical relations to establish strong normalisation in the context of concurrent process calculi [164, 135, 18, 121]. [18] proves strong normalisation for a concurrent language with higher-order store with a type and effect system that stratifies memory into regions so as to preclude circularities (such as the Landin's Knot), which could cause divergence. Interestingly, in our type system this *stratification* is implicitly guaranteed by the acyclicity inherent to Linear Logic.

Linear logical relations for session-typed languages based on Intuitionistic Linear Logic were studied in [121], and further extended in [32] to accommodate behavioural polymorphism, in [146] to accommodate recursion, and in [149] to prove full abstraction of an encoding between System F and a polymorphic session calculus. In [121], the linear

logical predicates need to be explicitly defined for each session type constructor. On the other hand, in our proof, since we are working in the realm of Classical Linear Logic, we explore the relationship of duality on types [63, 9, 1]. As a consequence, in our case, the definition of the basic logical predicates is given explicitly for only half of the connectives, the remaining half being obtained by orthogonality.

Our linear logical predicates depend on the the auxiliary interference-sensitive reference cells, which internalise interference resultant from state sharing in their operational semantics (Section 9.2) and which play a crucial role by allowing us to reason about state sharing compositionally. Interestingly, [27] defines, in the context of spatial-behavioral types for shared concurrency, a logical predicate which is also sensitive to possible interferences on the shared store, the definition relies on an interference-sensitive reduction.

As already discussed in the introduction, reduction $\rightarrow$ in CLASS is not strongly normalising, essentially due to the idempotency of sum ($P \equiv P + P$). Therefore, we have established strong normalisation for the session-typed calculus CLASS with collapsing reduction $\rightarrow_c$ instead. However, this result should not be seen as being too restrictive, as this implies a sort of strong normalisation result for reduction $\rightarrow$ in CLASS, provided applications of the idempotency law are controlled. That is, we conjecture that it should be possible to simulate $\rightarrow$-reductions of a process P in CLASS by $\rightarrow_c$-reductions in multisets of processes $\{P_1, P_2, \ldots\}$ in CLASS. The crucial point is then that the multisets used for the simulation are finite, provided applications of idempotency are bounded. This kind of result, that relates a logically motivated reduction (in our case $\rightarrow$) with a non-confluent collapsing reduction (in our case $\rightarrow_c$) is established in [28] for a session-typed process calculus with abortable computations. On the other hand, the session-typed calculus CLASS is interesting of its own since it captures more faithfully the semantics of our practical interpreter implementation (Chapter 5), where sums are also not present.

10

Conclusion

In this thesis, we developed CLASS, a core session-based language with a lightweight substructural type system, that results from a principled extension of the propositions-as-types correspondence with second-order classical linear logic. The typing rules for the imperative fragment are inspired by those for the exponentials and sum connectives of differential linear logic (DiLL), in particular state sharing corresponds to the computational interpretation of cocontraction.

CLASS offers support for session-based communication, first-class higher-order reference cells with locks, dynamic state sharing, generic polymorphic algorithms, data abstraction and primitive recursion, the pure fragment was presented in Chapter 2 and the imperative fragment in Chapter 3.

CLASS expresses and types significant realistic programs, such as memory-efficient linked data structures (linked lists, binary search trees) that support updates in-place, shareable concurrent ADTs (counters, stacks, functional and imperative queues), resource synchronisation methods (fork-joins, barriers, dining philosophers) and generic shared corecursive protocols. All of these examples are guaranteed to be safe, purely by the logical correspondence, several of which were presented in Chapter 4.

The feasibility of our propositions-as-types approach is witnessed by the implementation of a type checker and interpreter, written in Java, which we presented in Chapter 5.

As a consequence of its propositions-as-types logical foundations, CLASS provides strong guarantees in a highly compositional way: well-typed CLASS programs never block when executing (Chapter 6), the outcome of a program is independent of the order in which instructions are executed (Chapter 7), each program has a normal reduced form that summarises its behaviour (Chapter 8), each program is guaranteed to always terminate (Chapter 9). Furthermore, we can reason about program behaviour in a very simple and and algebraic-like way as we have illustrated with some examples.

We will now point out some future work. The session-typed language CLASS is already quite expressive, as we have showed with several examples. The crisp conditions imposed by cocontraction - that two concurrent threads, at at any given time, share only a single cell - might appear, a priori, to be too limiting. But we have showed that its to possible

to group the state shared by two threads in a single resource, as we did, for example, in coding the solution to the dining philosophers problem (Example 4.3). However, in this solution, the symmetry-breaker philosopher needs to traverse the whole passive shared structure in order to acquire the first and last fork. An interesting research direction is to investigate further type disciplines for CLASS that relax this condition and that allow finer-grained resource-access policies to be expressed.

Arrays are fundamental blocks, used to give an efficient implementation of many other data structures, such as lists, hash tables and queues, to name a few, due to the constant-time random access to data elements they provide. CLASS allows us to code, albeit inefficiently, arrays, for example, as we do in *examples/state/arrays.clls*, by representing arrays as functions on natural-valued indices. It would be interesting to study how to provide support for primitive arrays with efficient constant-time access in the implementation, compatible to those typed in the formal model.

CLASS already integrates shared mutable state with higher-order polymorphism and inductive/coinductive types in the context of the propositions-as-types interpretation of linear logic. The further integration with dependent types [148, 92] would allows us to have a general theory for concurrent stateful session-based computation, based on linear logic. This would allow us to express resource invariants and possibly notions of abstract separation [93, 84, 52].

Overall, it would be very important to continue developing a practical full-fledged programming language based on CLASS, providing strong guarantees at compile time. Of course, this is an ambitious goal, that has been approached perhaps for decades. In 1972, Hoare [69] already mention that: "Parallel programs are particularly prone to time-dependent errors, which either cannot be detected by program testing nor by run-time checks. It is therefore very important that a high-level language designed for this purpose should provide complete security against time-dependent errors by means of a *compile-time* check". This work intends to offer a valuable contribution towards that general goal.

Bibliography

[1] S. Abramsky. "Computational Interpretations of Linear Logic". In: *Theoret. Comput. Sci.* 111.1–2 (1993), pp. 3–57 (cit. on pp. 5, 48, 143, 166).

[2] S. Abramsky, S. J. Gay, and R. Nagarajan. "Interaction categories and the foundations of typed concurrent programming". In: *NATO ASI DPD*. 1996, pp. 35–113 (cit. on p. 48).

[3] A. Ahmed, M. Fluet, and G. Morrisett. "A step-indexed model of substructural state". In: *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. 2005, pp. 78–91 (cit. on pp. 10, 70).

[4] A. Ahmed, M. Fluet, and G. Morrisett. "L^3: A Linear Language with Locations". In: *Fundam. Inf.* 77.4 (2007-12), pp. 397–449. ISSN: 0169-2968 (cit. on pp. 7, 70).

[5] J.-M. Andreoli. "Logic Programming with Focusing Proofs in Linear Logic". In: *J. Log. Comput.* 2.3 (1992), pp. 297–347 (cit. on p. 49).

[6] A. Asperti and L. Roversi. "Intuitionistic light affine logic". In: *ACM Transactions on Computational Logic (TOCL)* 3.1 (2002), pp. 137–175 (cit. on pp. 10, 58, 70).

[7] R. Atkey, S. Lindley, and J. G. Morris. "Conflation Confers Concurrency". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by S. Lindley et al. Cham: Springer International Publishing, 2016, pp. 32–55. ISBN: 978-3-319-30936-1 (cit. on pp. 7, 70, 73, 123).

[8] M. van Atten. "The Development of Intuitionistic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Summer 2022. Metaphysics Research Lab, Stanford University, 2022 (cit. on p. 3).

[9] D. Baelde. "Least and greatest fixed points in linear logic". In: *TOCL* 13.1 (2012-01) (cit. on pp. 20, 49, 143, 166).

[10] S. Balzer and F. Pfenning. "Manifest Sharing with Session Types". In: *Proc. ACM Program. Lang.* 1.ICFP (2017-08) (cit. on pp. 7, 11, 70, 71, 73, 76, 85, 95, 123).

[11] S. Balzer, B. Toninho, and F. Pfenning. "Manifest Deadlock-Freedom for Shared Session Types". In: *Programming Languages and Systems*. Ed. by L. Caires. Cham: Springer International Publishing, 2019, pp. 611–639. ISBN: 978-3-030-17184-1 (cit. on pp. 7, 65, 70, 71, 73, 109).

[12] H. P. Barendregt et al. *The lambda calculus*. Vol. 3. North-Holland Amsterdam, 1984 (cit. on p. 123).

[13] E. Beffara. "An Algebraic Process Calculus". In: *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*. LICS '08. USA: IEEE Computer Society, 2008, pp. 130–141. ISBN: 9780769531830 (cit. on p. 123).

[14] G. Bellin and P. Scott. "On the π-Calculus and Linear Logic". In: *Theoret. Comput. Sci.* 135.1 (1994), pp. 11–65 (cit. on pp. 5, 48).

[15] G. Berry and G. Boudol. "The chemical abstract machine". In: *Theoretical computer science* 96.1 (1992), pp. 217–248 (cit. on p. 23).

[16] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013 (cit. on p. 4).

[17] M. Boreale. "On the Expressiveness of Internal Mobility in Name-Passing Calculi". In: *Proceedings of the 7th International Conference on Concurrency Theory*. CONCUR '96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 163–178. ISBN: 3540616047 (cit. on p. 35).

[18] G. Boudol. "Typing termination in a higher-order concurrent imperative language". In: *Information and Computation* 208.6 (2010), pp. 716–736 (cit. on p. 165).

[19] A. Bove, P. Dybjer, and U. Norell. "A brief overview of Agda–a functional language with dependent types". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 73–78 (cit. on p. 4).

[20] G. Bracha et al. "Making the future safe for the past: Adding genericity to the Java programming language". In: *Acm sigplan notices* 33.10 (1998), pp. 183–200 (cit. on p. 4).

[21] S. Brookes and P. W. O'Hearn. "Concurrent separation logic". In: *ACM SIGLOG News* 3.3 (2016), pp. 47–65 (cit. on pp. 70, 76, 85).

[22] L. Caires and J. A. Pérez. "Multiparty Session Types Within a Canonical Binary Theory, and Beyond". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by E. Albert and I. Lanese. Cham: Springer International Publishing, 2016, pp. 74–95. ISBN: 978-3-319-39570-8 (cit. on p. 6).

[23] L. Caires and F. Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 - Concurrency Theory*. Ed. by P. Gastin and F. Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 222–236. ISBN: 978-3-642-15375-4 (cit. on pp. 6, 8, 12, 17, 19, 22, 24, 27, 35, 36, 48, 49, 65, 70, 71, 101, 108, 123, 125, 144).

[24] L. Caires, F. Pfenning, and B. Toninho. "Linear logic propositions as session types". In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423 (cit. on pp. 8, 17, 19, 22, 48, 70, 71, 108, 125, 142, 144).

[25] L. Caires and H. T. Vieira. "Conversation Types". In: *Programming Languages and Systems*. Ed. by G. Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 285–300. ISBN: 978-3-642-00590-9 (cit. on p. 108).

[26] L. Caires et al. *Relational Parametricity for Polymorphic Session Types*. Tech. rep. CMU-CS-12-108. Carnegie Mellon Univ., 2012 (cit. on p. 109).

[27] L. Caires. "Spatial-behavioral types for concurrency and resource control in distributed systems". In: *Theoretical Computer Science* 402.2-3 (2008), pp. 120–141 (cit. on p. 166).

[28] L. Caires and J. A. Pérez. "Linearity, Control Effects, and Behavioral Types". In: *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 229–259. ISBN: 9783662544334 (cit. on pp. 6, 10, 58, 70, 109, 166).

[29] L. Caires, F. Pfenning, and B. Toninho. "Linear logic propositions as session types". In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423 (cit. on p. 49).

[30] L. Caires, F. Pfenning, and B. Toninho. "Towards Concurrent Type Theory". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 1–12. ISBN: 9781450311205 (cit. on pp. 29, 33).

[31] L. Caires and J. C. Seco. "The Type Discipline of Behavioral Separation". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 275–286. ISBN: 9781450318327 (cit. on pp. 7, 70).

[32] L. Caires et al. "Behavioral Polymorphism and Parametricity in Session-Based Communication". In: *Proceedings of the 22nd European Conference on Programming Languages and Systems*. ESOP'13. Rome, Italy: Springer-Verlag, 2013, pp. 330–349. ISBN: 9783642370359 (cit. on pp. 6, 143, 165).

[33] M. Carbone, K. Honda, and N. Yoshida. "Structured Interactional Exceptions in Session Types". In: *CONCUR 2008*. Vol. 5201. LNCS. Springer, 2008, pp. 402–417 (cit. on pp. 10, 70).

[34] M. Carbone et al. "Coherence Generalises Duality: a logical explanation of multiparty session types". In: *27 International Conference on Concurrency Theory (CONCUR'16)*. Québec City, Canada, 2016-08 (cit. on p. 6).

[35] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 471–523 (cit. on pp. 18, 76, 79).

[36] R. F. Chen, S. Balzer, and B. Toninho. *Ferrite: A Judgmental Embedding of Session Types in Rust*. 2022 (cit. on pp. 5, 95).

[37] A. Church and J. B. Rosser. "Some properties of conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–482 (cit. on p. 111).

[38] D. G. Clarke, J. M. Potter, and J. Noble. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 48–64. ISBN: 1581130058 (cit. on pp. 5, 10, 70).

[39] E. Cooper et al. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects*. Ed. by F. S. de Boer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 266–296. ISBN: 978-3-540-74792-5 (cit. on p. 5).

[40] T. Coquand and G. Huet. "The calculus of constructions". PhD thesis. INRIA, 1986 (cit. on p. 4).

[41] H. B. Curry. "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590 (cit. on p. 3).

[42] H. B. Curry et al. *Combinatory logic*. Vol. 1. North-Holland Amsterdam, 1958 (cit. on p. 3).

[43] O. Dardha and S. J. Gay. "A New Linear Logic for Deadlock-Free Session-Typed Processes". In: *Foundations of Software Science and Computation Structures*. Ed. by C. Baier and U. Dal Lago. Cham: Springer International Publishing, 2018, pp. 91–109. ISBN: 978-3-319-89366-2 (cit. on p. 108).

[44] O. Dardha and J. A. Pérez. "Comparing deadlock-free session typed processes". In: *arXiv preprint arXiv:1508.06707* (2015) (cit. on p. 109).

[45] R. DeLine and M. Fähndrich. "Enforcing High-Level Protocols in Low-Level Software". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: Association for Computing Machinery, 2001, pp. 59–69. ISBN: 1581134142 (cit. on p. 5).

[46] R. Demangeon, D. Hirschkoff, and D. Sangiorgi. "Mobile processes and termination". In: *Semantics and Algebraic Specification*. Springer, 2009, pp. 250–273 (cit. on p. 143).

[47] F. Derakhshan and F. Pfenning. *Circular Proofs as Session-Typed Processes: A Local Validity Condition*. 2019 (cit. on p. 49).

[48] M. Dezani-Ciancaglini and U. de'Liguoro. "Sessions and Session Types: an Overview". In: *6th Intl Workshop on Web Services and Formal Methods WS-FM'09*. Lecture Notes in Computer Science. Springer-Verlag, 2010 (cit. on p. 5).

[49] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. "On progress for structured communications". In: *International Symposium on Trustworthy Global Computing*. Springer. 2007, pp. 257–275 (cit. on p. 108).

[50] E. W. Dijkstra. "Cooperating sequential processes". In: *The origin of concurrent programming*. Springer, 1968, pp. 65–138 (cit. on p. 7).

[51] E. W. Dijkstra. "Hierarchical ordering of sequential processes". In: *The origin of concurrent programming*. Springer, 1971, pp. 198–227 (cit. on pp. 14, 76, 82).

[52] T. Dinsdale-Young et al. "Views: Compositional Reasoning for Concurrent Programs". In: *Proceedings of POPL*. 2013-01 (cit. on p. 168).

[53] T. Ehrhard. "An introduction to differential linear logic: proof-nets, models and antiderivatives". In: *Mathematical Structures in Computer Science* 28.7 (2018), pp. 995–1060 (cit. on pp. 10, 12, 15, 18, 60, 65, 68, 70, 73, 109).

[54] T. Ehrhard and L. Regnier. "Differential Interaction Nets". In: *Theor. Comput. Sci.* 364.2 (2006), pp. 166–195 (cit. on pp. 18, 123).

[55] T. Ehrhard and L. Regnier. "The differential lambda-calculus". In: *Theoretical Computer Science* 309.1-3 (2003), pp. 1–41 (cit. on p. 123).

[56] A. Filinski. "Linear continuations". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1992, pp. 27–38 (cit. on p. 38).

[57] S. Fowler et al. "Exceptional asynchronous session types: session types without tiers". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29 (cit. on pp. 5, 10, 70).

[58] S. J. Gay and V. T. Vasconcelos. "Linear type theory for asynchronous session types". In: *Journal of Functional Programming* 20.1 (2010), pp. 19–50 (cit. on p. 48).

[59] G. Gentzen. "Investigations into Logical Deduction". In: *American Philosophical Quarterly* 1.4 (1964), pp. 288–306. ISSN: 00030481 (cit. on p. 124).

[60] E. Giachino, N. Kobayashi, and C. Laneve. "Deadlock analysis of unbounded process networks". In: *International Conference on Concurrency Theory*. Springer. 2014, pp. 63–77 (cit. on p. 109).

[61] J.-Y. Girard. "Linear Logic". In: *Theoret. Comput. Sci.* 50.1 (1987), pp. 1–102 (cit. on p. 48).

[62] J.-Y. Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Éditeur inconnu, 1972 (cit. on p. 4).

[63] J.-Y. Girard. "Linear logic". In: *Theoretical computer science* 50.1 (1987), pp. 1–101 (cit. on pp. 20, 49, 143, 166).

[64] J.-Y. Girard. "Linear logic: its syntax and semantics". In: *London Mathematical Society Lecture Note Series* (1995), pp. 1–42 (cit. on pp. 5, 124).

[65] J.-Y. Girard. "On the unity of logic". In: *Annals of pure and applied logic* 59.3 (1993), pp. 201–217 (cit. on p. 48).

[66] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. USA: Cambridge University Press, 1989. ISBN: 0521371813 (cit. on pp. 5, 9, 24, 43).

[67] M. Hennessy and R. Milner. "Algebraic Laws for Nondeterminism and Concurrency". In: *J. ACM* 32.1 (1985), pp. 137–161 (cit. on pp. 15, 68, 74).

[68] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (1974), pp. 549–557 (cit. on pp. 7, 76, 85).

[69] C. A. R. Hoare. "Towards a theory of parallel programming". In: *The origin of concurrent programming*. Springer, 1972, pp. 231–244 (cit. on p. 168).

[70] J. S. Hodas. "Logic programming in intutionistic linear logic: Theory, design and implementation". In: *PhD Thesis, University of Pennsylvania, Department of Computer and Information Science* (1994) (cit. on p. 142).

[71] J. S. Hodas and D. Miller. "Logic programming in a fragment of intuitionistic linear logic". In: *Information and computation* 110.2 (1994), pp. 327–365 (cit. on pp. 92, 94).

[72] K. Honda, N. Yoshida, and M. Carbone. "Multiparty asynchronous session types". In: *POPL*. 2008, pp. 273–284 (cit. on p. 5).

[73] K. Honda. "Types for dyadic interaction". In: *CONCUR'93*. Ed. by E. Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–523. ISBN: 978-3-540-47968-0 (cit. on pp. 5, 22, 48).

[74] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language primitives and type discipline for structured communication-based programming". In: *Programming Languages and Systems*. Ed. by C. Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-69722-0 (cit. on pp. 5, 22).

[75] K. Honda and N. Yoshida. "On reduction-based process semantics". In: *Theoretical Computer Science* 151.2 (1995), pp. 437–486 (cit. on p. 23).

[76] W. A. Howard. "The formulae-as-types notion of construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by J. P. Seldin and J. R. Hindley. Academic Press, 1980, pp. 479–490 (cit. on pp. 3, 48).

[77] R. Hu, N. Yoshida, and K. Honda. "Session-based distributed programming in Java". In: *European Conference on Object-Oriented Programming*. Springer. 2008, pp. 516–541 (cit. on p. 5).

[78] P. Hudak and J. H. Fasel. "A gentle introduction to Haskell". In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52 (cit. on p. 4).

[79] A. Igarashi et al. "Gradual session types". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017), pp. 1–28 (cit. on p. 5).

[80] J. Jacobs, S. Balzer, and R. Krebbers. "Connectivity graphs: a method for proving deadlock freedom based on separation logic". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–33 (cit. on p. 70).

[81] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. "Session types for Rust". In: *Proceedings of the 11th acm sigplan workshop on generic programming*. 2015, pp. 13–22 (cit. on p. 5).

[82] S. P. Jones. "Beautiful concurrency". In: *Beautiful Code: Leading Programmers Explain How They Think* (2007), pp. 385–406 (cit. on p. 109).

[83] S. P. Jones, A. Gordon, and S. Finne. "Concurrent Haskell". In: *POPL*. Vol. 96. Citeseer. 1996, pp. 295–308 (cit. on pp. 13, 74).

[84] R. Jung et al. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20 (cit. on p. 168).

[85] R. Jung et al. "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 637–650 (cit. on p. 7).

[86] S. Klabnik and C. Nichols. "The Rust Programming Language". In: (2021) (cit. on pp. 5, 13, 75, 86).

[87] N. Kobayashi. "A new type system for deadlock-free processes". In: *International Conference on Concurrency Theory*. Springer. 2006, pp. 233–247 (cit. on p. 108).

[88] N. Kobayashi. "A type system for lock-free processes". In: *Information and Computation* 177.2 (2002), pp. 122–159 (cit. on p. 108).

[89] N. Kobayashi and C. Laneve. "Deadlock analysis of unbounded process networks". In: *Information and Computation* 252 (2017), pp. 48–70 (cit. on p. 109).

[90] W. Kokke, F. Montesi, and M. Peressotti. "Better late than never: a fully-abstract semantics for classical processes". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29 (cit. on p. 48).

[91] W. Kokke, J. G. Morris, and P. Wadler. "Towards Races in Linear Logic". In: *Coordination Models and Languages*. Ed. by H. Riis Nielson and E. Tuosto. Cham: Springer International Publishing, 2019, pp. 37–53. ISBN: 978-3-030-22397-7 (cit. on pp. 7, 17, 65, 70, 73, 108, 123).

[92] N. R. Krishnaswami, P. Pradic, and N. Benton. "Integrating linear and dependent types". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 17–30 (cit. on p. 168).

[93] N. R. Krishnaswami et al. "Superficially Substructural Types". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 41–54. ISBN: 9781450310543 (cit. on p. 168).

[94] N. Lagaillardie, R. Neykova, and N. Yoshida. "Stay safe under panic: Affine rust programming with multiparty session types". In: *arXiv preprint arXiv:2204.13464* (2022) (cit. on p. 70).

[95] D. Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000 (cit. on p. 93).

[96] X. Leroy. "The CompCert C verified compiler: Documentation and user's manual". PhD thesis. Inria, 2020 (cit. on p. 4).

[97] S. Lindley and J. G. Morris. "Embedding session types in Haskell". In: *ACM SIGPLAN Notices* 51.12 (2016), pp. 133–145 (cit. on p. 5).

[98] S. Lindley and J. G. Morris. "Embedding session types in Haskell". In: *9th International Symposium on Haskell, Haskell 2016*. 2016, pp. 133–145 (cit. on p. 94).

[99] S. Lindley and J. G. Morris. "Talking bananas: structural recursion for session types". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by J. Garrigue, G. Keller, and E. Sumii. ACM, 2016, pp. 434–447 (cit. on pp. 6, 49, 109).

[100] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[101] S. Lu et al. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *ACM SIGARCH Computer Architecture News*. Vol. 36. 1. ACM. 2008, pp. 329–339 (cit. on p. 7).

[102] S. Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. " O'Reilly Media, Inc.", 2013 (cit. on pp. 13, 74, 81).

[103] L. Meertens. "Algorithmics: Towards programming as a mathematical activity". In: (1986) (cit. on p. 4).

[104] F. Militão, J. Aldrich, and L. Caires. "Aliasing control with view-based typestate". In: *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*. 2010, pp. 1–7 (cit. on p. 70).

[105] R. Milner et al. "A calculus of communicating systems". In: (1980) (cit. on p. 48).

[106] R. Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999 (cit. on p. 111).

[107] R. Milner. "Functions as Processes". In: *Math. Struct. Comput. Sci.* 2.2 (1992), pp. 119–141 (cit. on p. 23).

[108] R. Milner, J. Parrow, and D. Walker. "A calculus of mobile processes, i". In: *Information and computation* 100.1 (1992), pp. 1–40 (cit. on p. 48).

[109] J. C. Mitchell and G. D. Plotkin. "Abstract types have existential type". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10.3 (1988), pp. 470–502 (cit. on pp. 4, 17, 18, 42, 43, 76, 79).

[110] D. Mostrous and V. T. Vasconcelos. "Affine Sessions". In: *Proc. of COORDINATION 2014*. Vol. 8459. LNCS. Springer, 2014, pp. 115–130 (cit. on pp. 10, 70).

[111] L. d. Moura et al. "The Lean theorem prover (system description)". In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388 (cit. on p. 4).

[112] A. Nanevski, J. G. Morrisett, and L. Birkedal. "Hoare type theory, polymorphism and separation". In: *J. Funct. Program.* 18.5-6 (2008), pp. 865–911 (cit. on pp. 7, 70).

[113] N. Ng, N. Yoshida, and K. Honda. "Multiparty Session C: Safe parallel programming with message optimisation". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2012, pp. 202–218 (cit. on pp. 5, 94).

[114] N. Ng et al. "Safe parallel programming with session java". In: *International Conference on Coordination Languages and Models*. Springer. 2011, pp. 110–126 (cit. on p. 5).

[115] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002 (cit. on p. 4).

[116] P. W. O'Hearn and J. C. Reynolds. "From Algol to polymorphic linear lambda-calculus". In: *J. ACM* 47.1 (2000), pp. 167–223 (cit. on pp. 7, 70).

[117] M. Odersky et al. "An overview of the Scala programming language". In: (2004) (cit. on p. 4).

[118] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998 (cit. on p. 77).

[119] M. Pagani and P. Tranquilli. "Parallel Reduction in Resource Lambda-Calculus". In: *Programming Languages and Systems*. Ed. by Z. Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 226–242. ISBN: 978-3-642-10672-9 (cit. on p. 123).

[120] M. Parigot. "Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction". In: *LPAR'92*. 1992, pp. 190–201 (cit. on p. 4).

[121] J. A. Pérez et al. "Linear logical relations and observational equivalences for session-based concurrency". In: *Information and Computation* 239 (2014), pp. 254–302 (cit. on pp. 20, 123, 143, 165).

[122] F. Pfenning. "Structural Cut Elimination". In: *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. LICS '95. USA: IEEE Computer Society, 1995, p. 156. ISBN: 0818670506 (cit. on pp. 20, 40, 49, 125, 142).

[123] F. Pfenning. *Structural Cut Elimination in Linear Logic*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1994 (cit. on pp. 20, 125, 142).

[124] F. Pfenning and D. Griffith. "Polarized Substructural Session Types". In: *Foundations of Software Science and Computation Structures*. Ed. by A. Pitts. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–22. ISBN: 978-3-662-46678-0 (cit. on p. 71).

[125] G. D. Plotkin. "A Powerdomain Construction". In: *SIAM J. Comput.* 5.3 (1976), pp. 452–487 (cit. on p. 74).

[126] J. Polakow. "Embedding a full linear lambda calculus in Haskell". In: *ACM SIGPLAN Notices* 50.12 (2015), pp. 177–188 (cit. on p. 94).

[127] R. Pucella and J. A. Tov. "Haskell session types with (almost) no class". In: *Proceedings of the first ACM SIGPLAN symposium on Haskell*. 2008, pp. 25–36 (cit. on p. 5).

[128] Z. Qian, G. A. Kavvos, and L. Birkedal. "Client-Server Sessions in Linear Logic". In: *Proc. ACM Program. Lang.* 5.ICFP (2021-08) (cit. on pp. 7, 11, 70, 72, 73, 108, 123).

[129] G. Restall. *An introduction to substructural logics*. Routledge, 2002 (cit. on p. 5).

[130] J. C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium*. Springer. 1974, pp. 408–425 (cit. on p. 4).

[131] P. Rocha and L. Caires. "Propositions-as-Types and Shared State (Artifact)". In: (2021-05) (cit. on p. 18).

[132] P. Rocha and L. Caires. *A Propositions-as-Types System for Shared State*. Tech. rep. NOVA Laboratory for Computer Science and Informatics, 2021-06 (cit. on p. 18).

[133] P. Rocha and L. Caires. *CLASS: Classical Linear Logical with Affine Shared State (Implementation)*. 2022 (cit. on pp. 19, 88).

[134] P. Rocha and L. Caires. "Propositions-as-Types and Shared State". In: *Proc. ACM Program. Lang.* 5.ICFP (2021-08) (cit. on pp. 18, 64, 70, 71).

[135] D. Sangiorgi. "Termination of processes". In: *Math. Struct. in Comp. Sci.* 16.1 (2006), pp. 1–39 (cit. on p. 165).

[136] D. Sangiorgi. "The Name Discipline of Uniform Receptiveness". In: *Theor. Comput. Sci.* 221.1-2 (1999), pp. 457–493 (cit. on pp. 6, 42).

[137] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. USA: Cambridge University Press, 2001. ISBN: 0521781779 (cit. on pp. 27, 101).

[138] A. Scalas and N. Yoshida. "Lightweight session programming in scala". In: (2016) (cit. on p. 94).

[139] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006 (cit. on p. 5).

[140] G. Steele. *Common LISP: the language*. Elsevier, 1990 (cit. on p. 4).

[141] J. Sunshine et al. "First-Class State Change in Plaid". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 713–732. ISBN: 9781450309400 (cit. on p. 7).

[142] M. Takahashi. "Parallel reductions in λ-calculus". In: *Information and computation* 118.1 (1995), pp. 120–127 (cit. on p. 123).

[143] B. Toninho. "A Logical Foundation for Session-Based Concurrent Computation". PhD thesis. NOVA School of Science and Technology, 2015-05 (cit. on p. 49).

[144] B. Toninho, L. Caires, and F. Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration". In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 350–369. ISBN: 978-3-642-37036-6 (cit. on p. 6).

[145] B. Toninho, L. Caires, and F. Pfenning. *Dependent Session Types via Intuitionistic Linear Type Theory*. Tech. rep. CMU-CS-11-139. Carnegie Mellon Univ., 2011 (cit. on p. 109).

[146] B. Toninho, L. Caires, and F. Pfenning. "Corecursion and Non-divergence in Session-Typed Processes". In: *TGC 2014*. Ed. by M. Maffei and E. Tuosto. Vol. 8902. Lecture Notes in Computer Science. Springer, 2014, pp. 159–175 (cit. on pp. 6, 109, 143, 165).

[147] B. Toninho, L. Caires, and F. Pfenning. "Corecursion and non-divergence in session-typed processes". In: *International Symposium on Trustworthy Global Computing*. Springer. 2014, pp. 159–175 (cit. on p. 49).

[148] B. Toninho, L. Caires, and F. Pfenning. "Dependent Session Types via Intuitionistic Linear Type Theory". In: *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*. PPDP '11. Odense, Denmark: Association for Computing Machinery, 2011, pp. 161–172. ISBN: 9781450307765 (cit. on pp. 6, 168).

[149] B. Toninho and N. Yoshida. "On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings". In: *ACM Trans. Program. Lang. Syst.* 43.2 (2021-06). ISSN: 0164-0925 (cit. on pp. 6, 9, 24, 43, 111, 165).

[150] J. A. Tov and R. Pucella. "Practical Affine Types". In: *POPL 2011*. 2011, pp. 447–458 (cit. on pp. 10, 70).

[151] P. Tranquilli. "Confluence of pure differential nets with promotion". In: *International Workshop on Computer Science Logic*. Springer. 2009, pp. 500–514 (cit. on p. 123).

[152] P. Tranquilli. "Intuitionistic differential nets and lambda-calculus". In: *Theoretical Computer Science* 412.20 (2011), pp. 1979–1997 (cit. on p. 123).

[153] V. T. Vasconcelos. "Fundamentals of session types". In: *Information and Computation* 217 (2012), pp. 52–70. ISSN: 0890-5401 (cit. on p. 5).

[154] H. T. Vieira and V. T. Vasconcelos. "Typing progress in communication-centred systems". In: *International Conference on Coordination Languages and Models*. Springer. 2013, pp. 236–250 (cit. on p. 108).

[155] A. L. Voinea, O. Dardha, and S. J. Gay. "Resource sharing via capability-based multiparty session types". In: *International Conference on Integrated Formal Methods*. Springer. 2019, pp. 437–455 (cit. on p. 70).

[156] P. Wadler. "Linear Types can Change the World!" In: *Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, 1990*. Ed. by M. Broy. North-Holland, 1990, p. 561 (cit. on pp. 5, 7).

[157] P. Wadler. "Propositions as Sessions". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 273–286. ISBN: 9781450310543 (cit. on pp. 6, 8, 12, 17, 19, 22, 24, 27, 35, 48, 49, 65, 70, 71, 108, 123, 125, 144).

[158] P. Wadler. "Propositions as sessions". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 384–418 (cit. on p. 36).

[159] P. Wadler. "Propositions as types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84 (cit. on p. 5).

[160] P. Wadler. "Propositions as types". In: *Commun. ACM* 58.12 (2015), pp. 75–84 (cit. on p. 48).

[161] P. Wadler. *Recursive types for free*. 1990 (cit. on pp. 43, 44, 94).

[162] M. Willsey, R. Prabhu, and F. Pfenning. "Design and implementation of Concurrent C0". In: *arXiv preprint arXiv:1701.04929* (2017) (cit. on p. 95).

[163] N. Yoshida. "Graph types for monadic mobile processes". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1996, pp. 371–386 (cit. on p. 108).

[164] N. Yoshida, M. Berger, and K. Honda. "Strong normalisation in the π-calculus". In: *Information and Computation* 191.2 (2004), pp. 145–202 (cit. on p. 165).

A

Type Preservation

Notation

Before presenting the complete proofs of type preservation for structural congruence $\equiv$ and reduction $\longrightarrow$ we introduce some handy notations that make the presentation of the proofs more succinct.

State Flavours

We introduce two state flavours, namely e (empty) and f (full). If $\mathcal{X}$ is a flavour, then the metavariable type $\mathbf{S}_{\mathcal{X}}\ A$ denotes either the full cell modality $\mathbf{S}_f\ A$, if $\mathcal{X} = f$, or either the empty cell modality $\mathbf{S}_e\ A$, if $\mathcal{X} = e$. Similarly, $\mathbf{U}_{\mathcal{X}}\ A$ denotes either $\mathbf{U}_f\ A$, if $\mathcal{X} = f$, or $\mathbf{U}_e\ A$, if $\mathcal{X} = e$. Two flavours can be combined through a partial binary operation $\oplus$, defined by

$$f \oplus f \triangleq f \quad f \oplus e \triangleq e \quad e \oplus f \triangleq e$$

The operation $\oplus$ is commutative and associative, furthermore the value of an expression $\mathcal{X}_1 \oplus \ldots \oplus \mathcal{X}_n$ is either f, whenever all the $\mathcal{X}_i$ are f; or e, in case one and only one of the $\mathcal{X}_i$ is e.

With this notation at hand, we can succinctly group all the typing rules for sharing ([Tsh], [TshL], [TshR]) in a single typing rule schema

$$\frac{P \vdash_\eta \Delta', c : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma \quad Q \vdash_\eta \Delta, c : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma \quad \mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}}{\mathbf{share}\ c\ \{P \mid\mid Q\} \vdash_\eta \Delta', \Delta, c : \mathbf{U}_{\mathcal{X}}\ A; \Gamma}\ [\text{TshX}]$$

Type Inversion

Often, in the following proofs of type preservation and progress, we appeal to inversion principles for the typing relation. By inspecting the principal form, i.e. the outermost constructor, of a process P for which a typing judgement $P \vdash_\eta \Delta; \Gamma$ holds we can infer some particularities of the typing contexts Δ and Γ. This works because, by inspecting the principal form of the process P, we can infer which was the typing rule that was applied to the root of a derivation tree for $P \vdash_\eta \Delta; \Gamma$. For example, in a derivation for

par $\{P_1 \mid\mid P_2\} \vdash_\eta \Delta; \Gamma$ the last rule has to be [Tmix], from which we conclude that there there are Δ_1, Δ_2 s.t. $\Delta = \Delta_1, \Delta_2$, $P_1 \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$. To make the presentation succinct, in the following proofs, we refer to the corresponding inversion principle associated with a typing rule adding the superscript -1 to the typing rule name. So, for [Tmix], it would be [Tmix^{-1}].

Auxiliary Lemmas

We state some auxiliary lemmas which are used during the proofs of type preservation. The first lemma states that every subprocess of a well-typed process is well-typed. Furthermore, if we replace a subprocess Q of a process a well-typed process P by a subprocess Q' that types with the same typing context as Q, then the resulting substitution types with same typing context as P.

Lemma 28. *Suppose $C[P] \vdash_\eta \Delta; \Gamma$, for some process context C. Then, there exists Δ', Γ' s.t.*

- $P \vdash_\eta \Delta'; \Gamma$.
- *For all $Q \vdash_\eta \Delta'; \Gamma'$, $C[Q] \vdash_\eta \Delta'; \Gamma'$.*

Proof. If $C = -$, then simply pick $\Delta' = \Delta$ and $\Gamma' = \Gamma$. The hypothesis for the cases in which $C \neq -$ is established by induction on the typing derivation tree that establishes $C[P] \vdash_\eta \Delta; \Gamma$.

We illustrate with some cases.

Case: [T1].

From $C[P] =$ **close** x we conclude that $C = -$ and $P =$ **close** x. Holds vacuously.

Case: [Tmix].

We have $C[P] \vdash_\eta \Delta_1, \Delta_2; \Gamma$, $C[P] =$ **par** $\{P_1 \mid\mid P_2\}$, $P_1 \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$.

Since $C[P] =$ **par** $\{P_1 \mid\mid P_2\}$, either (i) $C =$ **par** $\{C' \mid\mid R\}$ or (ii) $C =$ **par** $\{R \mid\mid C'\}$.

We consider (i) holds. The analysis is similar for (ii).

By applying the i.h. $C'[P] \vdash_\eta \Delta_1; \Gamma$ we infer the existence of Δ_1', Γ' s.t.

(a) $P \vdash_\eta \Delta_1'; \Gamma'$.

(b) $C'[Q] \vdash_\eta \Delta_1; \Gamma$ for all $Q' \vdash_\eta \Delta_1'; \Gamma'$.

Let $Q' \vdash_\eta \Delta_1'; \Gamma'$. From (b), $C'[Q] \vdash_\eta \Delta_1; \Gamma$.

Applying [Tmix] to $C'[Q] \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$ yields $C[Q] =$ **par** $\{C'[Q] \mid\mid P_2\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$.

□

Some formulations of the session-based interpretations of Linear Logic (cf. Wadler's CP) have explicit typing rules for weakening and contraction of the exponential modalities !, ?. In CLASS weakening and contraction are absorbed by the unrestricted typing context: we can adjoin an arbitrary formula in Γ (Lemma 29([Tweaken]) or substitute the use of one formula for another (Lemma 29([Tcontract]). Furthermore, we have a kind of *reverse* weakening principle: if a formula is not being used in a derivation, we can remove it from the unrestricted context (Lemma 29([Tstrength])), this property is often referred to as *strengthening*.

Lemma 29. *The following principles hold:*

[Tweaken] *If* $P \vdash_\eta \Delta; \Gamma$ *and* $x \notin dom(\Delta) \cup dom(\Gamma)$, *then* $P \vdash_\eta \Delta; \Gamma, x : A$.

[Tcontract] *If* $P \vdash_\eta \Delta; \Gamma, x : A, y : A$, *then* $\{x/y\}P \vdash_{\{x/y\}\eta} \Delta; \Gamma, x : A$.

[Tstrength] *If* $P \vdash_\eta \Delta; \Gamma, x : A$ *and* $x \notin \mathsf{fn}(P)$, *then* $P \vdash_\eta \Delta; \Gamma$.

Proof. **[Tweaken]** By induction on derivation tree for $P \vdash_\eta \Delta; \Gamma$. We illustrate with some cases.

Case: [T**0**].

We have the conclusion $\mathbf{0} \vdash_\eta \emptyset; \Gamma$. By applying [T**0**] we obtain $\mathbf{0} \vdash_\eta \emptyset; \Gamma, x : A$.

Case: [Tmix].

We have the conclusion $\mathbf{par}\ \{P_1 \,||\, P_2\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ from the premisses $P_1 \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$.

Applying i.h. to $P_1 \vdash_\eta \Delta_1; \Gamma$ and $P_2 \vdash_\eta \Delta_2; \Gamma$ yields $P_1 \vdash_\eta \Delta_1; \Gamma, x : A$ and $P_2 \vdash_\eta \Delta_2; \Gamma, x : A$, respectively.

Applying [Tmix] to $P_1 \vdash_\eta \Delta_1; \Gamma, x : A$ and $P_2 \vdash_\eta \Delta_2; \Gamma, x : A$ yields $\mathbf{par}\ \{P_1 \,||\, P_2\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, x : A$.

[Tcontract] By induction on derivation tree for $P \vdash_\eta \Delta; \Gamma$. We illustrate with some cases.

Case: [Tmix].

We have the conclusion $\mathbf{par}\ \{P_1 \,||\, P_2\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, x : A, y : A$ from the premisses $P_1 \vdash_\eta \Delta_1; \Gamma, x : A, y : A$ and $P_2 \vdash_\eta \Delta_2; \Gamma, x : A, y : A$.

Applying i.h. to $P_1 \vdash_\eta \Delta_1; \Gamma, x : A, y : A$ and $P_2 \vdash_\eta \Delta_2; \Gamma, x : A, y : A$ yields $\{x/y\}P_1 \vdash_\eta \Delta_1; \Gamma, x : A$ and $\{x/y\}P_2 \vdash_\eta \Delta_2; \Gamma, x : A$, respectively.

Applying [Tmix] to $\{x/y\}P_1 \vdash_\eta \Delta_1; \Gamma, x : A$ and $\{x/y\}P_2 \vdash_\eta \Delta_2; \Gamma, x : A$ yields $\mathbf{par}\ \{\{x/y\}P_1 \,||\, \{x/y\}P_2\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, x : A$.

Finally, note that $\{x/y\}(\mathbf{par}\ \{P_1 \,||\, P_2\}) = \mathbf{par}\ \{\{x/y\}P_1 \,||\, \{x/y\}P_2\}$.

Case: [Tcall].

There are three cases to consider, depending on wether the subject z of the call action is x, y or neither x nor y.

Case: $z \neq x, y$.

We have the conclusion $\mathbf{call}\ z(w); Q \vdash_\eta \Delta; \Gamma$, from the premiss $Q \vdash_\eta \Delta, w : B; \Gamma, x : A, y : A, z : B$.

Applying i.h. to $Q \vdash_\eta \Delta, w : B; \Gamma, x : A, y : A, z : B$ yields $\{x/y\}Q \vdash_\eta \Delta, w : B; \Gamma, x : A, z : B$.

Applying [Tcall] to $\{x/y\}Q \vdash_\eta \Delta; \Gamma, x : A, , z : B$ yields $\mathbf{call}\ z(w); \{x/y\}Q \vdash_\eta \Delta; \Gamma, x : A, z : B$.

Finally, note that $\{x/y\}(\mathbf{call}\ z(w); Q) = \mathbf{call}\ z(w); \{x/y\}Q$.

Case: $z = x$.

We have the conclusion $\mathbf{call}\ x(w); Q \vdash_\eta \Delta; \Gamma$, from the premiss $Q \vdash_\eta \Delta, w : A; \Gamma, x : A, y : A$.

Applying i.h. to $Q \vdash_\eta \Delta, w : A; \Gamma, x : A, y : A$ yields $\{x/y\}Q \vdash_\eta \Delta, w : A; \Gamma, x : A$.

Applying [Tcall] to $\{x/y\}Q \vdash_\eta \Delta, w : A; \Gamma, x : A$ yields $\mathbf{call}\ x(w); \{x/y\}Q \vdash_\eta \Delta; \Gamma, x : A$.

Finally, note that $\{x/y\}(\mathbf{call}\ x(w); Q) = \mathbf{call}\ x(w); \{x/y\}Q$.

Case: $z = y$.

We have the conclusion $\mathbf{call}\ y(w); Q \vdash_\eta \Delta; \Gamma$, from the premiss $Q \vdash_\eta \Delta, w : A; \Gamma, x : A, y : A$.

Applying i.h. to $Q \vdash_\eta \Delta, w : A; \Gamma, x : A, y : A$ yields $\{x/y\}Q \vdash_\eta \Delta, w : A; \Gamma, x : A$.

Applying [Tcall] to $\{x/y\}Q \vdash_\eta \Delta, w : A; \Gamma, x : A$ (this time on x) yields $\mathbf{call}\ x(w); \{x/y\}Q \vdash_\eta \Delta; \Gamma, x : A$.

Finally, note that $\{x/y\}(\mathbf{call}\ y(w); Q) = \mathbf{call}\ x(w); \{x/y\}Q$.

[Tstrength] Similar to [Tweaken].

□

The proof of type preservation also depends on a couple of auxiliary properties, which we will introduce now. The first (Lemma 30(1)) states that the domain of the linear typing context with which a process P types is always the same.

To introduce the second property (Lemma 30(2)) we need the following definition. Let Δ, Δ' be two partial maps from names to types. We say that Δ *is contained in* Δ' *up to usage flavours* iff the following hold

(1) if $x : A \in \Delta$ and $A \neq \mathbf{U}_\chi\ B$, then $x : A \in \Delta'$.

(2) if $x : \mathbf{U}_\chi\ B$, then $x : \mathbf{U}_\mathcal{Y}\ B \in \Delta'$ for some usage flavour $\mathcal{Y}$.

We say that Δ *and* Δ' *are the same up to usage flavours* iff Δ is contained in Δ' up to to usage flavours and vice-versa: Δ' is contained in Δ up to usage flavours.

Lemma 30. *The following properties hold*

(1) *If* $P \vdash_\eta \Delta;\Gamma$ *and* $P \vdash_\eta \Delta';\Gamma'$ *then* $dom(\Delta) = dom(\Delta')$.

(2) *Suppose* $P \vdash_\eta \Delta;\Gamma$, $P \vdash_\eta \Delta';\Gamma$ *and let* Δ, Δ' *be the same up to usage flavours. Then,* $\Delta = \Delta'$.

Proof. **(1)** By induction on P. We illustrate with some cases.

Case: $P = \mathbf{0}$.

Applying $[\mathrm{T}\mathbf{0}^{-1}]$ to $\mathbf{0} \vdash_\eta \Delta;\Gamma$ yields $\Delta = \emptyset$.

Applying $[\mathrm{T}\mathbf{0}^{-1}]$ to $\mathbf{0} \vdash_\eta \Delta;'\Gamma'$ yields $\Delta' = \emptyset$.

Then, $\mathrm{dom}(\Delta) = \emptyset = \mathrm{dom}(\Delta')$.

Case $P = \mathbf{fwd}\ x\ y$.

By applying $[\mathrm{Tfwd}^{-1}]$ to $\mathbf{fwd}\ x\ y \vdash_\eta \Delta;\Gamma$ we infer the existence of A s.t. $\Delta = x : \overline{A}, y : A$.

By applying $[\mathrm{Tfwd}^{-1}]$ to $\mathbf{fwd}\ x\ y \vdash_\eta \Delta';\Gamma$ we infer the existence of B s.t. $\Delta' = x : \overline{B}, y : B$.

Then, $\mathrm{dom}(\Delta) = \{x, y\} = \mathrm{dom}(\Delta')$.

Case: $P = \mathbf{par}\ \{P_1 \mid\mid P_2\}$.

By applying $[\mathrm{Tmix}^{-1}]$ to $\mathbf{par}\ \{P_1 \mid\mid P_2\} \vdash_\eta \Delta;\Gamma$ we infer the existence of Δ_1, Δ_2 s.t. $\Delta = \Delta_1, \Delta_2$, $P_1 \vdash_\eta \Delta_1;\Gamma$ and $P_2 \vdash_\eta \Delta_2;\Gamma$.

By applying $[\mathrm{Tmix}^{-1}]$ to $\mathbf{par}\ \{P_1 \mid\mid P_2\} \vdash_\eta \Delta';\Gamma'$ we infer the existence of Δ'_1, Δ'_2 s.t. $\Delta' = \Delta'_1, \Delta'_2$, $P_1 \vdash_\eta \Delta'_1;\Gamma'$ and $P_2 \vdash_\eta \Delta'_2;\Gamma'$.

Applying i.h. to $P_1 \vdash_\eta \Delta_1;\Gamma$ and $P_1 \vdash_\eta \Delta'_1;\Gamma'$ yields $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta'_1)$.

Applying i.h. to $P_2 \vdash_\eta \Delta_2;\Gamma'$ and $P_2 \vdash_\eta \Delta'_2;\Gamma'$ yields $\mathrm{dom}(\Delta_2) = \mathrm{dom}(\Delta'_2)$.

Then, $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2) = \mathrm{dom}(\Delta'_1) \cup \mathrm{dom}(\Delta'_2) = \mathrm{dom}(\Delta')$.

Case: $P = ?x;P'$.

By applying $[\mathrm{T}?^{-1}]$ to $?x;P \vdash_\eta \Delta;\Gamma$ we infer the existence of Δ_0, A s.t $\Delta = \Delta_0, x : ?A$ and $P \vdash_\eta \Delta_0;\Gamma, x : A$.

By applying $[\mathrm{T}?^{-1}]$ to $?x;P \vdash_\eta \Delta';\Gamma'$ we infer the existence of Δ'_0, B s.t $\Delta = \Delta'_0, x :?B$ and $P \vdash_\eta \Delta'_0;\Gamma', x : B$.

Applying i.h. to $P \vdash_\eta \Delta_0;\Gamma, x : A$ and $P \vdash_\eta \Delta'_0;\Gamma', x : B$ yields $\mathrm{dom}(\Delta_0) = \mathrm{dom}(\Delta'_0)$.

Then, $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta_0) \cup \{x\} = \mathrm{dom}(\Delta'_0) \cup \{x\} = \mathrm{dom}(\Delta')$.

(2) By induction on P and case analysis on its principal form. We illustrate with some cases.

Case $P = \mathbf{fwd}\ x\ y$.

By $[\mathrm{Tfwd}^{-1}]$ and $\mathbf{fwd}\ x\ y \vdash \Delta;\Gamma$ we conclude that $\Delta = x : A, y : \overline{A}$ for some type A. By $[\mathrm{Tfwd}^{-1}]$ and $\mathbf{fwd}\ x\ y \vdash \Delta';\Gamma$ we conclude that $\Delta' = x : B, y : \overline{B}$ for some type B.

Either A or $\overline{A}$ is not an usage modality. Suppose w.l.o.g. that $A \neq \mathbf{U}_\chi\ B$. Then $A = B$ and, as consequence, $\overline{A} = \overline{B}$.

Case $P = \textbf{share}\ x\ \{P_1 \mid\mid P_2\}$.

By [Tsh^{-1}] and $\textbf{share}\ x\ \{P_1 \mid\mid P_2\} \vdash_\eta \Delta;\Gamma$ we conclude that exists $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ s.t. (1) $P_1 \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A;\Gamma$, (2) $P_2 \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$, (3) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ and (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$.

By [Tsh^{-1}] and $\textbf{share}\ x\ \{P_1 \mid\mid P_2\} \vdash_\eta \Delta';\Gamma$ we conclude that exists $\Delta'_1, \Delta'_2, A', \mathcal{X}'_1, \mathcal{X}'_2, \mathcal{X}'$ s.t. (1′) $P_1 \vdash_\eta \Delta'_1, x : \mathbf{U}_{\mathcal{X}'_1}\ A';\Gamma$, (2′) $P_2 \vdash_\eta \Delta'_2, x : \mathbf{U}_{\mathcal{X}'_2}\ A';\Gamma$, (3′) $\Delta' = \Delta'_1, \Delta'_2, x : \mathbf{U}'_{\mathcal{X}}\ A'$ and (4′) $\mathcal{X}'_1 \oplus \mathcal{X}'_2 = \mathcal{X}'$.

From (3), (3′) and since Δ, Δ' are the same up to usage flavours we obtain $A = A'$. Furthermore, since $\Delta_1 = \Delta \upharpoonright (\mathsf{fn}(P_1) \setminus \{x\})$ and $\Delta'_1 = \Delta' \upharpoonright (\mathsf{fn}(P_1) \setminus \{x\})$, we conclude that Δ_1, Δ'_1 are the same up to usage flavours. Similarly, we conclude that Δ_2, Δ'_2 are the same up to usage flavours.

Applying the i.h. to P_1, (1) and (1′) yields $\Delta_1 = \Delta'_1$ and $\mathcal{X}_1 = \mathcal{X}'_1$. Applying the i.h. to P_2, (2) and (2′) yields $\Delta_2 = \Delta'_2$ and $\mathcal{X}_2 = \mathcal{X}'_2$.

Therefore, $\mathcal{X} = \mathcal{Y}$ and $\Delta = \Delta'$.

□

We conclude this section with a couple of auxiliary results that state how substitution (name by name, type variable by type, process variable by corecursive process definition) affect the typing relation.

Lemma 31. *The following properties hold*

(1) *If* $P \vdash_\eta \Delta;\Gamma$ *and* $x \notin dom(\Delta) \cup dom(\Gamma)$, *then* $\{x/y\}P \vdash_\eta \{x/y\}(\Delta;\Gamma)$.

(2) *If* $P \vdash_\eta \Delta;\Gamma$, *then* $\{A/X\}P \vdash_{\{A/X\}\eta} \{A/X\}(\Delta;\Gamma)$.

(3) *Suppose* $\textbf{corec}\ Y(z,\vec{w});P\ [z,\vec{w}] \vdash_\eta \Delta, z : \nu X.\ A;\Gamma$, $\eta' = \eta'', Y(z,\vec{w}) \mapsto \Delta, z : X;\Gamma$ *for some* η'' *which extends* η, *and suppose* $Q \vdash_{\eta'} \Delta';\Gamma'$. *Then,* $\{\textbf{corec}\ Y(z,\vec{w});P/Y\}Q \vdash_{\eta''} \{\nu X.\ A/X\}(\Delta';\Gamma')$.

Proof. Properties (1) and (2) are by induction on a derivation for $P \vdash_\eta \Delta;\Gamma$.

Property (3) is by induction on a derivation for $Q \vdash_{\eta'} \Delta', z : B;\Gamma'$. The only way of introducing the type variable X in the context $\Delta';\Gamma'$, with which Q types, is by appealing to rule [Tvar] on process variable Y. Consequently, if process variable Y does not occur free in Q, then the property holds trivially since $\{\textbf{corec}\ Y(z,\vec{w});P/Y\}Q = Q$ and $\{\nu X.\ A/X\}(\Delta';\Gamma') = \Delta';\Gamma'$. We illustrate the proof with some cases:

Case: [Tvar].

Then

$$\frac{\eta' = \eta'', Y(z,\vec{w}) \mapsto \Delta, z : X;\Gamma}{Y(x,\vec{y}) \vdash_{\eta'} \{\vec{y}/\vec{w}\}(\Delta, x : X;\Gamma)}\ [\text{Tvar}]$$

where $Q = Y(x,\vec{y})$.

By def.

$$\{\textbf{corec}\ Y(z,\vec{w});P/Y\}Y(x,\vec{y}) = \ \textbf{corec}\ Y(z,\vec{w});P\ [x,\vec{y}]$$

Since, by hypothesis $\textbf{corec}\ Y(z,\vec{w});P\ [z,\vec{w}] \vdash_\eta \Delta, z : \nu X.\ A;\Gamma$ and η'' extends η, then $\textbf{corec}\ Y(z,\vec{w});P\ [z,\vec{w}] \vdash_{\eta''} \Delta, z : \nu X.\ A;\Gamma$.

By name renaming, $\textbf{corec}\ Y(z,\vec{w});P\ [x,\vec{y}] \vdash_{\eta''} \{\vec{y}/\vec{w}\}(\Delta, x : \nu X.\ A;\Gamma)$.

Case: [Tmix].

Then

$$\frac{Q_1 \vdash_{\eta'} \Delta_1';\Gamma' \quad Q_2 \vdash_{\eta'} \Delta_2';\Gamma'}{\textbf{par}\ \{Q_1 \mid\mid Q_2\}\ \vdash_{\eta'} \Delta_1', \Delta_2', ;\Gamma'}\ [\text{Tmix}]$$

where $Q = \textbf{par}\ \{Q_1 \mid\mid Q_2\}$ and $\Delta' = \Delta_1', \Delta_2'$.

By def.

$$\{\textbf{corec}\,Y(z,\vec{w});P/Y\}(\textbf{par}\,\{Q_1 \mid\mid Q_2\}) = \textbf{par}\,(\{\textbf{corec}\,Y(z,\vec{w});P/Y\}Q_1) \mid\mid (\{\textbf{corec}\,Y(z,\vec{w});P/Y\}Q_2)\}$$

Applying i.h. to $Q_1 \vdash_{\eta'} \Delta_1';\Gamma'$ yields (a) $\{\textbf{corec}\,Y(z,\vec{w});P/Y\}Q_1 \vdash_{\eta''} \{\nu X.\,A/X\}(\Delta_1';\Gamma')$.

Applying i.h. to $Q_2 \vdash_{\eta'} \Delta_2';\Gamma'$ yields (b) $\{\textbf{corec}\,Y(z,\vec{w});P/Y\}Q_2 \vdash_{\eta''} \{\nu X.\,A/X\}(\Delta_2';\Gamma')$.

Applying [Tmix] to (a) and (b) yields

$$\{\textbf{corec}\ Y(z,\vec{w});P/Y\}(\textbf{par}\ \{Q_1 \mid\mid Q_2\}) \vdash_{\eta''} \{\nu X.\ A/X\}(\Delta_1', \Delta_2';\Gamma')$$

□

Type Preservation

We prove type preservation Theorem 1. We start with the proof of type preservation for structural congruence in 1(1)) and then we move to the proof of type preservation for reduction 1(2)).

Theorem 1(1)). *If $P \vdash_\eta \Delta;\Gamma$ and $P \equiv Q$, then $Q \vdash_\eta \Delta;\Gamma$.*

Proof. By induction on a derivation tree for $P \equiv Q$ and case analysis on the root rule. We consider an axiomatisation of $\equiv$ equivalent to Definition 13 but in which we drop rule [symm] $P \equiv Q \ \supset\ Q \equiv P$ and assume that each commuting conversion holds from left-to-right and right-to-left.

Case: [refl], $P \equiv P$.

Follows immediately.

Case: [trans], $P \equiv Q$ and $Q \equiv R \ \supset\ P \equiv R$.

(1) $Q \vdash_\eta \Delta;\Gamma$ (i.h., $P \vdash_\eta \Delta;\Gamma$ and $P \equiv Q$)
(2) $R \vdash_\eta \Delta;\Gamma$ (i.h., (1) and $Q \equiv R$)

Case: [cong], $P \equiv Q \supset C[P] \equiv C[Q]$.

(1) $P \vdash_\eta \Delta'; \Gamma'$, for some Δ', Γ' (Lemma 28 and $C[P] \vdash_\eta \Delta; \Gamma$)
(2) $Q \vdash_\eta \Delta'; \Gamma'$ (i.h., (1) and $P \equiv Q$)
(3) $C[Q] \vdash_\eta \Delta; \Gamma$ (Lemma 28, (1), (2) and $C[P] \vdash_\eta \Delta; \Gamma$)

Case: [fwd], $\mathbf{fwd}\ x\ y \equiv \mathbf{fwd}\ y\ x$.

(1) $\Delta = x : \overline{A}, y : A$ ([Tfwd^{-1}] and $\mathbf{fwd}\ x\ y \vdash_\eta \Delta; \Gamma$)
(2) $\mathbf{fwd}\ y\ x \vdash_\eta y : A, x : \overline{A}; \Gamma$ ([Tfwd])
(3) $\mathbf{fwd}\ y\ x \vdash_\eta \Delta; \Gamma$ ((1) and (2))

Case: [M], $\mathbf{par}\ \{P \parallel Q\} \equiv \mathbf{par}\ \{Q \parallel P\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1; \Gamma$ (3) $Q \vdash_\eta \Delta_2; \Gamma$, for some Δ_1, Δ_2
([Tmix^{-1}] and $\mathbf{par}\ \{P \parallel Q\} \vdash_\eta \Delta; \Gamma$)
(4) $\mathbf{par}\ \{Q \parallel P\} \vdash_\eta \Delta_2, \Delta_1; \Gamma$ ([Tmix], (3) and (2))
(5) $\mathbf{par}\ \{Q \parallel P\} \vdash_\eta \Delta; \Gamma$ ((1) and (4))

Case: [C], $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \equiv \mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (3) $Q \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \vdash_\eta \Delta; \Gamma$)
(4) $\mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\} \vdash_\eta \Delta_2, \Delta_1; \Gamma$ ([Tcut], (3) and (2))
(5) $\mathbf{cut}\ \{Q\ \ |x : \overline{A}|\ \ P\} \vdash_\eta \Delta; \Gamma$ ((1) and (4))

Case: [Sh], $\mathbf{share}\ x\ \{P \parallel Q\} \equiv \mathbf{share}\ x\ \{Q \parallel P\}$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ (2) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$
(3) $Q \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some Δ_1, Δ_2
([Tsh^{-1}] and $\mathbf{share}\ x\ \{P \parallel Q\} \vdash_\eta \Delta; \Gamma$)
(5) $\mathcal{X}_2 \oplus \mathcal{X}_1 = \mathcal{X}$ ($\oplus$ is commutative and (4))
(6) $\mathbf{share}\ x\ \{Q \parallel P\} \vdash_\eta \Delta_2, \Delta_1, x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ([Tsh], (3),(2) and (5))
(7) $\mathbf{share}\ x\ \{Q \parallel P\} \vdash_\eta \Delta; \Gamma$ ((1) and (6))

Case: [Sm], $P + Q \equiv Q + P$.

(1) $P \vdash_\eta \Delta; \Gamma$ (2) $Q \vdash_\eta \Delta; \Gamma$ ([Tsum^{-1}] and $P + Q \vdash_\eta \Delta; \Gamma$)
(3) $Q + P \vdash_\eta \Delta; \Gamma$ ([Tsum], (2) and (1))

Case: [MM] left-to-right, $\mathbf{par}\ \{P \parallel (\mathbf{par}\ \{Q \parallel R\})\} \equiv \mathbf{par}\ \{(\mathbf{par}\ \{P \parallel Q\}) \parallel R\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1; \Gamma$ (3) $\mathbf{par}\ \{Q \parallel R\} \vdash_\eta \Delta_2; \Gamma$, for some Δ_1, Δ_2 ([Tmix^{-1}] and $\mathbf{par}\ \{P \parallel (\mathbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2 = \Delta_{21}, \Delta_{22}$ (5) $Q \vdash_\eta \Delta_{21}; \Gamma$ (6) $R \vdash_\eta \Delta_{22}; \Gamma$, for some Δ_{21}, Δ_{22} ([Tmix^{-1}] and (3))
(7) $\mathbf{par}\ \{P \parallel Q\} \vdash_\eta \Delta_1, \Delta_{21}; \Gamma$ ([Tmix], (2) and (5))
(8) $\mathbf{par}\ \{(\mathbf{par}\ \{P \parallel Q\}) \parallel R\} \vdash_\eta \Delta_1, \Delta_{21}, \Delta_{22}; \Gamma$ ([Tmix], (7) and (6))
(9) $\Delta_1, \Delta_{21}, \Delta_{22} = \Delta$ ((1) and (4))
(10) $\mathbf{par}\ \{(\mathbf{par}\ \{P \parallel Q\}) \parallel R\} \vdash_\eta \Delta; \Gamma$ ((8) and (9))

Case: [MM] right-to-left, $\mathbf{par}\ \{(\mathbf{par}\ \{P \parallel Q\}) \parallel R\} \equiv \mathbf{par}\ \{P \parallel (\mathbf{par}\ \{Q \parallel R\})\}$. Similar to case [MM] left-to-right.

Case: [SmSm] left-to-right, $P + (Q + R) \equiv (P + Q) + R$.

(1) $P \vdash_\eta \Delta; \Gamma$ (2) $Q + R \vdash_\eta \Delta; \Gamma$ ([Tsum^{-1}] and $P + (Q + R) \vdash_\eta \Delta; \Gamma$)
(3) $Q \vdash_\eta \Delta; \Gamma$ (4) $R \vdash_\eta \Delta; \Gamma$ ([Tsum^{-1}] and (2))
(5) $P + Q \vdash_\eta \Delta; \Gamma$ ([Tsum], (1) and (3))
(6) $(P + Q) + R \vdash_\eta \Delta; \Gamma$ ([Tsum], (5) and (4))

Case: [SmSm] right-to-left, $(P + Q) + R \equiv P + (Q + R)$. Similar to case [SmSm] left-to-right.

Case: [CM] left-to-right, $\mathbf{cut}\ \{P\ |x : A|\ (\mathbf{par}\ \{Q \parallel R\})\} \equiv \mathbf{par}\ \{(\mathbf{cut}\ \{P\ |x : A|\ Q\}) \parallel R\}$, $x \in \mathrm{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (3) $\mathbf{par}\ \{Q \parallel R\} \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$, for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{P\ |x : A|\ (\mathbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2, x : \overline{A} = \Delta_{21}, \Delta_{22}$ (5) $Q \vdash_\eta \Delta_{21}; \Gamma$ (6) $R \vdash_\eta \Delta_{22}; \Gamma$, for some Δ_{21}, Δ_{22} ([Tmix^{-1}] and (3))
(7) $\Delta_{21} = \Delta'_{21}, x : \overline{A}$, for some Δ'_{21} ((4), (5) and $x \in \mathrm{fn}(Q)$)
(8) $Q \vdash_\eta \Delta'_{21}, x : \overline{A}$ ((5) and (7))
(9) $\mathbf{cut}\ \{P\ |x : A|\ Q\} \vdash_\eta \Delta_1, \Delta'_{21}; \Gamma$ ([Tcut], (2), (8))
(10) $\mathbf{par}\ \{(\mathbf{cut}\ \{P\ |x : A|\ Q\}) \parallel R\} \vdash_\eta \Delta_1, \Delta'_{21}, \Delta_{22}; \Gamma$ ([Tmix], (9) and (6))
(11) $\Delta_1, \Delta'_{21}, \Delta_{22} = \Delta$ ((1), (4) and (7))
(12) $\mathbf{par}\ \{(\mathbf{cut}\ \{P\ |x : A|\ Q\}) \parallel R\} \vdash_\eta \Delta; \Gamma$ ((10) and (11))

Case: [CM] right-to-left, $\mathbf{par}\ \{(\mathbf{cut}\ \{P\ |x : A|\ Q\}) \parallel R\} \equiv \mathbf{cut}\ \{P\ |x : A|\ (\mathbf{par}\ \{Q \parallel R\})\}$, $x \in \mathrm{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{cut}\ \{P\ |x : A|\ Q\} \vdash_\eta \Delta_1; \Gamma$ (3) $R \vdash_\eta \Delta_2; \Gamma$, for some Δ_1, Δ_2 ([Tmix^{-1}] and $\mathbf{par}\ \{(\mathbf{cut}\ \{P\ |x : A|\ Q\}) \parallel R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1 = \Delta_{11}, \Delta_{12}$ (5) $P \vdash_\eta \Delta_{11}, x : A; \Gamma$ (6) $Q \vdash_\eta \Delta_{12}, x : \overline{A}; \Gamma$, for some Δ_{11}, Δ_{12} ([Tcut^{-1}] and (2))

(7) $\mathbf{par}\ \{Q \parallel R\} \vdash_\eta \Delta_{12}, x : \overline{A}, \Delta_2; \Gamma$ ([Tmix], (6) and (3))
(8) $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta_{11}, \Delta_{12}, \Delta_2; \Gamma$ ([Tcut], (5) and (7))
(9) $\Delta_{11}, \Delta_{12}, \Delta_2 = \Delta$ ((4) and (1))
(10) $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta; \Gamma$ ((8) and (9))

Case: [CC] left-to-right,

$\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut}\ \{Q \ |y : B|\ R\})\} \equiv \mathbf{cut}\ \{(\mathbf{cut}\ \{P \ |x : A|\ Q\}) \ |y : B|\ R\},\ x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (3) $\mathbf{cut}\ \{Q \ |y : B|\ R\} \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$, for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut}\ \{Q \ |y : B|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2, x : \overline{A} = \Delta_{21}, \Delta_{22}$ (5) $Q \vdash_\eta \Delta_{21}, y : B; \Gamma$ (6) $R \vdash_\eta \Delta_{22}, y : \overline{B}; \Gamma$, for some Δ_{21}, Δ_{22} ([Tcut^{-1}] and (3))
(7) $\Delta_{21} = \Delta'_{21}, x : \overline{A}$, for some Δ'_{21} ((4), (5) and $x \in \mathsf{fn}(Q)$)
(8) $Q \vdash_\eta \Delta'_{21}, x : \overline{A}, y : B; \Gamma$ ((5) and (7))
(9) $\mathbf{cut}\ \{P \ |x : A|\ Q\} \vdash_\eta \Delta_1, \Delta'_{21}, y : B; \Gamma$ ([Tcut], (2), (8))
(10) $\mathbf{cut}\ \{(\mathbf{cut}\ \{P \ |x : A|\ Q\}) \ |y : B|\ R\} \vdash_\eta \Delta_1, \Delta'_{21}, \Delta_{22}; \Gamma$ ([Tcut], (9) and (6))
(11) $\Delta_1, \Delta'_{21}, \Delta_{22} = \Delta$ ((1), (4) and (7))
(12) $\mathbf{cut}\ \{(\mathbf{cut}\ \{P \ |x : A|\ Q\}) \ |y : B|\ R\} \vdash_\eta \Delta; \Gamma$ ((10) and (11))

Case: [CC] right-to-left,

$\mathbf{cut}\ \{(\mathbf{cut}\ \{P \ |x : A|\ Q\}) \ |y : B|\ R\} \equiv \mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut}\ \{Q \ |y : B|\ R\})\},\ x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{cut}\ \{P \ |x : A|\ Q\} \vdash_\eta \Delta_1, y : B; \Gamma$ (3) $R \vdash_\eta \Delta_2, y : \overline{B}; \Gamma$, for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{(\mathbf{cut}\ \{P \ |x : A|\ Q\}) \ |y : B|\ R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1, y : B = \Delta_{11}, \Delta_{12}$ (5) $P \vdash_\eta \Delta_{11}, x : A\Gamma$ (6) $Q \vdash_\eta \Delta_{12}, x : \overline{A}; \Gamma$, for some Δ_{11}, Δ_{12} ([Tcut^{-1}] and (2))
(7) $\Delta_{12} = \Delta'_{12}, y : B$, for some Δ'_{12} ((4), (6) and $y \in \mathsf{fn}(Q)$)
(8) $Q \vdash_\eta \Delta'_{12}, y : B, x : \overline{A}; \Gamma$ ((6) and (7))
(9) $\mathbf{cut}\ \{Q \ |y : B|\ R\} \vdash_\eta \Delta'_{12}, x : \overline{A}, \Delta_2; \Gamma$ ([Tcut], (8) and (3))
(10) $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut}\ \{Q \ |y : B|\ R\})\} \vdash_\eta \Delta_{11}, \Delta'_{12}, \Delta_2; \Gamma$ ([Tcut], (5) and (9))
(11) $\Delta_{11}, \Delta'_{12}, \Delta_2 = \Delta$ ((1), (4) and (7))
(12) $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut}\ \{Q \ |y : B|\ R\})\} \vdash_\eta \Delta; \Gamma$ ((10) and (11))

Case: [CC!] left-to-right,

$\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut!}\ \{y.Q \ |z : B|\ R\})\} \equiv \mathbf{cut!}\ \{y.Q \ |z : B|\ (\mathbf{cut}\ \{P \ |x : A|\ R\})\},\ z \notin \mathsf{fn}(P)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (3) $\mathbf{cut!}\ \{y.Q \ |z : B|\ R\} \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$, for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{P \ |x : A|\ (\mathbf{cut!}\ \{y.Q \ |z : B|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $Q \vdash_\eta y : B; \Gamma$ (5) $R \vdash_\eta \Delta_2, x : \overline{A}; \Gamma, z : \overline{B}$ ([Tcut!$^{-1}$] and (3))
(6) $P \vdash_\eta \Delta_1, x : A; \Gamma, z : \overline{B}$ (Lemma 29([Tweaken]), (2) and $z \notin \mathsf{fn}(P)$)

(7) $\mathbf{cut}\ \{P \ \ |x : A|\ \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, z : \overline{B}$ ([Tcut], (6) and (5))
(8) $\mathbf{cut!}\ \{y.Q \ |z : B|\ (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut!], (4) and (7))
(9) $\mathbf{cut!}\ \{y.Q \ |z : B|\ (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\})\} \vdash_\eta \Delta; \Gamma$ ((1) and (8))

Case: [CC!] right-to-left,

$\mathbf{cut!}\ \{y.Q \ |z : B|\ (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\})\} \equiv \mathbf{cut}\ \{P \ \ |x : A|\ \ (\mathbf{cut!}\ \{y.Q \ |z : B|\ R\})\},\ z \notin \mathsf{fn}(P)$.

(1) $Q \vdash_\eta y : B; \Gamma$ (2) $\mathbf{cut}\ \{P \ \ |x : A|\ \ R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.Q \ |z : B|\ (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $P \vdash_\eta \Delta_1, x : A; \Gamma, z : \overline{B}$ (5) $R \vdash_\eta \Delta_2, x : \overline{A}; \Gamma, z : \overline{B}$, for some Δ_1, Δ_2
([Tcut!$^{-1}$] and (2))
(6) $\mathbf{cut!}\ \{y.Q \ |z : B|\ R\} \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$ ([Tcut!], (1) and (5))
(7) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (Lemma 29([Tstrength], (4) and $z \notin \mathsf{fn}(P)$)
(8) $\mathbf{cut}\ \{P \ \ |x : A|\ \ (\mathbf{cut!}\ \{y.Q \ |z : B|\ R\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (7) and (5))
(9) $\mathbf{cut}\ \{P \ \ |x : A|\ \ (\mathbf{cut!}\ \{y.Q \ |z : B|\ R\})\} \vdash_\eta \Delta; \Gamma$ ((3) and (8))

Case: [C!M] left-to-right, $\mathbf{cut!}\ \{y.P \ |x : A|\ (\mathbf{par}\ \{Q \ || \ R\})\} \equiv \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P \ |x : A|\ Q\}) \ || \ R\}, x \notin \mathsf{fn}(R)$.

(1) $P \vdash_\eta y : A; \Gamma$ (2) $\mathbf{par}\ \{Q \ || \ R\} \vdash_\eta \Delta; \Gamma, x : A$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.P \ |x : A|\ (\mathbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $Q \vdash_\eta \Delta_1; \Gamma, x : A$ (5) $R \vdash_\eta \Delta_2; \Gamma, x : A$, for some Δ_1, Δ_2
([Tmix^{-1}] and (2))
(5) $\mathbf{cut!}\ \{y.P \ |x : A|\ Q\} \vdash_\eta \Delta_1; \Gamma$ ([Tcut!], (1) and (4))
(6) $R \vdash_\eta \Delta_2; \Gamma$ (Lemma 29([Tstrength]), (5) and $x \notin \mathsf{fn}(R)$)
(7) $\mathbf{par}\ \{(\mathbf{cut!}\ \{y.P \ |x : A|\ Q\}) \ || \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tmix], (5) and (6))
(8) $\mathbf{par}\ \{(\mathbf{cut!}\ \{y.P \ |x : A|\ Q\}) \ || \ R\} \vdash_\eta \Delta; \Gamma$ ((3) and (7))

Case: [C!M] right-to-left, $\mathbf{par}\ \{(\mathbf{cut!}\ \{y.P \ |x : A|\ Q\}) \ || \ R\} \equiv \mathbf{cut!}\ \{y.P \ |x : A|\ (\mathbf{par}\ \{Q \ || \ R\})\}, x \notin \mathsf{fn}(R)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{cut!}\ \{y.P \ |x : A|\ Q\} \vdash_\eta \Delta_1; \Gamma$ (3) $R \vdash_\eta \Delta_2; \Gamma$
([Tmix^{-1}] and $\mathbf{par}\ \{(\mathbf{cut!}\ \{y.P \ |x : A|\ Q\}) \ || \ R\} \vdash_\eta \Delta; \Gamma$)
(4) $P \vdash_\eta y : A; \Gamma$ (5) $Q \vdash_\eta \Delta_1; \Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(6) $R \vdash_\eta \Delta_2; \Gamma, x : \overline{A}$ (Lemma 29([Tweaken]) and (3))
(7) $\mathbf{par}\ \{Q \ || \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, x : \overline{A}$ ([Tmix], (5) and (6))
(8) $\mathbf{cut!}\ \{y.P \ |x : A|\ (\mathbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut!], (4) and (7))
(9) $\mathbf{cut!}\ \{y.P \ |x : A|\ (\mathbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta; \Gamma$ ((1) and (8))

Case: [C!C!] left-to-right,

$\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{cut!}\ \{w.Q\ |z : B|\ R\})\} \equiv \mathbf{cut!}\ \{w.Q\ |z : B|\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\}, x \notin \mathsf{fn}(Q), z \notin \mathsf{fn}(P)$.

(1) $P \vdash_\eta y : A; \Gamma$ (2) $\mathbf{cut!}\ \{w.Q\ |z : B|\ R\} \vdash_\eta \Delta; \Gamma, x : \overline{A}$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{cut!}\ \{w.Q\ |z : B|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $Q \vdash_\eta w : B; \Gamma, x : \overline{A}$ (4) $R \vdash_\eta \Delta; \Gamma, x : \overline{A}, z : \overline{B}$ ([Tcut!$^{-1}$] and (2))
(5) $P \vdash_\eta y : A; \Gamma, z : \overline{B}$ (Lemma 29([Tweaken]), (1) and $z \notin \mathsf{fn}(P)$)
(6) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$ ([Tcut!], (5) and (4))
(7) $Q \vdash_\eta w : B; \Gamma$ (Lemma 29([Tstrength]), (3) and $x \notin \mathsf{fn}(Q)$
(8) $\mathbf{cut!}\ \{w.Q\ |z : B|\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$ ([Tcut!], (7) and (6))

Case: [C!C!] right-to-left,

$\mathbf{cut!}\ \{w.Q\ |z : B|\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \equiv \mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{cut!}\ \{w.Q\ |z : B|\ R\})\}, x \notin \mathsf{fn}(Q), z \notin \mathsf{fn}(P)$.

(1) $Q \vdash_\eta w : B; \Gamma$ (2) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{w.Q\ |z : B|\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $P \vdash_\eta y : A; \Gamma, z : \overline{B}$ (4) $R \vdash_\eta \Delta; \Gamma, z : \overline{B}, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(5) $Q \vdash_\eta w : B; \Gamma, x : \overline{A}$ (Lemma 29([Tweaken]), (1) and $x \notin \mathsf{fn}(Q)$)
(6) $\mathbf{cut!}\ \{w.Q\ |z : B|\ R\} \vdash_\eta \Delta; \Gamma, x : \overline{A}$ ([Tcut!], (5) and (4))
(7) $P \vdash_\eta y : A; \Gamma$ (Lemma 29([Tstrength]), (3) and $z \notin \mathsf{fn}(P)$)
(8) $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{cut!}\ \{w.Q\ |z : B|\ R\})\} \vdash_\eta \Delta; \Gamma$ ([Tcut!], (7) and (6))

Case: [CSh] left-to-right, $\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{share}\ y\ \{Q\ ||\ R\}\} \equiv \mathbf{share}\ y\ \{\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\ ||\ R\},\ x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A; \Gamma$ (3) $\mathbf{share}\ y\ \{Q\ ||\ R\} \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{P\ \ |x : A|\ \ (\mathbf{share}\ y\ \{Q\ ||\ R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2, x : \overline{A} = \Delta_{21}, \Delta_{22}, y : \mathbf{U}_\mathcal{X}\ B$ (5) $Q \vdash_\eta \Delta_{21}, y : \mathbf{U}_{\mathcal{X}_1}\ B; \Gamma$
(6) $R \vdash_\eta \Delta_{22}, y : \mathbf{U}_{\mathcal{X}_2}\ B; \Gamma$ (7) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_{21}, \Delta_{22}, B, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ ([Tsh^{-1}] and (3))
(8) $\Delta_{21} = \Delta'_{21}, x : \overline{A}$, for some Δ'_{21} ((4), (5) and $x \in \mathsf{fn}(Q)$)
(9) $Q \vdash_\eta \Delta'_{21}, x : \overline{A}, y : \mathbf{U}_{\mathcal{X}_1}\ B; \Gamma$ ((5) and (8))
(10) $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\ \vdash_\eta \Delta_1, \Delta'_{21}, y : \mathbf{U}_{\mathcal{X}_1}\ B; \Gamma$ ([Tcut], (2), (9))
(11) $\mathbf{share}\ y\ \{(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})\ ||\ R\} \vdash_\eta \Delta_1, \Delta'_{21}, \Delta_{22}, y : \mathbf{U}_\mathcal{X}\ B; \Gamma$ ([Tsh], (10), (6) and (7))
(12) $\Delta_1, \Delta'_{21}, \Delta_{22}, y : \mathbf{U}_\mathcal{X}\ B\ = \Delta$ ((1), (4) and (8))
(13) $\mathbf{share}\ y\ \{(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\})\ ||\ R\} \vdash_\eta \Delta; \Gamma$ ((11) and (12))

Case: [CSh] right-to-left, $\mathbf{share}\ y\ \{\mathbf{cut}\ \{P\ |x : A|\ Q\}\ ||\ R\} \equiv \mathbf{cut}\ \{P\ |x : A|\ \mathbf{share}\ y\ \{Q\ ||\ R\}\},\ x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2, y : \mathbf{U}_\mathcal{X}\ B$ (2) $\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \vdash_\eta \Delta_1, y : \mathbf{U}_{\mathcal{X}_1}\ B; \Gamma$
(3) $R \vdash_\eta \Delta_2, y : \mathbf{U}_{\mathcal{X}_2}\ B; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, B, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$

([Tsh^{-1}] and **share** y {**cut** $\{P \ |x : A|\ Q\} \ || \ R\} \vdash_\eta \Delta;\Gamma$)
(5) $\Delta_1, y : \mathbf{U}_{\mathcal{X}_1} B = \Delta_{11}, \Delta_{12}$ (6) $P \vdash_\eta \Delta_{11}, x : A;\Gamma$ (7) $Q \vdash_\eta \Delta_{12}, x : \overline{A};\Gamma$, for some Δ_{11}, Δ_{12} ([Tcut^{-1}] and (2))
(8) $\Delta_{12} = \Delta'_{12}, y : \mathbf{U}_{\mathcal{X}_1} B$, for some Δ'_{12} ((5), (7) and $y \in \mathsf{fn}(Q)$)
(9) $Q \vdash_\eta \Delta'_{12}, y : \mathbf{U}_{\mathcal{X}_1} B, x : \overline{A};\Gamma$ ((7) and (8))
(10) **share** y $\{Q \ || \ R\} \vdash_\eta \Delta'_{12}, x : \overline{A}, \Delta_2, y : \mathbf{U}_{\mathcal{X}} B;\Gamma$ ([Tsh], (9), (3) and (4))
(11) **cut** $\{P \ |x : A|$ (**cut** $\{Q \ |y : B|\ R\})\} \vdash_\eta \Delta_{11}, \Delta'_{12}, \Delta_2, y : \mathbf{U}_{\mathcal{X}} B;\Gamma$ ([Tcut], (6) and (10))
(12) $\Delta_{11}, \Delta'_{12}, \Delta_2, y : \mathbf{U}_{\mathcal{X}} B = \Delta$ ((1), (5) and (8))
(13) **cut** $\{P \ |x : A|$ (**share** y $\{Q \ || \ R\})\} \vdash_\eta \Delta;\Gamma$ ((11) and (12))

Case: [ShM] left-to-right, **share** x $\{P \ || \ (\textbf{par}\ \{Q \ || \ R\})\} \equiv \textbf{par}\ \{\textbf{share}\ x\ \{P \ || \ Q\} \ || \ R\}$, $x \in \mathrm{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}} A$ (2) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1} A;\Gamma$
(3) **par** $\{Q \ || \ R\} \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2} A;\Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ ([Tsh^{-1}] and **share** x $\{P \ || \ (\textbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta;\Gamma$)
(5) $\Delta_2, x : \mathbf{U}_{\mathcal{X}_2} A = \Delta_{21}, \Delta_{22}$ (6) $Q \vdash_\eta \Delta_{21};\Gamma$ (7) $R \vdash_\eta \Delta_{22};\Gamma$, for some Δ_{21}, Δ_{22} ([Tmix^{-1}] and (3))
(8) $\Delta_{21} = \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}_2} A$, for some Δ'_{21} ((5), (6) and $x \in \mathsf{fn}(Q)$)
(9) $Q \vdash_\eta \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}_2} A$ ((6) and (8))
(10) **share** x $\{P \ || \ Q\} \vdash_\eta \Delta_1, \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}} A;\Gamma$ ([Tsh], (2), (9) and (4))
(11) **par** $\{(\textbf{share}\ x\ \{P \ || \ Q\}) \ || \ R\} \vdash_\eta \Delta_1, \Delta'_{21}, \Delta_{22}, x : \mathbf{U}_{\mathcal{X}} A;\Gamma$ ([Tmix], (10) and (7))
(12) $\Delta_1, \Delta'_{21}, \Delta_{22}, x : \mathbf{U}_{\mathcal{X}} A = \Delta$ ((1), (5) and (8))
(13) **par** $\{(\textbf{share}\ x\ \{P \ || \ Q\}) \ || \ R\} \vdash_\eta \Delta;\Gamma$ ((11) and (12))

Case: [ShM] right-to-left, **par** $\{\textbf{share}\ x\ \{P \ || \ Q\} \ || \ R\} \equiv \textbf{share}\ x\ \{P \ || \ (\textbf{par}\ \{Q \ || \ R\})\}$, $x \in \mathrm{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) **share** x $\{P \ || \ Q\} \vdash_\eta \Delta_1;\Gamma$ (3) $R \vdash_\eta \Delta_2;\Gamma$, for some Δ_1, Δ_2 ([Tmix^{-1}] and **par** $\{(\textbf{share}\ x\ \{P \ || \ Q\}) \ || \ R\} \vdash_\eta \Delta;\Gamma$)
(4) $\Delta_1 = \Delta_{11}, \Delta_{12}, x : \mathbf{U}_{\mathcal{X}} A$ (5) $P \vdash_\eta \Delta_{11}, x : \mathbf{U}_{\mathcal{X}_1} A;\Gamma$
(6) $Q \vdash_\eta \Delta_{12}, x : \mathbf{U}_{\mathcal{X}_2} A;\Gamma$ (7) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_{11}, \Delta_{12}, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ ([Tsh^{-1}] and (2))
(8) **par** $\{Q \ || \ R\} \vdash_\eta \Delta_{12}, x : \mathbf{U}_{\mathcal{X}_2} A, \Delta_2;\Gamma$ ([Tmix], (6) and (3))
(9) **share** x $\{P \ || \ (\textbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_{\mathcal{X}} A;\Gamma$ ([Tsh], (5) and (8))
(10) $\Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_{\mathcal{X}} A = \Delta$ ((4) and (1))
(11) **share** x $\{P \ || \ (\textbf{par}\ \{Q \ || \ R\})\} \vdash_\eta \Delta;\Gamma$ ((9) and (10))

Case: [ShC!] left-to-right, **share** x $\{P \ || \ (\textbf{cut!}\ \{y.Q \ |z : B|\ R\})\} \equiv \textbf{cut!}\ \{y.Q \ |z : B|\ (\textbf{share}\ x\ \{P \ || \ R\})\}$, $z \notin \mathrm{fn}(P)$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}} A$ (2) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1} A;\Gamma$
(3) **cut!** $\{y.Q \ |z : B|\ R\} \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2} A;\Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$

([Tsh^{-1}] and **share** $x\ \{P \mid\mid (\textbf{cut!}\ \{y.Q \mid z : B \mid R\})\} \vdash_\eta \Delta; \Gamma$)
(5) $Q \vdash_\eta y : B; \Gamma$ (6) $R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma, z : \overline{B}$ ([Tcut!$^{-1}$] and (3))
(7) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma, z : \overline{B}$ (Lemma 29([Tweaken]), (2) and $z \notin \mathsf{fn}(P)$)
(8) **share** $x\ \{P \mid\mid R\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma, z : \overline{B}$ ([Tsh], (7), (6) and (4))
(9) **cut!** $\{y.Q \mid z : B \mid (\textbf{share}\ x\ \{P \mid\mid R\})\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ([Tcut!], (5) and (8))
(10) **cut!** $\{y.Q \mid z : B \mid (\textbf{share}\ x\ \{P \mid\mid R\})\} \vdash_\eta \Delta; \Gamma$ ((1) and (9))

Case: [ShC!] right-to-left,

cut! $\{y.Q \mid z : B \mid (\textbf{share}\ x\ \{P \mid\mid R\})\} \equiv \textbf{share}\ x\ \{P \mid\mid (\textbf{cut!}\ \{y.Q \mid z : B \mid R\})\}$, $z \notin \mathsf{fn}(P)$.

(1) $Q \vdash_\eta y : B; \Gamma$ (2) **share** $x\ \{P \mid\mid R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$
([Tcut!$^{-1}$] and **cut!** $\{y.Q \mid z : B \mid (\textbf{share}\ x\ \{P \mid\mid R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ (4) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma, z : \overline{B}$
(5) $R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma, z : \overline{B}$ (6) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$
([Tsh^{-1}] and (2))
(7) **cut!** $\{y.Q \mid z : B \mid R\} \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ ([Tcut!], (1) and (5))
(8) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$ (Lemma 29([Tstrength]), (4) and $z \notin \mathsf{fn}(P)$)
(9) **share** $x\ \{P \mid\mid (\textbf{cut!}\ \{y.Q \mid z : B \mid R\})\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ([Tsh], (8), (7) and (6))
(10) **share** $x\ \{P \mid\mid (\textbf{cut!}\ \{y.Q \mid z : B \mid R\})\} \vdash_\eta \Delta; \Gamma$ ((3) and (9))

Case: [ShSh] left-to-right, **share** $x\ \{P \mid\mid (\textbf{share}\ y\ \{Q \mid\mid R\})\} \equiv \textbf{share}\ y\ \{(\textbf{share}\ x\ \{P \mid\mid Q\}) \mid\mid R\}$, $x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ (2) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$
(3) **share** $y\ \{Q \mid\mid R\} \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$
([Tsh^{-1}] and **share** $x\ \{P \mid\mid (\textbf{share}\ y\ \{Q \mid\mid R\})\} \vdash_\eta \Delta; \Gamma$)
(5) $\Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A = \Delta_{21}, \Delta_{22}, y : \mathbf{U}_{\mathcal{Y}}\ B$ (6) $Q \vdash_\eta \Delta_{21}, y : \mathbf{U}_{\mathcal{Y}_1}\ B; \Gamma$
(7) $R \vdash_\eta \Delta_{22}, y : \mathbf{U}_{\mathcal{Y}_2}\ B; \Gamma$ (8) $\mathcal{Y}_1 \oplus \mathcal{Y}_2 = \mathcal{Y}$, for some $\Delta_{21}, \Delta_{22}, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}$ ([Tsh^{-1}] and (3))
(9) $\Delta_{21} = \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}_2}\ A$, for some Δ'_{21} ((5), (6) and $x \in \mathsf{fn}(Q)$)
(10) $Q \vdash_\eta \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}_2}\ A, y : \mathbf{U}_{\mathcal{Y}_1}\ B; \Gamma$ ((6) and (9))
(11) **share** $x\ \{P \mid\mid Q\} \vdash_\eta \Delta_1, \Delta'_{21}, x : \mathbf{U}_{\mathcal{X}}\ A, y : \mathbf{U}_{\mathcal{Y}_1}\ B; \Gamma$ ([Tsh], (2), (10) an (4))
(12) **share** $y\ \{(\textbf{share}\ x\ \{P \mid\mid Q\}) \mid\mid R\} \vdash_\eta \Delta_1, \Delta'_{21}, \Delta_{22}, x : \mathbf{U}_{\mathcal{X}}\ A, y : \mathbf{U}_{\mathcal{Y}}\ B; \Gamma$
([Tsh], (11), (7) and (8))
(13) $\Delta_1, \Delta'_{21}, \Delta_{22}, x : \mathbf{U}_{\mathcal{X}}\ A, y : \mathbf{U}_{\mathcal{Y}}\ B = \Delta$ ((1), (5) and (9))
(14) **share** $y\ \{(\textbf{share}\ x\ \{P \mid\mid Q\}) \mid\mid R\} \vdash_\eta \Delta; \Gamma$ ((11) and (12))

Case: [ShSh] right-to-left, **share** $y\ \{(\textbf{share}\ x\ \{P \mid\mid Q\}) \mid\mid R\} \equiv \textbf{share}\ x\ \{P \mid\mid (\textbf{share}\ y\ \{Q \mid\mid R\})\}$, $x, y \in \mathsf{fn}(Q)$.

(1) $\Delta = \Delta_1, \Delta_2, y : \mathbf{U}_{\mathcal{Y}}\ B$ (2) **share** $x\ \{P \mid\mid Q\} \vdash_\eta \Delta_1, y : \mathbf{U}_{\mathcal{Y}_1}\ B; \Gamma$
(3) $R \vdash_\eta \Delta_2, y : \mathbf{U}_{\mathcal{Y}_2}\ B; \Gamma$ (4) $\mathcal{Y}_1 \oplus \mathcal{Y}_2 = \mathcal{Y}$, for some $\Delta_1, \Delta_2, B, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}$

([Tsh^{-1}] and **share** y $\{(\textbf{share}\ x\ \{P \parallel Q\}) \parallel R\} \vdash_\eta \Delta;\Gamma$)
(5) $\Delta_1, y : \mathbf{U}_{\mathcal{Y}_1}\ B = \Delta_{11}, \Delta_{12}, x : \mathbf{U}_{\mathcal{X}}\ A$ (6) $P \vdash_\eta \Delta_{11}, x : \mathbf{U}_{\mathcal{X}_1}\ A\Gamma$
(7) $Q \vdash_\eta \Delta_{12}, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ (8) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_{11}, \Delta_{12}, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ ([Tsh^{-1}] and (2))
(9) $\Delta_{12} = \Delta'_{12}, y : \mathbf{U}_{\mathcal{Y}_1}\ B$, for some Δ'_{12} ((5), (7) and $y \in \mathsf{fn}(Q)$)
(10) $Q \vdash_\eta \Delta'_{12}, y : \mathbf{U}_{\mathcal{Y}_1}\ B, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ ((7) and (9))
(11) **share** y $\{Q \parallel R\} \vdash_\eta \Delta'_{12}, x : \mathbf{U}_{\mathcal{X}_2}\ A, y : \mathbf{U}_{\mathcal{Y}}\ B, \Delta_2;\Gamma$ ([Tsh], (10), (3) and (4))
(12) **share** x $\{P \parallel (\textbf{share}\ y\ \{Q \parallel R\})\} \vdash_\eta \Delta_{11}, \Delta'_{12}, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A, y : \mathbf{U}_{\mathcal{Y}}\ B;\Gamma$ ([Tsh], (6) and (11))
(13) $\Delta_{11}, \Delta'_{12}, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A, y : \mathbf{U}_{\mathcal{Y}}\ B = \Delta$ ((1), (5) and (9))
(14) **share** x $\{P \parallel (\textbf{share}\ y\ \{Q \parallel R\})\} \vdash_\eta \Delta;\Gamma$ ((12) and (13))

Case: [D-C!M] left-to-right,

cut! $\{y.P\ |x : A|\ (\textbf{par}\ \{Q \parallel R\})\} \equiv \textbf{par}\ \{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\}) \parallel (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\}$.

(1) $P \vdash_\eta y : A;\Gamma$ (2) **par** $\{Q \parallel R\} \vdash_\eta \Delta;\Gamma, x : \overline{A}$
([Tcut!$^{-1}$] and **cut!** $\{y.P\ |x : A|\ (\textbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $Q \vdash_\eta \Delta_1;\Gamma, x : \overline{A}$ (5) $R \vdash_\eta \Delta_2;\Gamma, x : \overline{A}$, for some Δ_1, Δ_2 ([Tmix^{-1}] and (2))
(6) **cut!** $\{y.P\ |x : A|\ Q\} \vdash_\eta \Delta_1;\Gamma$ ([Tcut!], (1) and (4))
(7) **cut!** $\{y.P\ |x : A|\ R\} \vdash_\eta \Delta_2;\Gamma$ ([Tcut!], (1) and (5))
(8) **par** $\{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\}) \parallel (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tmix], (6) and (7))
(9) **par** $\{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\}) \parallel (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta;\Gamma$ ((3) and (8))

Case: [D-C!M] right-to-left,

par $\{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\}) \parallel (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\} \equiv \textbf{cut!}\ \{y.P\ |x : A|\ (\textbf{par}\ \{Q \parallel R\})\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) **cut!** $\{y.P\ |x : A|\ Q\} \vdash_\eta \Delta_1;\Gamma$ (3) **cut!** $\{y.P\ |x : A|\ R\} \vdash_\eta \Delta_2;\Gamma$, for some Δ_1, Δ_2 ([Tmix^{-1}] and **par** $\{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\}) \parallel (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta;\Gamma$)
(4) $P \vdash_\eta y : A;\Gamma$ (5) $Q \vdash_\eta \Delta_1;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(6) $R \vdash_\eta \Delta_2;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (3))
(7) **par** $\{Q \parallel R\} \vdash_\eta \Delta_1, \Delta_2;\Gamma, x : \overline{A}$ ([Tmix], (5) and (6))
(8) **cut!** $\{y.P\ |x : A|\ (\textbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tcut!], (4) and (7))
(9) **cut!** $\{y.P\ |x : A|\ (\textbf{par}\ \{Q \parallel R\})\} \vdash_\eta \Delta;\Gamma$ ((1) and (8))

Case: [D-C!C] left-to-right,

cut! $\{y.P\ |x : A|\ (\textbf{cut}\ \{Q\ |z : B|\ R\})\} \equiv \textbf{cut}\ \{(\textbf{cut!}\ \{y.P\ |x : A|\ Q\})\ |z : B|\ (\textbf{cut!}\ \{y.P\ |x : A|\ R\})\}$.

(1) $P \vdash_\eta y : A;\Gamma$ (2) **cut** $\{Q\ |z : B|\ R\} \vdash_\eta \Delta;\Gamma, x : \overline{A}$
([Tcut!$^{-1}$ and **cut!** $\{y.P\ |x : A|\ (\textbf{cut}\ \{Q\ |z : B|\ R\})\} \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $Q \vdash_\eta \Delta_1, z : B;\Gamma, x : \overline{A}$ (5) $R \vdash_\eta \Delta_2, z : \overline{B};\Gamma, x : \overline{A}$, for some Δ_1, Δ_2

([Tcut!$^{-1}$] and (2))
(6) **cut!** $\{y.P \ |x : A|\ Q\} \vdash_\eta \Delta_1, z : B; \Gamma$ ([Tcut!], (1) and (4))
(7) **cut!** $\{y.P \ |x : A|\ R\} \vdash_\eta \Delta_2, z : \overline{B}; \Gamma$ ([Tcut!], (1) and (5))
(8) **cut** $\{(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ \ |z : B|\ \ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (6) and (7))
(9) **cut** $\{(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ \ |z : B|\ \ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$ ((3) and (8))

Case: [D-C!C] right-to-left,

cut $\{(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ \ |z : B|\ \ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \equiv \textbf{cut!}\ \{y.P \ |x : A|\ (\textbf{cut}\ \{Q \ \ |z : B|\ \ R\})\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) **cut!** $\{y.P \ |x : A|\ Q\} \vdash_\eta \Delta_1, z : B; \Gamma$ (3) **cut!** $\{y.P \ |x : A|\ R\} \vdash_\eta \Delta_2, z : \overline{B}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and **cut** $\{(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ \ |z : B|\ \ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(4) $P \vdash_\eta y : A; \Gamma$ (5) $Q \vdash_\eta \Delta_1, z : B; \Gamma, x : \overline{A}$ ([Tcut^{-1}] and (2))
(6) $R \vdash_\eta \Delta_2, z : \overline{B}; \Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (3))
(7) **cut** $\{Q \ \ |z : B|\ \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma, x : \overline{A}$ ([Tcut], (5) and (6))
(8) **cut!** $\{y.P \ |x : A|\ (\textbf{cut}\ \{Q \ \ |z|\ \ R\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut!], (4) and (7))
(9) **cut!** $\{y.P \ |x : A|\ (\textbf{cut}\ \{Q \ \ |z|\ \ R\})\} \vdash_\eta \Delta; \Gamma$ ((1) and (8))

Case: [D-C!C!] left-to-right,

cut! $\{y.P \ |x : A|\ (\textbf{cut!}\ \{w.Q \ |z : B|\ R\})\} \equiv \textbf{cut!}\ \{w.(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ |z : B|\ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\}$.

(1) $P \vdash_\eta y : A; \Gamma$ (2) **cut!** $\{w.Q \ |z : B|\ R\} \vdash_\eta \Delta; \Gamma, x : \overline{A}$
([Tcut!$^{-1}$] and **cut!** $\{y.P \ |x : A|\ (\textbf{cut!}\ \{w.Q \ |z : B|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $Q \vdash_\eta w : B; \Gamma, x : \overline{A}$ (4) $R \vdash_\eta \Delta; \Gamma, x : \overline{A}, z : \overline{B}$ ([Tcut!$^{-1}$] and (2))
(5) **cut!** $\{y.P \ |x : A|\ Q\} \vdash_\eta w : B; \Gamma$ ([Tcut!] (1) and (3))
(6) **cut!** $\{y.P \ |x : A|\ R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$ ([Tcut!], (1) and (4))
(7) **cut!** $\{w.(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ |z : B|\ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$ ([Tcut!], (5) and (6))

Case: [D-C!C!] right-to-left,

cut! $\{w.(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ |z : B|\ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \equiv \textbf{cut!}\ \{y.P \ |x : A|\ (\textbf{cut!}\ \{w.Q \ |z : B|\ R\})\}$.

(1) **cut!** $\{y.P \ |x : A|\ Q\} \vdash_\eta w : B; \Gamma$ (2) **cut!** $\{y.P \ |x : A|\ R\} \vdash_\eta \Delta; \Gamma, z : \overline{B}$
([Tcut!$^{-1}$] and **cut!** $\{w.(\textbf{cut!}\ \{y.P \ |x : A|\ Q\})\ |z : B|\ (\textbf{cut!}\ \{y.P \ |x : A|\ R\})\} \vdash_\eta \Delta; \Gamma$)
(3) $P \vdash_\eta y : A; \Gamma$ (4) $Q \vdash_\eta w : B; \Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (1))
(5) $R \vdash_\eta \Delta; \Gamma, z : \overline{B}, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(6) **cut!** $\{w.Q \ |z : B|\ R\} \vdash_\eta \Delta; \Gamma, x : \overline{A}$ ([Tcut!], (4) and (5))

(7) $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{cut!}\ \{w.Q\ |z|\ R\})\} \vdash_\eta \Delta;\Gamma$ ([Tcut!], (3) and (6))

Case: [D-C!Sh] left-to-right,

$\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{share}\ z\ \{Q\ ||\ R\}\} \equiv \mathbf{share}\ z\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\}$.

(1) $P \vdash_\eta y : A;\Gamma$ (2) $\mathbf{share}\ z\ \{Q\ ||\ R\} \vdash_\eta \Delta;\Gamma, x : \overline{A}$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{share}\ z\ \{Q\ ||\ R\})\} \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{Y}}\ B$ (4) $Q \vdash_\eta \Delta_1, z : \mathbf{U}_{\mathcal{Y}_1}\ B;\Gamma, x : \overline{A}$
(5) $R \vdash_\eta \Delta_2, z : \mathbf{U}_{\mathcal{Y}_2}\ B;\Gamma, x : \overline{A}$ (6) $\mathcal{Y}_1 \oplus \mathcal{Y}_2 = \mathcal{Y}$, for some $\Delta_1, \Delta_2, B, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}$
([Tsh^{-1}] and (2))
(7) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \vdash_\eta \Delta_1, z : \mathbf{U}_{\mathcal{Y}_1}\ B;\Gamma$ ([Tcut!], (1) and (4))
(8) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta_2, z : \mathbf{U}_{\mathcal{Y}_2}\ B;\Gamma$ ([Tcut!], (1) and (5))
(9) $\mathbf{share}\ z\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{Y}}\ B;\Gamma$
([Tsh], (7), (8) and (6))
(10) $\mathbf{share}\ z\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta;\Gamma$ ((3) and (9))

Case: [D-C!Sh] right-to-left,

$\mathbf{share}\ z\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \equiv \mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{share}\ z\ \{Q\ ||\ R\}\}$.

(1) $\Delta = \Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{Y}}\ B$ (2) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \vdash_\eta \Delta_1, z : \mathbf{U}_{\mathcal{Y}_1}\ B;\Gamma$
(3) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta_2, z : \mathbf{U}_{\mathcal{Y}_2}\ B;\Gamma$ (4) $\mathcal{Y}_1 \oplus \mathcal{Y}_2 = \mathcal{Y}$, for some $\Delta_1, \Delta_2, B, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}$
([Tsh^{-1}] and $\mathbf{share}\ z\ \{(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})\} \vdash_\eta \Delta;\Gamma$)
(5) $P \vdash_\eta y : A;\Gamma$ (6) $Q \vdash_\eta \Delta_1, z : \mathbf{U}_{\mathcal{Y}_1}\ B;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(7) $R \vdash_\eta \Delta_2, z : \mathbf{U}_{\mathcal{Y}_2}\ B;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (3))
(8) $\mathbf{share}\ z\ \{Q\ ||\ R\} \vdash_\eta \Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{Y}}\ B;\Gamma, x : \overline{A}$ ([Tsh], (6), (7) and (4))
(9) $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{share}\ z\ \{Q\ ||\ R\})\} \vdash_\eta \Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{Y}}\ B;\Gamma$ ([Tcut!], (5) and (8))
(10) $\mathbf{cut!}\ \{y.P\ |x : A|\ (\mathbf{share}\ z\ \{Q\ ||\ R\})\} \vdash_\eta \Delta;\Gamma$ ((1) and (0)

Case: [C!Sm] left-to-right,

$\mathbf{cut!}\ \{y.P\ |x : A|\ (Q + R)\} \equiv (\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}) + (\mathbf{cut!}\ \{y.P\ |x : A|\ R\})$.

(1) $P \vdash_\eta y : A;\Gamma$ (2) $Q + R \vdash_\eta \Delta;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.P\ |x : A|\ (Q + R)\} \vdash_\eta \Delta;\Gamma$)
(3) $Q \vdash_\eta \Delta;\Gamma, x : \overline{A}$ (4) $R \vdash_\eta \Delta;\Gamma, x : \overline{A}$ ([Tsum^{-1}] and (2))
(5) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \vdash_\eta \Delta;\Gamma$ ([Tcut!], (1) and (3))
(6) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta;\Gamma$ ([Tcut!], (1) and (4))
(7) $(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}) + (\mathbf{cut!}\ \{y.P\ |x : A|\ R\}) \vdash_\eta \Delta;\Gamma$ ([Tsum], (5) and (6))

Case: [C!Sm] right-to-left,

$(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}) + (\mathbf{cut!}\ \{y.P\ |x : A|\ R\}) \equiv \mathbf{cut!}\ \{y.P\ |x : A|\ (Q + R)\}$.

(1) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \vdash_\eta \Delta;\Gamma$ (2) $\mathbf{cut!}\ \{y.P\ |x : A|\ R\} \vdash_\eta \Delta;\Gamma$
([Tsum^{-1}] and $(\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}) + (\mathbf{cut!}\ \{y.P\ |x : A|\ R\}) \vdash_\eta \Delta;\Gamma$)
(3) $P \vdash_\eta y : A;\Gamma$ (4) $Q \vdash_\eta \Delta;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (1))
(5) $R \vdash_\eta \Delta;\Gamma, x : \overline{A}$ ([Tcut!$^{-1}$] and (2))
(6) $Q + R \vdash_\eta \Delta;\Gamma, x : \overline{A}$ ([Tsum], (4) and (5))
(8) $\mathbf{cut!}\ \{y.P\ |x : A|\ (Q + R)\} \vdash_\eta \Delta;\Gamma$ ([Tcut!], (3) and (6))

Case: [MSm] left-to-right, $\mathbf{par}\ \{P \parallel (Q + R)\} \equiv (\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\})$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1;\Gamma$ (3) $Q + R \vdash_\eta \Delta_2;\Gamma$ for some Δ_1, Δ_2
([Tmix^{-1}] and $\mathbf{par}\ \{P \parallel (Q + R)\} \vdash_\eta \Delta;\Gamma$)
(4) $Q \vdash_\eta \Delta_2;\Gamma$ (5) $R \vdash_\eta \Delta_2;\Gamma$ ([Tsum^{-1}] and (3))
(5) $\mathbf{par}\ \{P \parallel Q\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tmix], (2) and (4))
(6) $\mathbf{par}\ \{P \parallel R\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tmix], (2) and (5))
(7) $(\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\}) \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tsum], (5) and (6))
(8) $(\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\}) \vdash_\eta \Delta;\Gamma$ ((1) and (7))

Case: [MSm] right-to-left, $(\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\}) \equiv \mathbf{par}\ \{P \parallel (Q + R)\}$.

(1) $\mathbf{par}\ \{P \parallel Q\} \vdash_\eta \Delta;\Gamma$ (2) $\mathbf{par}\ \{P \parallel R\} \vdash_\eta \Delta;\Gamma$
([Tsum^{-1}] and $(\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\}) \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $P \vdash_\eta \Delta_1;\Gamma$ (5) $Q \vdash_\eta \Delta_2;\Gamma$, for some Δ_1, Δ_2 ([Tmix^{-1}] and (1))
(6) $\Delta = \Delta'_1, \Delta'_2$ (7) $P \vdash_\eta \Delta'_1;\Gamma$ (8) $R \vdash_\eta \Delta'_2;\Gamma$, for some Δ'_1, Δ'_2 ([Tmix^{-1}] and (2))
(9) $\Delta_1 = \Delta \upharpoonright \mathrm{dom}(\Delta_1)$ (10) $\Delta'_1 = \Delta \upharpoonright \mathrm{dom}(\Delta'_1)$ ((3) and (6))
(11) $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta'_1)$ (Lemma 30(1), (4) and (7))
(12) $\Delta_1 = \Delta'_1$ ((9), (10) and (11))
(13) $\Delta_2 = \Delta'_2$ ((3), (6) and (12))
(14) $R \vdash_\eta \Delta_2;\Gamma$ ((8) and (13))
(15) $Q + R \vdash_\eta \Delta_2;\Gamma$ ([Tsum], (5) and (14))
(16) $\mathbf{par}\ \{P \parallel (Q + R)\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tmix], (4) and (15))
(17) $\mathbf{par}\ \{P \parallel (Q + R)\} \vdash_\eta \Delta;\Gamma$ ((3) and (16))

Case: [CSm] left-to-right, $\mathbf{cut}\ \{P\ |x : A|\ (Q + R)\} \equiv (\mathbf{cut}\ \{P\ |x : A|\ Q\}) + (\mathbf{cut}\ \{P\ |x : A|\ R\})$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1, x : A;\Gamma$ (3) $Q + R \vdash_\eta \Delta_2, x : \overline{A};\Gamma$ for some Δ_1, Δ_2
([Tcut!$^{-1}$] and $\mathbf{cut}\ \{P\ |x : A|\ (Q + R)\} \vdash_\eta \Delta;\Gamma$)
(4) $Q \vdash_\eta \Delta_2, x : \overline{A};\Gamma$ (5) $R \vdash_\eta \Delta_2, x : \overline{A};\Gamma$ ([Tsum^{-1}] and (3))
(5) $\mathbf{cut}\ \{P\ |x : A|\ Q\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tcut], (2) and (4))
(6) $\mathbf{cut}\ \{P\ |x : A|\ R\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tcut], (2) and (5))
(7) $(\mathbf{cut}\ \{P\ |x : A|\ Q\}) + (\mathbf{cut}\ \{P\ |x : A|\ R\}) \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tsum], (5) and (6))

(8) $(\mathbf{cut}\ \{P \ \ |x : A|\ \ Q\}) + (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\}) \vdash_\eta \Delta;\Gamma$ ((1) and (7))

Case: [CSm] right-to-left, $(\mathbf{cut}\ \{P \ \ |x : A|\ \ Q\}) + (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\}) \equiv \mathbf{cut}\ \{P \ \ |x : A|\ \ (Q + R)\}$.

(1) $\mathbf{cut}\ \{P \ \ |x : A|\ \ Q\} \vdash_\eta \Delta;\Gamma$ (2) $\mathbf{cut}\ \{P \ \ |x : A|\ \ R\} \vdash_\eta \Delta;\Gamma$
([Tsum^{-1}] and $(\mathbf{cut}\ \{P \ \ |x : A|\ \ Q\}) + (\mathbf{cut}\ \{P \ \ |x : A|\ \ R\}) \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2$ (4) $P \vdash_\eta \Delta_1, x : A;\Gamma$ (5) $Q \vdash_\eta \Delta_2, x : \overline{A};\Gamma$, for some Δ_1, Δ_2 ([Tcut!$^{-1}$] and (1))
(6) $\Delta = \Delta'_1, \Delta'_2$ (7) $P \vdash_\eta \Delta'_1, x : A;\Gamma$ (8) $R \vdash_\eta \Delta'_2, x : \overline{A};\Gamma$, for some Δ'_1, Δ'_2 ([Tcut!$^{-1}$] and (2))
(9) $\Delta_1 = \Delta \upharpoonright \mathrm{dom}(\Delta_1)$ (10) $\Delta'_1 = \Delta \upharpoonright \mathrm{dom}(\Delta'_1)$ ((3) and (6))
(11) $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta'_1)$ (Lemma 30(1), (4) and (7))
(12) $\Delta_1 = \Delta'_1$ ((9), (10) and (11))
(13) $\Delta_2 = \Delta'_2$ ((3), (6) and (12))
(14) $R \vdash_\eta \Delta_2, x : \overline{A};\Gamma$ ((8) and (13))
(15) $Q + R \vdash_\eta \Delta_2, x : \overline{A};\Gamma$ ([Tsum], (5) and (14))
(16) $\mathbf{cut}\ \{P \ \ |x : A|\ \ (Q + R)\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tcut], (4) and (15))
(17) $\mathbf{cut}\ \{P \ \ |x : A|\ \ (Q + R)\} \vdash_\eta \Delta;\Gamma$ ((3) and (16))

Case: [ShSm] left-to-right, $\mathbf{share}\ x\ \{P \,||\, (Q + R)\} \equiv (\mathbf{share}\ x\ \{P \,||\, Q\}) + (\mathbf{share}\ x\ \{P \,||\, R\})$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A$ (2) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A;\Gamma$
(3) $Q + R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some Δ_1, Δ_2, A
([Tsh^{-1}] and $\mathbf{share}\ x\ \{P \,||\, (Q + R)\} \vdash_\eta \Delta;\Gamma$)
(5) $Q \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ (6) $R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ ([Tsum^{-1}], (3))
(7) $\mathbf{share}\ x\ \{P \,||\, Q\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A;\Gamma$ ([Tsh], (2), (5) and (4))
(8) $\mathbf{share}\ x\ \{P \,||\, R\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A;\Gamma$ ([Tsh], (2), (6) and (4))
(9) $(\mathbf{share}\ x\ \{P \,||\, Q\}) + (\mathbf{share}\ x\ \{P \,||\, R\}) \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A;\Gamma$ ([Tsum], (7) and (8))
(10) $(\mathbf{share}\ x\ \{P \,||\, Q\}) + (\mathbf{share}\ x\ \{P \,||\, R\}) \vdash_\eta \Delta;\Gamma$ ((1) and (9))

Case: [ShSm] right-to-left, $(\mathbf{share}\ x\ \{P \,||\, Q\}) + (\mathbf{share}\ x\ \{P \,||\, R\}) \equiv \mathbf{share}\ x\ \{P \,||\, (Q + R)\}$.

(1) $\mathbf{share}\ x\ \{P \,||\, Q\} \vdash_\eta \Delta;\Gamma$ (2) $\mathbf{share}\ x\ \{P \,||\, R\} \vdash_\eta \Delta;\Gamma$
([Tsum^{-1}] and $(\mathbf{share}\ x\ \{P \,||\, Q\}) + (\mathbf{share}\ x\ \{P \,||\, R\}) \vdash_\eta \Delta;\Gamma$)
(3) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A$ (4) $P \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A;\Gamma$
(5) $Q \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A;\Gamma$ (6) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ ([Tsh^{-1}] and (1))
(7) $\Delta = \Delta'_1, \Delta'_2, x : \mathbf{U}_\mathcal{Y}\ B$ (8) $P \vdash_\eta \Delta'_1, x : \mathbf{U}_{\mathcal{Y}_1}\ B;\Gamma$
(9) $R \vdash_\eta \Delta'_2, x : \mathbf{U}_{\mathcal{Y}_2}\ B;\Gamma$ (10) $\mathcal{Y}_1 \oplus \mathcal{Y}_2 = \mathcal{Y}$, for some $\Delta'_1, \Delta'_2, B, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Y}$ ([Tsh^{-1}] and (2))
(11) $\Delta_1 = \Delta \upharpoonright \mathrm{dom}(\Delta_1)$ (12) $\Delta'_1 = \Delta \upharpoonright \mathrm{dom}(\Delta'_1)$ (13) $A = B$ (14) $\mathcal{X} = \mathcal{Y}$
((3) and (7))
(15) $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta'_1)$ (Lemma 30(1), (4) and (8))
(16) $\Delta_1 = \Delta'_1$ ((11), (12) and (15))

(17)$\mathcal{X}_1 = \mathcal{Y}_1$ (Lemma 30(2), (4), (8), (16) and (13))
(18) $\mathcal{X}_2 = \mathcal{Y}_2$ ((6), (10), (14) and (17))
(19) $\Delta_2 = \Delta_2'$ ((3), (7) and (16))
(20) $R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ ((9),(13), (18) and (19))
(21) $Q + R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ ([Tsum], (5) and (20))
(22) **share** x $\{P \,||\, (Q + R)\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ([Tsh], (4), (21) and (6))
(23) **share** x $\{P \,||\, (Q + R)\} \vdash_\eta \Delta; \Gamma$ ((3) and (22))

Case: [RSh] left-to-right, **share** x $\{$**release** x $||$ $P\} \equiv P$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ (2) **release** $x \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$
(3) $P \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$
([Tsh^{-1}] and **share** x $\{$**release** x $||$ $P\} \vdash_\eta \Delta; \Gamma$)
(5) $\Delta_1 = \emptyset$ (6) $\mathcal{X}_1 = f$ ([Tfree^{-1}] and (2))
(7) $P \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ ((3) and (5))
(8) $\mathcal{X} = \mathcal{X}_2$ ((4) and (6))
(9) $P \vdash_\eta \Delta; \Gamma$ ((1), (7) and (8))

Case: [RSh] right-to-left, $P \equiv$ **share** x $\{$**release** x $||$ $P\}$.

(1) $\Delta = \Delta', x : \mathbf{U}_{\mathcal{X}}\ A$, for some $\Delta', A, \mathcal{X}$ (Def. 13 (proviso of [RSh]) and $P \vdash_\eta \Delta; \Gamma$)
(2) $P \vdash_\eta \Delta', x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ((1) and $P \vdash_\eta \Delta; \Gamma$
(3) **release** $x \vdash_\eta x : \mathbf{U}_f\ A; \Gamma$ ([Trelease])
(4) **share** x $\{$**release** x $||$ $P\} \vdash_\eta \Delta', x : \mathbf{U}_{\mathcal{X}}\ A; \Gamma$ ([Tsh], (3), (2) and $f \oplus \mathcal{X} = \mathcal{X}$
(5) **share** x $\{$**release** x $||$ $P\} \vdash_\eta \Delta; \Gamma$ ((1) and (4))

Case: [TSh] left-to-right,

share x $\{$**take** $x(y_1); P_1$ $||$ **take** $x(y_2); P_2)\}$
$\equiv$ **take** $x(y_1)$; **share** x $\{P_1$ $||$ **take** $x(y_2); P_2\}$ + **take** $x(y_2)$; **share** x $\{$**take** $x(y_1); P_1$ $||$ $P_2\}$

.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_{\mathcal{X}}\ A$ (2) **take** $x(y_1); P_1 \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$
(3) **take** $x(y_2); P_2 \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $\Delta_1, \Delta_2, A, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$
([Tsh^{-1}] and **share** x $\{$**take** $x(y_1); P_1$ $||$ **take** $x(y_2); P_2\} \vdash_\eta \Delta; \Gamma$)
(5) $P_1 \vdash_\eta \Delta_1, x : \mathbf{U}_e\ A, y_1 : \vee A; \Gamma$ (6) $\mathcal{X}_1 = f$ ([Ttake^{-1}] and (2))
(7) $P_2 \vdash_\eta \Delta_2, x : \mathbf{U}_e\ A, y_2 : \vee A; \Gamma$ (8) $\mathcal{X}_2 = f$ ([Ttake^{-1}] and (3))
(9) **share** x $\{P_1$ $||$ **take** $x(y_2); P_2\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_e\ A, y_1 : \vee A; \Gamma$ ([Tsh],(5), (3), (8) and $e \oplus f = e$)
(10) **take** $x(y_1)$; **share** x $\{P_1$ $||$ **take** $x(y_2); P_2\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ ([Ttake] and (9))
(11) **share** x $\{$**take** $x(y_1); P_1$ $||$ $P_2\} \vdash_\eta \Delta_1, \Delta_2, y_2 : \vee A, x : \mathbf{U}_e\ A; \Gamma$ ([Tsh], (2), (6), (7) and $f \oplus e = e$)
(12) **take** $x(y_2)$; **share** x $\{$**take** $x(y_1); P_1$ $||$ $P_2\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ ([Ttake] and (11))
(13) **take** $x(y_1)$; **share** x $\{P_1$ $||$ **take** $x(y_2); P_2\}$ +

$\mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, P_2\} \vdash_\eta \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ ([Tsum], (10) and (12))
(14) $\mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} +$
$\mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, P_2\} \vdash_\eta \Delta; \Gamma$ ((1), (4), (6), (8), $f \oplus f = f$ and (13))

Case: [TSh] right-to-left, $P \equiv \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, \mathbf{take}\ x(y_2); P_2\}$,
where $P = \mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} + \mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, P_2\}$.
Provisos: $y_1 \in \mathsf{fn}(P_1)$ and $y_2 \in \mathsf{fn}(P_2)$.

(1) $\mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} \vdash_\eta \Delta; \Gamma$
(2) $\mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, P_2\} \vdash_\eta \Delta; \Gamma$ ([Tsum^{-1}] and $P \vdash_\eta \Delta; \Gamma$)
(3) $\Delta = \Delta_1, x : \mathbf{U}_f\ A$ (4) $\mathbf{share}\ x\ \{P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} \vdash_\eta \Delta_1, x : \mathbf{U}_e\ A, y_1 : \vee A; \Gamma$
([Ttake^{-1}] and (1))
(5) $\Delta_1, y_1 : \vee A = \Delta_{11}, \Delta_{12}$ (6) $P_1 \vdash_\eta \Delta_{11}, \mathbf{U}_e\ A; \Gamma$ (7) $\mathbf{take}\ x(y_2); P_2 \vdash_\eta \Delta_{12}, x :$
$\mathbf{U}_f\ A; \Gamma$ ([Tsh^{-1}] and (4))
(8) $\Delta_{11} = \Delta'_{11}, y_1 : \vee A$ ((5), (6) and $y_1 \in \mathsf{fn}(P_1)$)
(9) $P_1 \vdash_\eta \Delta'_{11}, y_1 : \vee A, x : \mathbf{U}_e\ A; \Gamma$ ((6) and (8))
(10) $\mathbf{take}\ x(y_1); P_1 \vdash_\eta \Delta'_{11}, x : \mathbf{U}_f\ A; \Gamma$ ([Ttake] and (9))
(11) $\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} \vdash_\eta \Delta'_{11}, \Delta_{12}, x : \mathbf{U}_f\ A; \Gamma$
([Tsh], (10), (7) and $f \oplus f = f$)
(12) $\Delta'_{11}, \Delta_{12}, x : \mathbf{U}_f\ A = \Delta$ ((3), (5) and (8))
(13) $\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \,||\, \mathbf{take}\ x(y_2); P_2\} \vdash_\eta \Delta; \Gamma$ ((11) and (12))

Case: [PSh] left-to-right, $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \,||\, R\} \equiv \mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \,||\, R\}$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_\mathcal{X}\ A$ (2) $\mathbf{put}\ x(y.P); Q \vdash_\eta \Delta_1, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$
(3) $R \vdash_\eta \Delta_2, x : \mathbf{U}_{\mathcal{X}_2}\ A; \Gamma$ (4) $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$, for some $A, \Delta_1, \Delta_2, \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$
([Tsh^{-1}] and $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \,||\, R\} \vdash_\eta \Delta; \Gamma$)
(5) $\mathcal{X}_1 = e$ (6) $\Delta_1 = \Delta_{11}, \Delta_{12}$ (7) $P \vdash_\eta \Delta_{11}, y : \wedge\overline{A}; \Gamma$ (8) $Q \vdash_\eta \Delta_{12}, x : \mathbf{U}_f\ A; \Gamma$
([Tput^{-1}] and (2))
(9) $\mathcal{X}_2 = f$ (10) $\mathcal{X} = e$ ((4) and (5))
(10) $\mathbf{share}\ x\ \{Q \,||\, R\} \vdash_\eta \Delta_{12}, \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ ([Tsh], (8), (3), (9) and $f \oplus f = f$)
(11) $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \,||\, R\} \vdash_\eta \Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_e\ A; \Gamma$ ([Tput], (7) and (10))
(12) $\Delta_{11}, \Delta_{12}, \Delta_2, x : \mathbf{U}_e\ A = \Delta$ ((1), (6) and (10))
(13) $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \,||\, R\} \vdash_\eta \Delta; \Gamma$ ((11) and (12))

Case: [PSh] right-to-left, $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \,||\, R\} \equiv \mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \,||\, R\}$.

(1) $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_e\ A$ (2) $P \vdash_\eta \Delta_1, y : \wedge\overline{A}; \Gamma$
(3) $\mathbf{share}\ x\ \{Q \,||\, R\} \vdash_\eta \Delta_2, x : \mathbf{U}_f\ A; \Gamma$, for some Δ_1, Δ_2, A
([Tput^{-1}] and $\mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \,||\, R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_2 = \Delta_{21}, \Delta_{22}$ (5) $Q \vdash_\eta \Delta_{21}, x : \mathbf{U}_{\mathcal{X}_1}\ A; \Gamma$

(6) $R \vdash_\eta \Delta_{22}, x : \mathbf{U}_{\mathcal{X}_2}\, A; \Gamma$ (7) $\mathcal{X}_1 \oplus \mathcal{X}_2 = f$, for some $\Delta_{21}, \Delta_{22}, \mathcal{X}_1, \mathcal{X}_2$ ([Tsh^{-1}] and (3))
(8) $\mathcal{X}_1 = f = \mathcal{X}_2$ ((7))
(9) **put** $x(y.P); Q \vdash_\eta \Delta_1, \Delta_{21}, x : \mathbf{U}_e\, A; \Gamma$ ([Tput], (2), (5) and (8))
(10) **share** $x\,\{\textbf{put}\, x(y.P); Q \,||\, R\} \vdash_\eta \Delta_1, \Delta_{21}, \Delta_{22}, x : \mathbf{U}_e\, A; \Gamma$ ([Tsh], (9), (6), (8) and $e \oplus f = e$)
(11) $\Delta_1, \Delta_{21}, \Delta_{22}, x : \mathbf{U}_e\, A = \Delta$ ((1) and (4))
(12) **share** $x\ \{\textbf{put}\, x(y.P); Q \,||\, R\} \vdash_\eta \Delta; \Gamma$ ((10) and (11))

Case: [0M] left-to-right, **par** $\{P \,||\, \mathbf{0}\} \equiv P$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $P \vdash_\eta \Delta_1; \Gamma$ (3) $\mathbf{0} \vdash_\eta \Delta_3; \Gamma$, for some Δ_1, Δ_2
([Tmix^{-1}] and **par** $\{P \,||\, \mathbf{0}\} \vdash_\eta \Delta; \Gamma$)
(4)$\Delta_3 = \emptyset$ ([T$\mathbf{0}^{-1}$] and (3))
(5)$\Delta = \Delta_1$ ((1) and (4))
(6)$P \vdash_\eta \Delta; \Gamma$ ((2) and (5))

Case: [0M] right-to-left, $P \equiv$ **par** $\{P \,||\, \mathbf{0}\}$.

(1) $\mathbf{0} \vdash_\eta \emptyset; \Gamma$ ([T$\mathbf{0}$])
(2) **par** $\{P \,||\, \mathbf{0}\} \vdash_\eta \Delta; \Gamma$ ([Tmix], $P \vdash_\eta \Delta; \Gamma$ and (1))

Case: [0Sm] left-to-right, $\mathbf{0} + \mathbf{0} \equiv \mathbf{0}$.

(1) $\mathbf{0} \vdash_\eta \Delta; \Gamma$ ([Tsum^{-1}] and $\mathbf{0} + \mathbf{0} \vdash_\eta \Delta; \Gamma$)

Case: [0Sm] right-to-left, $\mathbf{0} \equiv \mathbf{0} + \mathbf{0}$.

(1) $\mathbf{0} + \mathbf{0} \vdash_\eta \Delta; \Gamma$ ([Tsum] and $\mathbf{0} \vdash_\eta \Delta; \Gamma$)

□

Theorem 1(2)). *If $P \vdash_\eta \Delta; \Gamma$ and $P \longrightarrow Q$, then $Q \vdash_\eta \Delta; \Gamma$.*

Proof. By induction on a derivation tree for $P \longrightarrow Q$ and case analysis on the root rule.

Case: [fwd], **cut** $\{\textbf{fwd}\ x\ y\ \ |y : A|\ \ P\} \longrightarrow \{x/y\}P$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) **fwd** $x\ y\ \vdash_\eta \Delta_1, y : A; \Gamma$ (3) $P \vdash_\eta \Delta_2, y : \overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and **cut** $\{\textbf{fwd}\ x\ y\ \ |y : A|\ \ P\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1, y : A = x : \overline{B}, y : B$, for some B ([Tfwd^{-1}] and (2))
(5) $\Delta_1 = x : \overline{A}$ and $A = B$ ((4))
(6) $\{x/y\}P \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$ (Lemma 31(1) and (3))
(7) $\{x/y\}P \vdash_\eta \Delta_2, \Delta_1; \Gamma$ ((5) and (6))
(8) $\{x/y\}P \vdash_\eta \Delta; \Gamma$ ((1) and (7))

Case: $[\mathbf{1}\bot]$, $\mathbf{cut}\ \{\mathbf{close}\ x\ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\} \longrightarrow P$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{close}\ x \vdash_\eta \Delta_1, x : \mathbf{1}; \Gamma$ (3) $\mathbf{wait}\ x; P \vdash_\eta \Delta_2, x : \bot; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{close}\ x\ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; P\} \vdash_\eta \Delta; \Gamma$)
(3) $\Delta_1 = \emptyset$ ([T$\mathbf{1}^{-1}$] and (2))
(4) $P \vdash_\eta \Delta_2; \Gamma$ ([T$\bot^{-1}$] and (3))
(5) $P \vdash_\eta \Delta; \Gamma$ ((1), (3) and (4))

Case: $[\otimes\parr]$,

$\mathbf{cut}\ \{\mathbf{send}\ x(y.P); Q\ \ |x : A \otimes B|\ \ \mathbf{recv}\ x(z); R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x : B|\ \ (\mathbf{cut}\ \{P\ \ |y : A|\ \ \{y/z\}R\})\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{send}\ x(y.P); Q \vdash_\eta \Delta_1, x : A \otimes B; \Gamma$ (3) $\mathbf{recv}\ x(z); R \vdash_\eta \Delta_2, x : \overline{A} \parr \overline{B}; \Gamma$
for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{send}\ x(y.P); Q\ \ |x : A \otimes B|\ \ \mathbf{recv}\ x(z); R\} \vdash_\eta \Delta; \Gamma$)
(4) $\Delta_1 = \Delta_{11}, \Delta_{12}$ (5) $P \vdash_\eta \Delta_{11}, y : A; \Gamma$ (6) $Q \vdash_\eta \Delta_{12}, x : B; \Gamma$, for some Δ_{11}, Δ_{12}
([T$\otimes^{-1}$] and (2))
(7) $R \vdash_\eta \Delta_2, z : \overline{A}, x : \overline{B}; \Gamma$ ([T$\parr^{-1}$] and (3))
(8) $\{y/z\}R \vdash_\eta \Delta_2, y : \overline{A}, x : \overline{B}; \Gamma$ (Lemma 31(1) and (7))
(9) $\mathbf{cut}\ \{P\ \ |y : A|\ \ \{y/z\}R\} \vdash_\eta \Delta_{11}, \Delta_2, x : \overline{B}; \Gamma$ ([Tcut], (5) and (8))
(10) $\mathbf{cut}\ \{Q\ \ |x : B|\ \ (\mathbf{cut}\ \{P\ \ |y : A|\ \ \{y/z\}R\})\} \vdash_\eta \Delta_{12}, \Delta_{11}, \Delta_2; \Gamma$ ([Tcut], (6) and (9))
(11) $\mathbf{cut}\ \{Q\ \ |x : B|\ \ (\mathbf{cut}\ \{P\ \ |y : A|\ \ \{y/z\}R\})\} \vdash_\eta \Delta; \Gamma$ ((1), (4) and (10))

Case: $[\&\oplus_l]$, $\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inl}; R\} \longrightarrow \mathbf{cut}\ \{P\ \ |x : A|\ \ R\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\} \vdash_\eta \Delta_1, x : A \,\&\, B; \Gamma$
(3) $x.\mathbf{inl}; R \vdash_\eta \Delta_2, x : \overline{A} \oplus \overline{B}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inl}; R\} \vdash_\eta \Delta; \Gamma$
(4) $P \vdash_\eta \Delta_1, x : A$ (5) $Q \vdash_\eta \Delta_1, x : B; \Gamma$ ([T$\&^{-1}$] and (2))
(6) $R \vdash_\eta \Delta_2, x : \overline{A}; \Gamma$ (T$\oplus_l^{-1}$] and (3))
(7) $\mathbf{cut}\ \{P\ \ |x : A|\ \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (4) and (6))
(8) $\mathbf{cut}\ \{P\ \ |x : A|\ \ R\} \vdash_\eta \Delta; \Gamma$ ((1) and (7))

Case: $[\&\oplus_r]$, $\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inr}; R\} \longrightarrow \mathbf{cut}\ \{Q\ \ |x : B|\ \ R\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\} \vdash_\eta \Delta_1, x : A \,\&\, B; \Gamma$
(3) $x.\mathbf{inr}; R \vdash_\eta \Delta_2, x : \overline{A} \oplus \overline{B}; \Gamma$, for some Δ_1, Δ_2
(Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{case}\ x\ \{|\mathsf{inl} : P \mid \mathsf{inr} : Q\}\ \ |x : A \,\&\, B|\ \ x.\mathbf{inl}; R\} \vdash_\eta \Delta; \Gamma$
(4) $P \vdash_\eta \Delta_1, x : A$ (5) $Q \vdash_\eta \Delta_1, x : B; \Gamma$ ([T$\&^{-1}$] and (2))
(6) $R \vdash_\eta \Delta_2, x : \overline{B}; \Gamma$ ([T$\oplus_r^{-1}$] and (3))
(7) $\mathbf{cut}\ \{Q\ \ |x : B|\ \ R\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (5) and (6))

(8) $\mathbf{cut}\ \{P\ \ |x : A|\ \ R\}\ \vdash_\eta \Delta; \Gamma$ ((1) and (7))

Case: [!?], $\mathbf{cut}\ \{!x(y); P\ \ |x :!A|\ \ ?x; Q\} \longrightarrow \mathbf{cut!}\ \{y.P\ |x : A|\ Q\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $!x(y); P \vdash_\eta \Delta_1, x :!A; \Gamma$
(3) $?x; Q \vdash_\eta \Delta_2, x :?\overline{A}; \Gamma$, for some Δ_1, Δ_2 ([Tcut^{-1}] and $\mathbf{cut}\ \{!x(y); P\ \ |x :!A|\ \ ?x; Q\} \vdash_\eta \Delta; \Gamma$
(4) $\Delta_1 = \emptyset$ (5) $P \vdash_\eta y : A; \Gamma$ ([T!$^{-1}$] and (2))
(6) $Q \vdash_\eta \Delta_2; \Gamma, x : \overline{A}$ ([T?$^{-1}$] and (3))
(7) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}\ \vdash_\eta \Delta_2; \Gamma$ ([Tcut!], (5) and (6))
(8) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\}\ \vdash_\eta \Delta; \Gamma$ ((1), (4) and (7))

Case: [call], $\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{call}\ x(z); Q\} \longrightarrow \mathbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ (\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\}$.

(1) $P \vdash_\eta y : A; \Gamma$ (2) $\mathbf{call}\ x(z); Q \vdash_\eta \Delta; \Gamma, x : \overline{A}$
([Tcut!$^{-1}$] and $\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{call}\ x(z); Q\} \vdash_\eta \Delta; \Gamma$
(3) $Q \vdash_\eta \Delta, z : \overline{A}; \Gamma, x : \overline{A}$ ([Tcall^{-1}] and (2))
(4) $\mathbf{cut!}\ \{y.P\ |x : A|\ Q\} \vdash_\eta \Delta, z : \overline{A}; \Gamma$ ([Tcut!], (1) and (3))
(5) $\{z/y\}P\ \vdash_\eta z : A; \Gamma$ (Lemma 31(1) and (1))
(6) $\mathbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ (\mathbf{cut!}\ \{y.P\ |x : A|\ Q\})\} \vdash_\eta \Delta; \Gamma$ ([Tcut], (5) and (4))

Case: [$\exists\forall$], $\mathbf{cut}\ \{\mathbf{sendty}\ x\ A; P\ \ |x : \exists X.B|\ \ \mathbf{recvty}\ x(X); Q\} \longrightarrow \mathbf{cut}\ \{P\ \ |x : \{A/X\}B|\ \ \{A/X\}Q\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{sendty}\ x\ A; P \vdash_\eta \Delta_1, x : \exists X.B; \Gamma$
(3) $\mathbf{recvty}\ x(X); Q \vdash_\eta \Delta_2, x : \forall X.\overline{B}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{sendty}\ x\ A; P\ \ |x : \exists X.B|\ \ \mathbf{recvty}\ x(X); Q\} \vdash_\eta \Delta; \Gamma$
(4) $P \vdash_\eta \Delta_1, x : \{A/X\}B; \Gamma$ ([T$\exists^{-1}$] and (2))
(5) $Q \vdash_\eta \Delta_2, x : \overline{B}; \Gamma$ ([T$\forall^{-1}$] and (3))
(6) $\{A/X\}Q \vdash_\eta \Delta_2, x : \{A/X\}\overline{B}; \Gamma$ (Lemma 31(2) and (5))
(7) $\{A/X\}Q \vdash_\eta \Delta_2, x : \overline{\{A/X\}B}; \Gamma$ ($\overline{\{A/X\}B} = \{A/X\}\overline{B}$ and (6))
(8) $\mathbf{cut}\ \{P\ \ |x : \{A/X\}B|\ \ \{A/X\}Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (4) and (7))
(9) $\mathbf{cut}\ \{P\ \ |x : \{A/X\}B|\ \ \{A/X\}Q\} \vdash_\eta \Delta; \Gamma$ ((1) and (8))

Case: [$\mu\nu$] , $\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x; P\ \ |x : \mu X.\ A|\ \ \mathbf{unfold}_\nu\ x; Q\} \longrightarrow \mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ Q\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{unfold}_\mu\ x; P \vdash_\eta \Delta_1, x : \mu X.\ A; \Gamma$
(3)$\mathbf{unfold}_\nu\ x; Q \vdash_\eta \Delta_2, x : \nu X.\ \{\overline{X}/X\}\overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x; P\ \ |x : \mu X.\ A|\ \ \mathbf{unfold}_\nu\ x; Q\} \vdash \Delta; \Gamma$)
(4) $P \vdash_\eta \Delta_1, x : \{\mu X.\ A/X\}A; \Gamma$ ([Tμ^{-1}] and (2))
(5) $Q \vdash_\eta \Delta_2, x : \{\nu X.\ \{\overline{X}/X\}\overline{A}/X\}(\{\overline{X}/X\}\overline{A}); \Gamma$ ([Tν^{-1}] and (3))
(6) $Q \vdash_\eta \Delta_2, x : \{\mu X.\ A/X\}\overline{A}; \Gamma$ ((5) and (*)
(7) $\mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (5) and (6))

(8) $\mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ Q\} \vdash_\eta \Delta;\Gamma$ ((7) and (1))

To obtain (*):

$$\begin{aligned}\{\nu X.\ \{\overline{X}/X\}\overline{A}/X\}(\{\overline{X}/X\}\overline{A}) &= \overline{\{\nu X.\ \{\overline{X}/X\}\overline{A}/X\}\overline{A})}\\ &= \{(\mu X.\ \{\overline{X}/X\}\overline{\{\overline{X}/X\}\overline{A}})/X\}\overline{A}\\ &= \{(\mu X.\ \{\overline{X}/X\}\{\overline{X}/X\}\overline{\overline{A}})/X\}\overline{A})\\ &= \{(\mu X.\ \{\overline{X}/X\}\{\overline{X}/X\}A)/X\}\overline{A})\\ &= \{\mu X.\ A/X\}\overline{A}\end{aligned}$$

Case: [corec],

$$\begin{aligned}&\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x;P\ \ |x : \mu X.\ A|\ \ \mathbf{corec}\ Y(z,\vec{w});Q\ [x,\vec{y}]\}\\ &\quad\to \mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ \sigma(\{\mathbf{corec}\ Y(z,\vec{w});Q/Y\}Q)\}\end{aligned}$$

where σ is the substitution map given by $\sigma = \{x/z\}\{\vec{y}/\vec{w}\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{unfold}_\mu\ x;P \vdash_\eta \Delta_1, x : \mu X.\ A;\Gamma$
(3) $\mathbf{corec}\ Y(z,\vec{w});Q\ [x,\vec{y}] \vdash_\eta \Delta_2, x : \nu X.\ \{\overline{X}/X\}\overline{A};\Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{unfold}_\mu\ x;P\ \ |x : \mu X.\ A|\ \ \mathbf{corec}\ Y(x,\vec{y});Q\} \vdash \Delta;\Gamma$)
(4) $P \vdash_\eta \Delta_1, x : \{\mu X.\ A/X\}A;\Gamma$ ([Tμ^{-1}] and (2))
(5) $\eta' = \eta, Y(z,\vec{w}) \mapsto \sigma^{-1}(\Delta_2, z : X;\Gamma)$ (6) $Q \vdash_{\eta'} \sigma^{-1}(\Delta_2, z : \{\overline{X}/X\}\overline{A};\Gamma)$
([Tcorec^{-1}] and (3))
(7) $\{\mathbf{corec}\ Y(z,\vec{w});Q/Y\}Q \vdash_\eta \sigma^{-1}(\Delta_2, x : \{\nu X.\ \{\overline{X}/X\}\overline{A}/X\}(\{\overline{X}/X\}\overline{A});\Gamma)$
(Lemma 31(3), (3), (5) and (6))
(8) $\sigma(\{\mathbf{corec}\ Y(z,\vec{w});Q/Y\}Q) \vdash_\eta \Delta_2, x : \{\nu X.\ \{\overline{X}/X\}\overline{A}/X\}(\{\overline{X}/X\}\overline{A});\Gamma$
((7) and since σ^{-1} is the inverse of σ
(9) $\sigma(\{\mathbf{corec}\ Y(x,\vec{y});Q/Y\}Q) \vdash_\eta \Delta_2, x : \{\mu X.\ A/X\}A;\Gamma$ ((8) and (*) from case [$\mu\nu$] above)
(10) $\mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ \{\mathbf{corec}\ Y(x,\vec{y});Q/Y\}Q\} \vdash_\eta \Delta_1, \Delta_2;\Gamma$ ([Tcut], (4) and (9))
(11) $\mathbf{cut}\ \{P\ \ |x : \{\mu X.\ A/X\}A|\ \ \{\mathbf{corec}\ Y(x,\vec{y});Q/Y\}Q\} \vdash_\eta \Delta;\Gamma$ ((1) and (10))

Case: [$\wedge\vee$d], $\mathbf{cut}\ \{\mathbf{affine}_{\vec{c},\vec{a}}\ a;P\ \ |a : \wedge A|\ \ \mathbf{discard}\ a\} \to \mathbf{par}\ \{\mathbf{release}\ \vec{c}\ ||\ \mathbf{discard}\ \vec{a}\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{affine}_{\vec{c},\vec{a}}\ a;P \vdash_\eta \Delta_1, v : \wedge A;\Gamma$ (3) $\mathbf{discard}\ a \vdash_\eta \Delta_2, v : \vee\overline{A};\Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{affine}_{\vec{c},\vec{a}}\ a;P\ \ |a : \wedge A|\ \ \mathbf{discard}\ a\} \vdash_\eta \Delta;\Gamma$
(4) $\Delta_1 = \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}$ (5) $P \vdash_\eta \Delta_1, a : A;\Gamma$, for some $\vec{c}, \vec{B}, \vec{a}, \vec{C}$ ([Taffine^{-1}] and (2))
(6) $\Delta_2 = \emptyset$ ([Tdiscard^{-1}] and (3))
(7) $\mathbf{par}\ \{\mathbf{release}\ \vec{c}\ ||\ \mathbf{discard}\ \vec{a}\} \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C};\Gamma$
([Tmix], [Tdiscard] and [Trelease])
(8) $\vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C} = \Delta$ ((1), (4) and (6))
(9) $\mathbf{par}\ \{\mathbf{release}\ \vec{c}\ ||\ \mathbf{discard}\ \vec{a}\} \vdash_\eta \Delta;\Gamma$ ((7) and (8))

Case: [$\wedge\vee$u], $\mathbf{cut}\ \{\mathbf{affine}_{\vec{c},\vec{a}}\ a; P \ \ |a : \wedge A|\ \ \mathbf{use}\ a; Q\} \longrightarrow \mathbf{cut}\ \{P \ \ |a : A|\ \ Q\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{affine}_{\vec{c},\vec{a}}\ a; P \vdash_\eta \Delta_1, v : \wedge A; \Gamma$
(3) $\mathbf{use}\ a; Q \vdash_\eta \Delta_2, v : \vee\overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{affine}_{\vec{c},\vec{a}}\ a; P \ \ |a : \wedge A|\ \ \mathbf{use}\ a; Q\} \vdash_\eta \Delta; \Gamma$
(4) $\Delta_1 = \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}$ (5) $P \vdash_\eta \Delta_1, a : A; \Gamma$, for some $\vec{c}, \vec{B}, \vec{a}, \vec{C}$ ([Taffine^{-1}] and (2))
(6) $Q \vdash_\eta \Delta_2, a : \overline{A}; \Gamma$ ([Tuse^{-1}] and (3))
(7) $\mathbf{cut}\ \{P \ \ |a : A|\ \ Q\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (5) and (6))
(8) $\mathbf{cut}\ \{P \ \ |a : A|\ \ Q\} \vdash_\eta \Delta; \Gamma$ ((1) and (7))

Case: [$\mathbf{S}_f\ \mathbf{U}_f$ f], $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P) \ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{release}\ c\} \longrightarrow \mathbf{cut}\ \{P \ \ |a : \wedge A|\ \ \mathbf{discard}\ a\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{cell}\ c(a.P) \vdash_\eta \Delta_1, c : \mathbf{S}_f\ A; \Gamma$
(3) $\mathbf{release}\ c \vdash_\eta \Delta_2, c : \mathbf{U}_f\ \overline{A}; \Gamma$, for some$\Delta_1, \Delta_2$
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P) \ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{release}\ c\} \vdash_\eta \Delta; \Gamma$)
(4) $P \vdash_\eta \Delta_1, a : \wedge A; \Gamma$ ([Tcell^{-1}] and (2))
(5) $\Delta_2 = \emptyset$ ([Tfree^{-1}] and (3))
(6) $\mathbf{discard}\ a \vdash_\eta a : \vee\overline{A}; \Gamma$ ([Tdiscard])
(7) $\mathbf{cut}\ \{P \ \ |a : \wedge A|\ \ \mathbf{discard}\ a\} \vdash_\eta \Delta_1; \Gamma$ ([Tcut], (4) and (6))
(8) $\Delta = \Delta_1$ ((1) and (5))
(9) $\mathbf{cut}\ \{P \ \ |a : \wedge A|\ \ \mathbf{discard}\ a\} \vdash_\eta \Delta; \Gamma$ ((7) and (8))

Case: [$\mathbf{S}_f\ \mathbf{U}_f$ t],

$\mathbf{cut}\ \{\mathbf{cell}\ c(a.P) \ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{take}\ c(a'); Q\} \longrightarrow \mathbf{cut}\ \{\{a'/a\}P \ \ |a' : \wedge A|\ \ (\mathbf{cut}\ \{\mathbf{empty}\ c \ \ |c : \mathbf{S}_e\ A|\ \ Q\})\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{cell}\ c(a.P) \vdash_\eta \Delta_1, c : \mathbf{S}_f\ A; \Gamma$
(3) $\mathbf{take}\ c(a'); Q \vdash_\eta \Delta_2, c : \mathbf{U}_f\ \overline{A}; \Gamma$, for some$\Delta_1, \Delta_2$
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P) \ \ |c : \mathbf{S}_f\ A|\ \ \mathbf{take}\ c(a'); Q\} \vdash_\eta \Delta; \Gamma$)
(4) $P \vdash_\eta \Delta_1, a : \wedge A; \Gamma$ ([Tcell^{-1}] and (2))
(5) $Q \vdash_\eta \Delta_2, a : \vee\overline{A}, c : \mathbf{U}_e\ \overline{A}; \Gamma$ ([Ttake^{-1}] and (3))
(6) $\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma$ ([Tempty])
(7) $\mathbf{cut}\ \{\mathbf{empty}\ c \ \ |c : \mathbf{S}_e\ A|\ \ Q\} \vdash_\eta \Delta_2, a : \vee\overline{A}; \Gamma$ ([Tcut], (6) and (5))
(8) $\{a'/a\}P \vdash_\eta \Delta_1, a :' \wedge A; \Gamma$ (Lemma 29(Tsubs) and (4))
(9) $\mathbf{cut}\ \{\{a'/a\}P \ |a' : \wedge A|\ (\mathbf{cut}\ \{\mathbf{empty}\ c \ |c : \mathbf{S}_e\ A|\ Q\})\} \vdash_\eta \Delta_1, \Delta_2; \Gamma$ ([Tcut], (8) and (7))
(10) $\mathbf{cut}\ \{\{a'/a\}P \ \ |a' : \wedge A|\ \ (\mathbf{cut}\ \{\mathbf{empty}\ c \ \ |c : \mathbf{S}_e\ A|\ \ Q\})\} \vdash_\eta \Delta; \Gamma$ ((1) and (9))

Case: [$\mathbf{S}_e\ \mathbf{U}_e$], $\mathbf{cut}\ \{\mathbf{empty}\ c \ \ |c : \mathbf{S}_e\ A|\ \ \mathbf{put}\ c(a.P); Q\} \longrightarrow \mathbf{cut}\ \{\mathbf{cell}\ c(a.P) \ \ |c : \mathbf{S}_f\ A|\ \ Q\}$.

(1) $\Delta = \Delta_1, \Delta_2$ (2) $\mathbf{empty}\ c \vdash_\eta \Delta_1, c : \mathbf{S}_e\ A; \Gamma$
(3) $\mathbf{put}\ c(a.P); Q \vdash_\eta \Delta_2, c : \mathbf{U}_e\ \overline{A}; \Gamma$, for some Δ_1, Δ_2
([Tcut^{-1}] and $\mathbf{cut}\ \{\mathbf{empty}\ c \ \ |c : \mathbf{S}_e\ A|\ \ \mathbf{put}\ c(a.P); Q\} \vdash_\eta \Delta; \Gamma$)

(4) $\Delta_1 = \emptyset$ ([Tempty^{-1}] and (2))
(5) $\Delta_2 = \Delta_{21}, \Delta_{22}$ (6) $P \vdash_\eta \Delta_{21}, a : \wedge A; \Gamma$ (7) $Q \vdash_\eta \Delta_{22}, c : \mathbf{U}_f\, \overline{A}; \Gamma$ ([Tput^{-1}] and (3))
(8) $\mathbf{cell}\ c(a.P) \vdash_\eta \Delta_{21}, c : \mathbf{S}_f\, A; \Gamma$ ([Tcell] and (6))
(9) $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ |c : \mathbf{S}_f\, A|\ Q\} \vdash_\eta \Delta_{21}, \Delta_{22}; \Gamma$ ([Tcut], (8) and (7))
(10) $\Delta = \Delta_{21}, \Delta_{22}$ ((1), (4) and (5))
(11) $\mathbf{cut}\ \{\mathbf{cell}\ c(a.P)\ |c : \mathbf{S}_f\, A|\ Q\} \vdash_\eta \Delta; \Gamma$ ((9) and (10))

Case: [≡], $P \equiv P'$ and $P' \to Q'$ and $Q' \equiv Q \supset P \to Q$.

(1) $P' \vdash_\eta \Delta; \Gamma$ (Theorem 1(1)), $P \vdash_\eta \Delta; \Gamma$ and $P \equiv P'$)
(2) $Q' \vdash_\eta \Delta; \Gamma$ (i.h., (1) and $P' \to Q'$)
(3) $Q \vdash_\eta \Delta; \Gamma$ (Theorem 1(1)), (2) and $Q' \equiv P$)

Case: [cong], $P \to Q \supset C[P] \to C[Q]$.

(1) $P \vdash_\eta \Delta'; \Gamma'$, for some Δ', Γ' (Lemma 28 and $C[P] \vdash_\eta \Delta; \Gamma$)
(2) $Q \vdash_\eta \Delta'; \Gamma'$ (i.h., (1) and $P \to Q$)
(3) $C[Q] \vdash_\eta \Delta; \Gamma$ (Lemma 28 , (1), (2) and $C[P] \vdash_\eta \Delta; \Gamma$)

□

B

Progress

We prove properties Lemma 3(1)-(7) of the observability predicate. Then, we conclude with the proof of liveness Lemma 4.

Lemma 3(1). *Let* $P \vdash \Delta, x : \boldsymbol{U}_f\ A; \Gamma$ *and* $Q \vdash \Delta', x : \boldsymbol{U}_f\ A; \Gamma$ *be processes for which* $P \downarrow_{x:act}$ *and* $Q \downarrow_{x:act}$. *Then,* $\mathbf{share}\ x\ \{P \parallel Q\} \downarrow_{x:act}$.

Proof. By double induction on derivation trees for $P \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$. For the base cases we apply either one of $\equiv$ rules [RSh] or [TSh] in order to expose an observable action. For the inductive cases we consider that we are given a derivation tree for $P \downarrow_x$. This is w.l.o.g. since $\mathbf{share}\ x\ \{P \parallel Q\} \equiv \mathbf{share}\ x\ \{Q \parallel P\}$. For cases [mix], [cut], [cut!], [share] we commute the share on x with the principal form of P by applying either $\equiv$ rule [ShM], [CSh], [ShC!], [ShSh] or [ShSm]. The inductive case [$\equiv$] follows immediately because the relation $\equiv$ is a congruence, i.e. satisfies $\equiv$ rule [cong].

Case: The root rule of both $P \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$ is [act]. We have

$$\frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_{x:\text{act}}}\ [\text{act}] \qquad \frac{s(\mathcal{B}) = x}{\mathcal{B} \downarrow_{x:\text{act}}}\ [\text{act}]$$

where $P = \mathcal{A}$ and $Q = \mathcal{B}$.

Since the subject of both actions $\mathcal{A}, \mathcal{B}$ - x - has the type $\mathbf{U}_f\ A$ (in the linear typing context), we conclude that $\mathcal{A}, \mathcal{B}$ are either release or take actions.

Case: $\mathcal{A} = \mathbf{release}\ x$.

By applying $\equiv$ rule [RSh] we obtain

$$\mathbf{share}\ x\ \{P \parallel Q\} = \mathbf{share}\ x\ \{\mathbf{release}\ x \parallel Q\} \equiv Q$$

Hence

$$\frac{\mathbf{share}\ x\ \{P \parallel Q\} \equiv Q \qquad \dfrac{\vdots}{Q \downarrow_{x:\text{act}}}}{\mathbf{share}\ x\ \{P \parallel Q\} \downarrow_{x:\text{act}}}\ [\equiv]$$

Case: $\mathcal{B} = \textbf{release}\ x$. Similar to case $\mathcal{A} = \textbf{release}\ x$.

Case: $\mathcal{A} = \textbf{take}\ x(y); P'$ and $\mathcal{B} = \textbf{take}\ x(z); Q'$.

By applying $\equiv$ rule [TSh] we obtain

$$\textbf{share}\ x\ \{\textbf{take}\ x(y); P' \mid\mid \textbf{take}\ x(z); Q'\} \equiv \textbf{take}\ x(y); R_1 + \textbf{take}\ x(z); R_2, \text{ where}$$
$$R_1 = \textbf{share}\ x\ \{P' \mid\mid \textbf{take}\ x(z); Q'\}$$
$$R_2 = \textbf{share}\ x\ \{\textbf{take}\ x(z); P' \mid\mid Q'\} \qquad (\equiv \text{[TSh]})$$

Hence

$$\dfrac{\textbf{share}\ x\ \{P \mid\mid Q\} \equiv \textbf{take}\ x(y); R_1 + \textbf{take}\ x(z); R_2 \qquad \dfrac{\dfrac{s(\textbf{take}\ x(y); R_1) = x}{\textbf{take}\ x(y); R_1 \downarrow_{x:\text{act}}}\ [\text{act}]}{(\textbf{take}\ x(y); R_1 + \textbf{take}\ x(z); R_2) \downarrow_{x:\text{act}}}\ [\text{sum}]}{\textbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:\text{act}}}\ [\equiv]$$

In fact, there is other possible derivation for $\textbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:\text{act}}$

$$\dfrac{\textbf{share}\ x\ \{P \mid\mid Q\} \equiv \textbf{take}\ x(z); R_2 + \textbf{take}\ x(y); R_1 \qquad \dfrac{\dfrac{s(\textbf{take}\ x(z); R_2) = x}{\textbf{take}\ x(z); R_2 \downarrow_{x:\text{act}}}\ [\text{act}]}{(\textbf{take}\ x(z); R_2 + \textbf{take}\ x(y); R_1) \downarrow_{x:\text{act}}}\ [\text{sum}]}{\textbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:\text{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [mix].

Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [mix]. We have

$$\dfrac{P_1 \downarrow_{x:\text{act}}}{(\textbf{par}\ \{P_1 \mid\mid P_2\}) \downarrow_{x:\text{act}}}\ [\text{mix}]$$

where $P = \textbf{par}\ \{P_1 \mid\mid P_2\}$.

Since $\textbf{par}\ \{P_1 \mid\mid P_2\} \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ for which $P_1 \vdash \Delta_1, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash \Delta_2; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_{x:\text{act}}$.

We have

$$\begin{aligned}\textbf{share}\ x\ \{P \mid\mid Q\} &= \textbf{share}\ x\ \{(\textbf{par}\ \{P_1 \mid\mid P_2\}) \mid\mid Q\} \\ &\equiv \textbf{par}\ \{\underbrace{\textbf{share}\ x\ \{P_1 \mid\mid Q\}}_{R} \mid\mid P_2\} \qquad (\equiv \text{[ShM]}, x \in \text{fn}(P_1))\end{aligned}$$

By induction on $P_1 \downarrow_x$ and $Q \downarrow_x$ we conclude that $R \downarrow_{x:\text{act}}$.

Hence

$$\dfrac{\textbf{share}\ x\ \{P \mid\mid Q\} \equiv \textbf{par}\ \{R \mid\mid P_2\} \qquad \dfrac{R \downarrow_{x:\text{act}}}{(\textbf{par}\ \{R \mid\mid P_2\}) \downarrow_{x:\text{act}}}\ [\text{mix}]}{(\textbf{share}\ x\ \{P \mid\mid Q\}) \downarrow_{x:\text{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\mathrm{act}}$ or the root rule of $Q \downarrow_{x:\mathrm{act}}$ is [cut].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\mathrm{act}}$ is [cut]. We have

$$\frac{P_1 \downarrow_{x:\mathrm{act}} \quad y \neq x}{\mathbf{cut}\ \{P_1\ \ |y|\ \ P_2\} \downarrow_{x:\mathrm{act}}}\ [\mathrm{cut}]^S$$

where $P = \mathbf{cut}\ \{P_1\ \ |y|\ \ P_2\}$.

Since $\mathbf{cut}\ \{P_1\ \ |y|\ \ P_2\} \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ and a type B for which $P_1 \vdash \Delta_1, y : \overline{B}, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash \Delta_2, y : B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_{x:\mathrm{act}}$.

We have

$$\begin{aligned}\mathbf{share}\ x\ \{P \mid\mid Q\} &= \mathbf{share}\ x\ \{(\mathbf{cut}\ \{P_1\ \ |y|\ \ P_2\}) \mid\mid Q\} \\ &\equiv \mathbf{cut}\ \{\underbrace{\mathbf{share}\ x\ \{P_1 \mid\mid Q\}}_{R}\ \ |y|\ \ P_2\} \qquad (\equiv [\mathrm{CSh}], x, y \in \mathrm{fn}(P_1))\end{aligned}$$

By induction on $P_1 \downarrow_{x:\mathrm{act}}$ and $Q \downarrow_{x:\mathrm{act}}$ we conclude that $(\mathbf{share}\ x\ \{P_1 \mid\mid Q\}) \downarrow_{x:\mathrm{act}}$.

Hence

$$\frac{\mathbf{share}\ x\ \{P \mid\mid Q\} \equiv \mathbf{cut}\ \{R\ \ |y|\ \ P_2\} \qquad \dfrac{R \downarrow_x \quad y \neq x}{(\mathbf{cut}\ \{R\ \ |y|\ \ P_2\}) \downarrow_{x:\mathrm{act}}}\ [\mathrm{cut}]}{(\mathbf{share}\ x\ \{P \mid\mid Q\}) \downarrow_{x:\mathrm{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\mathrm{act}}$ or the root rule of $Q \downarrow_{x:\mathrm{act}}$ is [cut!].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\mathrm{act}}$ is [cut!]. We have

$$\frac{P_2 \downarrow_{x:\mathrm{act}} \quad z \neq x}{\mathbf{cut!}\ \{y.P_1\ |z : B|\ P_2\} \downarrow_{x:\mathrm{act}}}\ [\mathrm{cut!}]$$

where $P = \mathbf{cut!}\ \{y.P_1\ |z : B|\ P_2\}$.

Since $\mathbf{cut!}\ \{y.P_1\ |z : B|\ P_2\} \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that $P_1 \vdash y : \overline{B}; \Gamma$ and $P_2 \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma, z : B$.

We have

$$\begin{aligned}\mathbf{share}\ x\ \{P \mid\mid Q\} &= \mathbf{share}\ x\ \{(\mathbf{cut!}\ \{y.P_1\ |z : B|\ P_2\}) \mid\mid Q\} \\ &\equiv \mathbf{cut!}\ \{y.P_1\ |z : B|\ \underbrace{(\mathbf{share}\ x\ \{P_2 \mid\mid Q\})}_{R}\} \qquad (\equiv [\mathrm{ShC!}]\ z \notin \mathrm{fn}(Q))\end{aligned}$$

By induction on $P_2 \downarrow_{x:\mathrm{act}}$ and $Q \downarrow_{x:\mathrm{act}}$ we conclude that $R \downarrow_{x:\mathrm{act}}$.

Hence

$$\frac{\mathbf{share}\ x\ \{P \mid\mid Q\} \equiv \mathbf{cut!}\ \{y.P_1\ |z|\ R\} \qquad \dfrac{R \downarrow_{x:\mathrm{act}} \quad z \neq x}{(\mathbf{cut!}\ \{y.P_1\ |z|\ R\}) \downarrow_{x:\mathrm{act}}}\ [\mathrm{cut!}]}{(\mathbf{share}\ x\ \{P \mid\mid Q\}) \downarrow_{x:\mathrm{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [share].

Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [share]. We have

$$\frac{P_1 \downarrow_{x:\text{act}} \quad y \neq x}{\textbf{share}\ y\ \{P_1 \parallel P_2\} \downarrow_{x:\text{act}}}\ [\text{share}]$$

where $P = \textbf{share}\ y\ \{P_1 \parallel P_2\}$.

The root rule of a derivation for $\textbf{share}\ y\ \{P_1 \parallel P_2\} \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ can be either [Tsh], [TshL] or [TshR]. We assume w.l.o.g. it is [Tsh]. The proof works in the same way for the other cases [TshL] and [TshR].

By inverting [Tsh] on $\textbf{share}\ y\ \{P_1 \parallel P_2\} \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ, a type B for which $P_1 \vdash \Delta_1, y : \mathbf{U}_f\ B, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash \Delta_2, y : \mathbf{U}_f\ B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_{x:\text{act}}$.

We have

$$\begin{aligned}\textbf{share}\ x\ \{P \parallel Q\} &= \textbf{share}\ x\ \{\textbf{share}\ y\ \{P_1 \parallel P_2\} \parallel Q\} \\ &\equiv \textbf{share}\ y\ \{\underbrace{\textbf{share}\ x\ \{P_1 \parallel Q\}}_{R} \parallel P_2\} \qquad (\equiv [\text{ShSh}], x, y \in \text{fn}(P_1))\end{aligned}$$

By induction on $P_1 \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$ we conclude that $(\textbf{share}\ x\ \{P_1 \parallel Q\}) \downarrow_{x:\text{act}}$. Hence

$$\frac{\textbf{share}\ x\ \{P \parallel Q\} \equiv \textbf{share}\ y\ \{R \parallel P_2\} \qquad \dfrac{R \downarrow_{x:\text{act}} \quad y \neq x}{(\textbf{share}\ y\ \{R \parallel P_2\}) \downarrow_{x:\text{act}}}\ [\text{share}]}{(\textbf{share}\ x\ \{P \parallel Q\}) \downarrow_{x:\text{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [sum].

Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [sum]. We have

$$\frac{P_1 \downarrow_{x:\text{act}}}{P_1 + P_2 \downarrow_{x:\text{act}}}\ [\text{sum}]$$

where $P = P_1 + P_2$.

Since $P_1 + P_2 \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ we conclude that $P_1 \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$.

We have

$$\begin{aligned}\textbf{share}\ x\ \{P \parallel Q\} &= \textbf{share}\ x\ \{(P_1 + P_2) \parallel Q\} \\ &\equiv \underbrace{\textbf{share}\ x\ \{P_1 \parallel Q\}}_{R} + \textbf{share}\ x\ \{P_2 \parallel Q\} \qquad (\equiv [\text{D-ShSm}])\end{aligned}$$

By induction on $P_1 \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$ we conclude that $R \downarrow_{x:\text{act}}$.

Hence

$$\frac{\mathbf{share}\ x\ \{P \mid\mid Q\} \equiv R + \mathbf{share}\ x\ \{P_2 \mid\mid Q\} \qquad \frac{R \downarrow_{x:\mathrm{act}}}{(R + \mathbf{share}\ x\ \{P_2 \mid\mid Q\}) \downarrow_{x:\mathrm{act}}}\ [\mathrm{sum}]}{(\mathbf{share}\ x\ \{P \mid\mid Q\}) \downarrow_{x:\mathrm{act}}}\ [\equiv]$$

Case: Either the root rule of $P \downarrow_{x:\mathrm{act}}$ or the root rule of $Q \downarrow_{x:\mathrm{act}}$ is [≡].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\mathrm{act}}$ is [≡]. We have

$$\frac{P \equiv P' \quad P' \downarrow_{x:\mathrm{act}}}{P \downarrow_{x:\mathrm{act}}}\ [\equiv]$$

Since $P \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$, $P \equiv P'$ and structural congruence preserves typing (Theorem 1), then $P' \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$.

By induction on $P' \downarrow_{x:\mathrm{act}}, Q \downarrow_{x:\mathrm{act}}$, we conclude that $\mathbf{share}\ x\ \{P' \mid\mid Q\} \downarrow_{x:\mathrm{act}}$.

Observe that

$$\mathbf{share}\ x\ \{P \mid\mid Q\} \equiv \mathbf{share}\ x\ \{P' \mid\mid Q\} \qquad (\equiv [\mathrm{cong}])$$

Hence

$$\frac{\mathbf{share}\ x\ \{P \mid\mid Q\} \equiv \mathbf{share}\ x\ \{P' \mid\mid Q\} \quad \mathbf{share}\ x\ \{P' \mid\mid Q\} \downarrow_{x:\mathrm{act}}}{\mathbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:\mathrm{act}}}\ [\equiv]$$

□

Lemma 3(2). *Let* $P \vdash \Delta, x : \mathbf{U}_e\ A; \Gamma, Q \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$. *If* $P \downarrow_{x:act}$, *then* $\mathbf{share}\ x\ \{P \mid\mid Q\} \downarrow_{x:act}$.

Proof. By induction on the structure of a derivation for $P \downarrow_{x:act}$ and case analysis on the root rule. The base case [act] follows by applying ≡ rule [PSh] in order to expose the put action. For the inductive cases [mix], [cut], [cut!], [share], [sum] and [≡] see the proof of Lemma 3(1).

Case: The root rule of both $P \downarrow_{x:\mathrm{act}}$ is [act]. We have

$$\frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_{x:\mathrm{act}}}\ [\mathrm{act}]$$

where $P = \mathcal{A}$.

Since the subject of action $\mathcal{A}$- x - has the type $\mathbf{U}_e\ A$ (in the linear typing context), we conclude that $\mathcal{A}$ is a put action, i.e. $\mathcal{A} = \mathbf{put}\ x(y.P_1); P_2$ for some y, P_1, P_2.

By applying ≡ rule [PSh] we obtain

$$\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P_1); P_2 \mid\mid Q\} \equiv \mathbf{put}\ x(y.P_1); \underbrace{\mathbf{share}\ x\ \{P_2 \mid\mid Q\}}_{R} \qquad (\equiv [\mathrm{PSh}])$$

Hence

$$\dfrac{\textbf{share}\ x\ \{P \parallel Q\} \equiv \textbf{put}\ x(y.R); \qquad \dfrac{s(\textbf{put}\ x(y.P_1); R) = x}{\textbf{put}\ x(y.P_1); R \downarrow_{x:\text{act}}}\ [\text{act}]}{\textbf{share}\ x\ \{P \parallel Q\} \downarrow_{x:\text{act}}}\ [\equiv]$$

□

Lemma 3(3). *Let* $P \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma, Q \vdash \Delta, x : \mathbf{U}_e\ A; \Gamma$. *If* $Q \downarrow_{x:act}$, *then* $\textbf{share}\ x\ \{P \parallel Q\} \downarrow_{x:act}$.

Proof. Applying Lemma 3(2) to $Q \vdash \Delta, x : \mathbf{U}_e\ A; \Gamma$ and $P \vdash \Delta, x : \mathbf{U}_f\ A; \Gamma$ yields $\textbf{share}\ x\ \{Q \parallel P\} \downarrow_{x:\text{act}}$.

By ≡ rule [Sh] we have $\textbf{share}\ x\ \{P \parallel Q\} \equiv \textbf{share}\ x\ \{Q \parallel P\}$.

Hence,

$$\dfrac{\textbf{share}\ x\ \{P \parallel Q\} \equiv \textbf{share}\ x\ \{Q \parallel P\} \qquad \textbf{share}\ x\ \{Q \parallel P\} x : \text{act}}{(\textbf{share}\ x\ \{P \parallel Q\}) \downarrow_{x:\text{act}}}\ [\equiv]$$

□

Lemma 3(4). *Let* $P \vdash \Delta, x : \overline{A}; \Gamma$ *and* $Q \vdash \Delta', x : A; \Gamma$ *be processes for which* $P \downarrow_{x:act}$ *and* $Q \downarrow_{x:act}$. *Then,* $\textbf{cut}\ \{P\ \ |x|\ \ Q\}$ *reduces.*

Proof. By double induction on derivation trees for $P \downarrow_{x:\text{act}}$ and $Q \downarrow_{x:\text{act}}$. For the base cases we apply one of the principal cut reductions. For the inductive cases we consider that we are given a derivation tree for $P \downarrow_x$. This is w.l.o.g. since $\textbf{cut}\ \{P\ \ |x|\ \ Q\} \equiv \textbf{cut}\ \{Q\ \ |x|\ \ P\}$. For cases [mix], [cut], [cut!], [share] we commute the cut on x with the principal form of P by applying either ≡ rule [CM], [CC], [CC!], [CSh] or [CSm]. The inductive case $P \downarrow_x$ rule [≡] follows immediately because the relation $\rightarrow$ is closed by structural congruence, i.e. satisfies $\rightarrow$ rule [≡].

Case: The root rule of both $P \downarrow_x$ and $Q \downarrow_x$ is [act]. We have

$$\dfrac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_x}\ [\text{act}] \qquad \dfrac{s(\mathcal{B}) = x}{\mathcal{B} \downarrow_x}\ [\text{act}]$$

where $P = \mathcal{A}$ and $Q = \mathcal{B}$.

Since $\mathcal{A} \vdash \Delta, x : \overline{A}; \Gamma$ and $\mathcal{B} \vdash \Delta, x : A; \Gamma$ we conclude that $\mathcal{A}, \mathcal{B}$ is a pair of dual actions with the same subject. Hence, $\textbf{cut}\ \{P\ \ |x|\ \ Q\}$ reduces by applying one of the principal cut reductions.

For example, if $A = \bot$, we have

$$\mathcal{A} = \textbf{close}\ x \ \text{ and } \ \mathcal{B} = \textbf{wait}\ x; Q'$$

Consequently

$$\textbf{cut}\ \{\textbf{close}\ x\ \ |x|\ \ \textbf{wait}\ x; Q'\} \rightarrow Q' \qquad (\rightarrow [\mathbf{1}\bot])$$

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [mix].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [mix]. We have

$$\frac{P_1 \downarrow_x}{(\textbf{par}\ \{P_1 \parallel P_2\}) \downarrow_x}\ [\text{mix}]$$

where $P = \textbf{par}\ \{P_1 \parallel P_2\}$.

Since $\textbf{par}\ \{P_1 \parallel P_2\} \vdash \Delta, x : \overline{A}; \Gamma$ we conclude that there exists a partition Δ_1, Δ_2 of Δ s.t. $P_1 \vdash \Delta_1, x : \overline{A}; \Gamma$ and $P_2 \vdash \Delta_2; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_x$.

Then

$$\begin{aligned} \textbf{cut}\ \{P\ \ |x|\ \ Q\} &= \textbf{cut}\ \{(\textbf{par}\ \{P_1 \parallel P_2\})\ \ |x|\ \ Q\} \\ &\equiv \textbf{par}\ \{(\textbf{cut}\ \{P_1\ \ |x|\ \ Q\}) \parallel P_2\} \qquad (\equiv [\text{CM}], x \in \text{fn}(P_1)) \end{aligned}$$

By induction on $P_1 \downarrow_x$ and $Q \downarrow_x$ we conclude that $\textbf{cut}\ \{P_1\ \ |x|\ \ Q\}$, and hence $\textbf{par}\ \{(\textbf{cut}\ \{P_1\ \ |x|\ \ Q\}) \parallel P_2\}$, reduces.

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [cut].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [cut]. We have

$$\frac{P_1 \downarrow_x \quad y \neq x}{(\textbf{cut}\ \{P_1\ \ |y|\ \ P_2\}) \downarrow_x}\ [\text{cut}]$$

where $P = \textbf{cut}\ \{P_1\ \ |y|\ \ P_2\}$.

Since $\textbf{cut}\ \{P_1\ \ |y|\ \ P_2\} \vdash \Delta, x : \overline{A}; \Gamma$ we conclude that there exists a partition Δ_1, Δ_2 of Δ and a type B s.t. $P_1 \vdash \Delta_1, x : \overline{A}, y : \overline{B}; \Gamma$ and $P_2 \vdash \Delta_2, y : B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_x$.

Then

$$\begin{aligned} \textbf{cut}\ \{P\ \ |x|\ \ Q\} &= \textbf{cut}\ \{(\textbf{cut}\ \{P_1\ \ |y|\ \ P_2\})\ \ |x|\ \ Q\} \\ &\equiv \textbf{cut}\ \{(\textbf{cut}\ \{P_1\ \ |x|\ \ Q\})\ \ |y|\ \ P_2\} \qquad (\equiv [\text{CC}], x, y \in \text{fn}(P_1)) \end{aligned}$$

By induction on $P_1 \downarrow_x$ and $Q \downarrow_x$ we conclude that $\textbf{cut}\ \{P_1\ \ |x|\ \ Q\}$, and hence $\textbf{cut}\ \{(\textbf{cut}\ \{P_1\ \ |x|\ \ Q\})\ \ |y|\ \ P_2\}$, reduces.

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [cut!].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [cut!]. We have

$$\frac{P_2 \downarrow_x \quad z \neq x}{(\textbf{cut!}\ \{y.P_1\ |z|\ P_2\}) \downarrow_x}\ [\text{cut!}]$$

where $P = \textbf{cut!}\ \{y.P_1\ |z|\ P_2\}$.

Since $\mathbf{cut!}\ \{y.P_1\ |z|\ P_2\} \vdash \Delta, x : \overline{A}; \Gamma$ we conclude that there exists a type B s.t. $P_1 \vdash y : \overline{B}; \Gamma$ and $P_2 \vdash \Delta, x : \overline{A}; \Gamma, z : B$.

Then

$$\begin{aligned}\mathbf{cut}\ \{P\ \ |x|\ \ Q\} &= \mathbf{cut}\ \{(\mathbf{cut!}\ \{y.P_1\ |z|\ P_2\})\ \ |x|\ \ Q\} \\ &\equiv \mathbf{cut!}\ \{y.P_1\ |z|\ (\mathbf{cut}\ \{P_2\ \ |x|\ \ Q\})\} \qquad (\equiv \text{[CC!]}, z \notin \text{fn}(Q))\end{aligned}$$

By induction on $P_2 \downarrow_x$ and $Q \downarrow_x$ we conclude that $\mathbf{cut}\ \{P_2\ \ |x|\ \ Q\}$, and hence $\mathbf{cut!}\ \{y.P_1\ |z|\ (\mathbf{cut}\ \{P_2\ \ |x|\ \ Q\})\}$, reduces.

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [share].

Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [share]. We have

$$\frac{P_1 \downarrow_x \quad y \neq x}{(\mathbf{share}\ y\ \{P_1\ ||\ P_2\}) \downarrow_x}\ \text{[share]}$$

where $P = \mathbf{share}\ y\ \{P_1\ ||\ P_2\}$.

The root rule of a derivation for $\mathbf{share}\ y\ \{P_1\ ||\ P_2\} \vdash \Delta, x : \overline{A}; \Gamma$ can be either [Tsh], [TshL] or [TshR]. We assume w.l.o.g. it is [Tsh]. The proof works in the same way for the other cases [TshL] and [TshR].

By inverting [Tsh] on $\mathbf{share}\ y\ \{P_1\ ||\ P_2\} \vdash \Delta, x : \overline{A}; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ, a type B for which $P_1 \vdash \Delta_1, y : \mathbf{U}_f\ B, x : \overline{A}; \Gamma$ and $P_2 \vdash \Delta_2, y : \mathbf{U}_f\ B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_{x:\text{act}}$.

Then

$$\begin{aligned}\mathbf{cut}\ \{P\ \ |x|\ \ Q\} &= \mathbf{cut}\ \{(\mathbf{share}\ y\ \{P_1\ ||\ P_2\})\ \ |x|\ \ Q\} \\ &\equiv \mathbf{share}\ y\ \{(\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\})\ ||\ P_2\} \qquad (\equiv \text{[CSh]}, x, y \in \text{fn}(P_1))\end{aligned}$$

By induction on $P_1 \downarrow_x$ and $Q \downarrow_x$ we conclude that $\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\}$, and hence $\mathbf{share}\ y\ \{(\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\})\ ||\ P_2\}$, reduces.

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [sum].

Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [sum]. We have

$$\frac{P_1 \downarrow_x}{(P_1 + P_2) \downarrow_x}\ \text{[sum]}$$

where $P = P_1 + P_2$.

Since $P_1 + P_2 \vdash \Delta, x : \overline{A}; \Gamma$ we conclude that $P_1 \vdash \Delta, x : \overline{A}; \Gamma$ and $P_2 \vdash \Delta, x : \overline{A}; \Gamma$.

Then

$$\begin{aligned}\mathbf{cut}\ \{P\ \ |x|\ \ Q\} &= \mathbf{cut}\ \{(P_1 + P_2)\ \ |x|\ \ Q\} \\ &\equiv (\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\}) + (\mathbf{cut}\ \{P_2\ \ |x|\ \ Q\}) \qquad (\equiv \text{[D-CSm]})\end{aligned}$$

By induction on $P_1 \downarrow_x$ and $Q \downarrow_x$ we conclude that $\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\}$, and hence $(\mathbf{cut}\ \{P_1\ \ |x|\ \ Q\}) + (\mathbf{cut}\ \{P_2\ \ |x|\ \ Q\})$, reduces.

Case: Either the root rule of $P \downarrow_{x:\text{act}}$ or the root rule of $Q \downarrow_{x:\text{act}}$ is [≡].
Suppose w.l.o.g. that the root rule of $P \downarrow_{x:\text{act}}$ is [≡]. We have

$$\frac{P \equiv P' \quad P' \downarrow_x}{P \downarrow_x}$$

Observe that since $P \vdash \Delta, x : \overline{A}; \Gamma$, $P \equiv P'$ and structural congruence preserves typing (Theorem 1), then $P' \vdash \Delta, x : \overline{A}; \Gamma$.

By induction on $P' \downarrow_x$, $Q \downarrow_x$ we conclude that $\mathbf{cut}\ \{P'\ \ |x|\ \ Q\}$ reduces. Since $\mathbf{cut}\ \{P\ \ |x|\ \ Q\} \equiv \mathbf{cut}\ \{P'\ \ |x|\ \ Q\}$, $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ reduces as well (rule $\longrightarrow$ [≡]).

□

Lemma 3(5)). *Let $P \vdash \Delta, x : \overline{A}; \Gamma$, $Q \vdash \Delta', x : A; \Gamma$ be processes for which $P \downarrow_{x:fwd}$. Then, $\mathbf{cut}\ \{P\ \ |x|\ \ Q\}$ reduces.*

Proof. By induction on a derivation trees for $P \downarrow_{x:\text{fwd}}$. We handle the base case, which follows by applying the principal cut conversion $\longrightarrow$ [fwd]. For the inductive cases see the proof of Lemma 3(4).

Case [fwd]
We have

$$\frac{}{\mathbf{fwd}\ x\ y \downarrow_x}\ [\text{fwd}]$$

where $P = \mathbf{fwd}\ x\ y$.

Then

$$\begin{aligned} \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ \ |x|\ \ Q\} &\equiv \mathbf{cut}\ \{\mathbf{fwd}\ y\ x\ \ |x|\ \ Q\} && (\equiv [\text{fwd}]) \\ &\longrightarrow \{y/x\}Q && (\longrightarrow [\text{fwd}]) \end{aligned}$$

□

Lemma 3(6). *Let $P \vdash y : \overline{A}; \Gamma$ and $Q \vdash \Delta; \Gamma, x : A$ be processes for which $Q \downarrow_x$. Then, $\mathbf{cut!}\ \{y.P\ |x|\ Q\}$ reduces.*

Proof. By induction on a derivation tree for $Q \downarrow_x$ and case analysis on the root rule. The base case [act] follows by applying the principal cut conversion $\longrightarrow$ [call]. The inductive cases [mix], [cut], [cut!], [share] and [sum] follow by distributing the unrestricted cut over the arguments of Q (with ≡ rules [D-C!M], [D-C!C], [D-C!C!], [D-C!Sh] or [C!Sm]) and then apply the inductive hypothesis. The inductive case [≡] follows because reduction $\longrightarrow$ is closed by structural congruence, i.e. satisfies rule $\longrightarrow$ [≡].

Case: The root rule of $Q \downarrow_x$ is [act]. We have

$$\frac{s(\mathcal{A}) = x}{\mathcal{A} \downarrow_x}$$

where $Q = \mathcal{A}$.

Since $\mathcal{A} \vdash \Delta; \Gamma, x : A$, we have $\mathcal{A} = \mathbf{call}\ x(z); Q'$, for some Q'. Hence

$$\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{call}\ x(z); Q'\} \to \mathbf{cut}\ \{\{z/y\}P\ \ |z|\ \ (\mathbf{cut!}\ \{y.P\ |x|\ Q'\})\} \qquad (\to [\text{call}])$$

Case: The root rule of $Q \downarrow_x$ is [mix]. We have

$$\frac{Q_1 \downarrow_x}{(\mathbf{par}\ \{Q_1\ ||\ Q_2\}) \downarrow_x}$$

where $Q = \mathbf{par}\ \{Q_1\ ||\ Q_2\}$.

Since $\mathbf{par}\ \{Q_1\ ||\ Q_2\}\ \vdash \Delta; \Gamma, x : A$, there exists a partition Δ_1, Δ_2 of Δ for which $Q_1 \vdash \Delta_1; \Gamma, x : A$ and $Q_2 \vdash \Delta_2; \Gamma, x : A$.

We have

$$\begin{aligned}\mathbf{cut!}\ \{y.P\ |x|\ Q\} &= \mathbf{cut!}\ \{y.P\ |x|\ (\mathbf{par}\ \{Q_1\ ||\ Q_2\})\} \\ &\equiv \mathbf{par}\ \{(\mathbf{cut!}\ \{y.P\ |x|\ Q_1\})\ ||\ (\mathbf{cut!}\ \{y.P\ |x|\ Q_2\})\} \qquad (\equiv [\text{D-C!M}])\end{aligned}$$

By induction on $Q_1 \downarrow_x$ we conclude that $\mathbf{cut!}\ \{y.P\ |x|\ Q_1\}$, and hence $\mathbf{cut!}\ \{y.P\ |x|\ Q\}$, reduces.

Case: The root rule of $Q \downarrow_x$ is [cut]. We have

$$\frac{Q_1 \downarrow_x \quad z \neq x}{(\mathbf{cut}\ \{Q_1\ \ |z|\ \ Q_2\}) \downarrow_x}$$

where $Q = \mathbf{cut}\ \{Q_1\ \ |z|\ \ Q_2\}$.

Since $\mathbf{cut}\ \{Q_1\ \ |z|\ \ Q_2\}\ \vdash \Delta; \Gamma, x : A$, there exists a partition Δ_1, Δ_2 of Δ and a type B for which $Q_1 \vdash \Delta_1, z : \overline{B}; \Gamma, x : A$ and $Q_2 \vdash \Delta_2, z : B; \Gamma, x : A$.

We have

$$\begin{aligned}\mathbf{cut!}\ \{y.P\ |x|\ Q\} &= \mathbf{cut!}\ \{y.P\ |x|\ (\mathbf{cut}\ \{Q_1\ \ |z|\ \ Q_2\})\} \\ &\equiv \mathbf{cut}\ \{(\mathbf{cut!}\ \{y.P\ |x|\ Q_1\})\ \ |z|\ \ (\mathbf{cut!}\ \{y.P\ |x|\ Q_2\})\} \qquad (\equiv [\text{D-C!C}])\end{aligned}$$

By induction on $Q_1 \downarrow_x$ we conclude that $\mathbf{cut!}\ \{y.P\ |x|\ Q_1\}$, and hence $\mathbf{cut!}\ \{y.P\ |x|\ Q\}$, reduces.

Case: The root rule of $Q \downarrow_x$ is [cut!]. We have

$$\frac{Q_2 \downarrow_x \quad z \neq x}{(\textbf{cut!}\ \{w.Q_1\ |z|\ Q_2\}) \downarrow_x}$$

where $Q = \textbf{cut!}\ \{w.Q_1\ |z|\ Q_2\}$.

Since $\textbf{cut!}\ \{w.Q_1\ |z|\ Q_2\} \vdash \Delta; \Gamma, x : A$, we conclude that exists a type B for which $Q_1 \vdash w : \overline{B}; \Gamma, x : A$ and $Q_2 \vdash \Delta; \Gamma, z : B, x : A$.

We have

$$\begin{aligned}\textbf{cut!}\ \{y.P\ |x|\ Q\} &= \textbf{cut!}\ \{y.P\ |x|\ (\textbf{cut!}\ \{w.Q_1\ |z|\ Q_2\})\} \\ &\equiv \textbf{cut!}\ \{w.(\textbf{cut!}\ \{y.P\ |x|\ Q_1\})\ |z|\ (\textbf{cut!}\ \{y.P\ |x|\ Q_2\})\} \quad (\equiv \text{[D-C!C!]})\end{aligned}$$

By induction on $Q_2 \downarrow_x$ we conclude that $\textbf{cut!}\ \{y.P\ |x|\ Q_2\}$, and hence $\textbf{cut!}\ \{y.P\ |x|\ Q\}$, reduces.

Case: The root rule of $Q \downarrow_x$ is [share]. We have

$$\frac{Q_1 \downarrow_x \quad z \neq x}{(\textbf{share}\ z\ \{Q_1 \mid\mid Q_2\}) \downarrow_x}$$

where $Q = \textbf{share}\ z\ \{Q_1 \mid\mid Q_2\}$.

Since $\textbf{share}\ z\ \{Q_1 \mid\mid Q_2\} \vdash \Delta; \Gamma, x : A$, there are state flavours $\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}$ and a partition $\Delta_1, \Delta_2, z : \mathbf{U}_{\mathcal{X}}\ B$ of Δ for which $Q_1 \vdash \Delta_1, z\mathbf{U}_{\mathcal{X}_1}\ B; \Gamma, x : A$,$Q_2 \vdash \Delta_2, z : \mathbf{U}_{\mathcal{X}_2}\ B; \Gamma, x : A$ and $\mathcal{X}_1 \oplus \mathcal{X}_2 = \mathcal{X}$.

The root rule of a derivation for $\textbf{share}\ y\ \{P_1 \mid\mid P_2\} \vdash \Delta; \Gamma, x : A$ can be either [Tsh], [TshL] or [TshR]. We assume w.l.o.g. it is [Tsh]. The proof works in the same way for the other cases [TshL] and [TshR].

By inverting [Tsh] on $\textbf{share}\ y\ \{P_1 \mid\mid P_2\} \vdash\vdash \Delta; \Gamma, x : A$ we conclude that exists a partition Δ_1, Δ_2 of Δ, a type B for which $P_1 \vdash \Delta_1, y : \mathbf{U}_f\ B; \Gamma, x : A$ and $P_2 \vdash \Delta_2, y : \mathbf{U}_f\ B; \Gamma, x : A$.

We have

$$\begin{aligned}\textbf{cut!}\ \{y.P\ |x|\ Q\} &= \textbf{cut!}\ \{y.P\ |x|\ (\textbf{share}\ z\ \{Q_1 \mid\mid Q_2\})\} \\ &\equiv \textbf{share}\ z\ \{(\textbf{cut!}\ \{y.P\ |x|\ Q_1\}) \mid\mid (\textbf{cut!}\ \{y.P\ |x|\ Q_2\})\} \quad (\equiv \text{[D-C!Sh]})\end{aligned}$$

By induction on $Q_1 \downarrow_x$ we conclude that $\textbf{cut!}\ \{y.P\ |x|\ Q_1\}$, and hence $\textbf{cut!}\ \{y.P\ |x|\ Q\}$ reduces.

Case: The root rule of $Q \downarrow_x$ is [sum]. We have

$$\frac{Q_1 \downarrow_x}{(Q_1 + Q_2) \downarrow_x}$$

where $Q = Q_1 + Q_2$.

Since $\mathbf{par}\ \{Q_1 \mid\mid Q_2\} \vdash \Delta; \Gamma, x : A$, then $Q_1 \vdash \Delta; \Gamma, x : A$ and $Q_2 \vdash \Delta; \Gamma, x : A$.

We have

$$\begin{aligned} \mathbf{cut!}\ \{y.P \mid x \mid Q\} &= \mathbf{cut!}\ \{y.P \mid x \mid (Q_1 + Q_2)\} \\ &\equiv (\mathbf{cut!}\ \{y.P \mid x \mid Q_1\}) + (\mathbf{cut!}\ \{y.P \mid x \mid Q_2\}) \qquad (\equiv \text{[D-C!Sm]}) \end{aligned}$$

By induction on $Q_1 \downarrow_x$ we conclude that $\mathbf{cut!}\ \{y.P \mid x \mid Q_1\}$, and hence $\mathbf{cut!}\ \{y.P \mid x \mid Q\}$, reduces.

Case: The root rule of $Q \downarrow_x$ is [≡]. We have

$$\frac{Q \equiv Q' \quad Q' \downarrow_x}{Q \downarrow_x}$$

Observe that since $Q \vdash \Delta; \Gamma, x : A$, $Q \equiv Q'$ and structural congruence preserves typing (Theorem 1), we have $Q' \vdash \Delta; \Gamma, x : A$.

By induction on $Q' \downarrow_x$ we conclude that $\mathbf{cut!}\ \{y.P \mid x \mid Q'\}$ reduces. Since $\mathbf{cut!}\ \{y.P \mid x \mid Q\} \equiv \mathbf{cut!}\ \{y.P \mid x \mid Q'\}$, $\mathbf{cut!}\ \{y.P \mid x \mid Q\}$ reduces as well ($\rightarrow$ rule [≡]).

□

Lemma 3(7). *Let $P \vdash \Delta, x : A; \Gamma$ and suppose that $A \neq \mathbf{S}_{\mathcal{X}}\ B$. If $P \downarrow_{x:fwd}$, then either (i) $P \downarrow_{y:fwd}$ for some $y : \overline{A} \in \Delta$ or (ii) P reduces.*

Proof. The proof is by structural induction on the derivation tree $P \downarrow_{x:\mathrm{fwd}}$ and case analysis on the root rule.

Case: The root rule of $P \downarrow_{x:\mathrm{fwd}}$ is [fwd].

We have

$$\frac{}{\mathbf{fwd}\ x\ y \downarrow_{x:\mathrm{fwd}}}\ [\mathrm{fwd}]$$

where $P = \mathbf{fwd}\ x\ y$.

By inversion on $\mathbf{fwd}\ x\ y \vdash \Delta, x : A; \Gamma$ we conclude that $\Delta = y : \overline{A}$.

Observe that

$$\mathbf{fwd}\ x\ y \equiv \mathbf{fwd}\ y\ x \qquad (\equiv \text{[fwd]})$$

Then

$$\frac{\mathbf{fwd}\ x\ y \equiv \mathbf{fwd}\ y\ x \qquad \dfrac{}{\mathbf{fwd}\ y\ x \downarrow_{y:\mathrm{fwd}}}\ [\mathrm{fwd}]}{\mathbf{fwd}\ x\ y \downarrow_{y:\mathrm{fwd}}}\ [\equiv]$$

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [mix].

We have

$$\frac{P_1 \downarrow_{x:\text{fwd}}}{(\textbf{par}\ \{P_1 \parallel P_2\}) \downarrow_{x:\text{fwd}}}\ [\text{mix}]$$

where $P = \textbf{par}\ \{P_1 \parallel P_2\}$.

By inversion on the typing judgment $\textbf{par}\ \{P_1 \parallel P_2\} \vdash \Delta, x : A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ s.t. $P_1 \vdash \Delta_1, x : A; \Gamma$ and $P_2 \vdash \Delta_2; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2 because $P_1 \downarrow_x$.

By induction on $P_1 \downarrow_{x:\text{fwd}}$, we conclude that either (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1$ or (ii) P_1 reduces.

Case (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1$.

Then

$$\frac{P_1 \downarrow_{y:\text{fwd}}}{(\textbf{par}\ \{P_1 \parallel P_2\}) \downarrow_{y:\text{fwd}}}\ [\text{mix}]$$

Furthermore, since $y : \overline{A} \in \Delta_1$ and $\Delta = \Delta_1, \Delta_2$, then $y : \overline{A} \in \Delta$.

Case (ii) P_1 reduces.

Since reduction is a congruence, then $\textbf{par}\ \{P_1 \parallel P_2\}$ reduces as well.

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [cut].

We have

$$\frac{P_1 \downarrow_{x:\text{fwd}} \quad z \neq x}{(\textbf{cut}\ \{P_1\ \ |z|\ \ P_2\}) \downarrow_{x:\text{fwd}}}\ [\text{cut}]$$

where $P = \textbf{cut}\ \{P_1\ \ |z|\ \ P_2\}$.

By inversion on the typing judgment $\textbf{cut}\ \{P_1\ \ |z|\ \ P_2\} \vdash \Delta, x : A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ and a type B s.t. $P_1 \vdash \Delta_1, x : A, z : \overline{B}; \Gamma$ and $P_2 \vdash \Delta_2, z : B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2 because $P_1 \downarrow_x$.

By induction on $P_1 \downarrow_{x:\text{fwd}}$, we conclude that either (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1, z : \overline{B}$ or (ii) P_1 reduces. There are three cases to consider, depending on wether (i-i) $y \neq z$ or (i-ii) $y = z$.

Case (i-i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1$.

Then

$$\frac{P_1 \downarrow_{y:\text{fwd}} \quad y \neq z}{(\textbf{cut}\ \{P_1\ \ |z|\ \ P_2\}) \downarrow_{y:\text{fwd}}}\ [\text{cut}]$$

Furthermore, since $y : \overline{A} \in \Delta_1$ and $\Delta = \Delta_1, \Delta_2$, then $y : \overline{A} \in \Delta$.

Case (i-ii) $P_1 \downarrow_{z:\text{fwd}}$ and $y = z$.

By Lemma 3(5), we conclude that $\textbf{cut}\ \{P_1\ |z|\ P_2\}$ reduces.

Case (ii) P_1 reduces.

Since reduction is a congruence, then $\textbf{cut}\ \{P_1\ |z|\ P_2\}$ reduces as well.

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [cut!].

We have

$$\frac{P_1 \downarrow_{x:\text{fwd}} \quad z \neq x}{(\textbf{cut!}\ \{w.P_1\ |z|\ P_2\}) \downarrow_{x:\text{fwd}}}\ [\text{cut!}]$$

where $P = \textbf{cut!}\ \{w.P_1\ |z|\ P_2\}$.

By inversion on the typing judgment $\textbf{cut!}\ \{w.P_1\ |z|\ P_2\} \vdash \Delta, x : A; \Gamma$ we conclude that exists a type B s.t. $P_1 \vdash w : \overline{B}; \Gamma$ and $P_2 \vdash \Delta, x : A; \Gamma, z : B$.

By induction on $P_2 \downarrow_{x:\text{fwd}}$, we conclude that either (i) $P_2 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$ or (ii) P_2 reduces.

Case (i) $P_2 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$. Then

$$\frac{P_2 \downarrow_{y:\text{fwd}} \quad y \neq z}{(\textbf{cut!}\ \{w.P_1\ |z|\ P_2\}) \downarrow_{y:\text{fwd}}}\ [\text{cut!}]$$

Case (ii) P_2 reduces.

Since reduction is a congruence, then $\textbf{cut!}\ \{w.P_1\ |z|\ P_2\}$ reduces as well.

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [share].

We have

$$\frac{P_1 \downarrow_{x:\text{fwd}} \quad z \neq x}{(\textbf{share}\ z\ \{P_1\ ||\ P_2\}) \downarrow_{x:\text{fwd}}}\ [\text{share}]$$

where $P = \textbf{share}\ z\ \{P_1\ ||\ P_2\}$.

The root rule of a derivation for $\textbf{share}\ y\ \{P_1\ ||\ P_2\} \vdash \Delta, x : A; \Gamma$ can be either [Tsh], [TshL] or [TshR]. We assume w.l.o.g. it is [Tsh]. The proof works in the same way for the other cases [TshL] and [TshR].

By inverting [Tsh] on $\textbf{share}\ y\ \{P_1\ ||\ P_2\} \vdash \Delta, x : A; \Gamma$ we conclude that exists a partition Δ_1, Δ_2 of Δ, a type B for which $P_1 \vdash \Delta_1, z : \mathbf{U}_f\ B, x : A; \Gamma$ and $P_2 \vdash \Delta_2, z : \mathbf{U}_f\ B; \Gamma$. Observe that x lies in the linear typing context of P_1 and not of P_2, because $P_1 \downarrow_{x:\text{act}}$.

By induction on $P_1 \downarrow_{x:\text{fwd}}$, we conclude that either (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1, z : \mathbf{U}_f\ B$ or (ii) P_1 reduces.

Notice that, by hypothesis, $\overline{A} \neq \mathbf{U}_f\ B$. Hence, $y : \overline{A} \in \Delta_1$.

There are then two cases to consider.

Case (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta_1$.

Then

$$\frac{P_1 \downarrow_{y:\text{fwd}} \quad y \neq z}{(\textbf{share}\ z\ \{P_1 \mid\mid P_2\}) \downarrow_{y:\text{fwd}}}\ [\text{share}]$$

Case (ii) P_1 reduces

Since reduction is a congruence, then **share** $z\ \{P_1 \mid\mid P_2\}$ reduces as well.

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [sum].

We have

$$\frac{P_1 \downarrow_{x:\text{fwd}}}{(\textbf{par}\ \{P_1 \mid\mid P_2\}) \downarrow_{x:\text{fwd}}}\ [\text{sum}]$$

where $P = P_1 + P_2$.

By inversion on the typing judgment $P_1 + P_2 \vdash \Delta, x : A; \Gamma$ we conclude that $P_1 \vdash \Delta, x : A; \Gamma$ and $P_2 \vdash \Delta, x : A; \Gamma$.

By induction on $P_1 \downarrow_{x:\text{fwd}}$, we conclude that either (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$ or (ii) P_1 reduces.

Case (i) $P_1 \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$.

Then

$$\frac{P_1 \downarrow_{y:\text{fwd}}}{(P_1 + P_2) \downarrow_{y:\text{fwd}}}\ [\text{sum}]$$

Case (ii) P_1 reduces.

Since reduction is a congruence, then $P_1 + P_2$ reduces as well.

Case: The root rule of $P \downarrow_{x:\text{fwd}}$ is [≡].

We have

$$\frac{P \equiv Q \quad Q \downarrow_{x:\text{fwd}}}{P \downarrow_{x:\text{fwd}}}\ [\equiv]$$

Since $P \vdash \Delta, x : A; \Gamma$ and $P \equiv Q$, then $Q \vdash \Delta, x : A; \Gamma$.

By induction on $Q \downarrow_{x:\text{fwd}}$ we conclude that either (i) $Q \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$ or (ii) Q reduces.

Case (i) $Q \downarrow_{y:\text{fwd}}$ for some $y : \overline{A} \in \Delta$.

Then

$$\frac{P \equiv Q \quad Q \downarrow_{y:\text{fwd}}}{P \downarrow_{y:\text{fwd}}}\ [\equiv]$$

Case (ii) Q reduces.

Since reduction is closed by structural congruence, then P reduces as well.

□

Lemma 4. *Let $P \vdash_\emptyset \Delta; \Gamma$ be such that P is live. Either $P \downarrow_x$, for some x, or P reduces.*

Proof. The proof is by structural induction on derivation tree for $P \vdash_\emptyset \Delta; \Gamma$ and case analysis on the root rule.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [T**0**].

We have

$$\frac{}{\mathbf{0} \vdash_\emptyset \emptyset; \Gamma}\ [\mathrm{T}\mathbf{0}]$$

where $P = \mathbf{0}$. Holds vacuously because $\mathbf{0}$ is not live.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tfwd].

We have

$$\frac{}{\mathbf{fwd}\ x\ y \vdash_\emptyset x : \overline{A}, y : A; \Gamma}\ [\mathrm{Tfwd}]$$

Then

$$\frac{}{(\mathbf{fwd}\ x\ y) \downarrow_x}\ [\mathrm{fwd}]$$

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [T**1**].

We have

$$\frac{}{\mathbf{close}\ x \vdash_\emptyset x : \mathbf{1}; \Gamma}\ [\mathrm{T}\mathbf{1}]$$

where $P = \mathbf{close}\ x$. Observe that $\mathbf{close}\ x$ is an action. Then

$$\frac{s(\mathbf{close}\ x) = x}{\mathbf{close}\ x \downarrow_x}\ [\mathrm{act}]$$

Similarly for the the other rules which introduce an action: [T⊥], [T⊗], [T⅋], [T$\oplus_l$], [T$\oplus_r$], [T&], [T?], [T!], [Tcall], [T∃], [T∀], [Tcorec], [Tμ], [Tν], [Taffine], [Tuse], [Tdiscard], [Tcell], [Tempty], [Trelease], [Ttake], [Tput].

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tvar].

We have

$$\frac{\eta = \eta', X(\vec{y}) \ \mapsto \Delta'; \Gamma'}{X(\vec{x}) \vdash_\emptyset \{\vec{x}/\vec{y}\}(\Delta'; \Gamma')}\ [\mathrm{Tvar}]$$

where $P = X(\vec{x})$. Holds vacuously because assumes a nonempty η context.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tmix].

We have

$$\frac{P_1 \vdash_\emptyset \Delta_1; \Gamma \quad P_2 \vdash_\emptyset \Delta_2; \Gamma}{\mathbf{par}\ \{P_1 \mid\mid P_2\} \vdash_\emptyset \Delta_1, \Delta_2; \Gamma}\ [\text{Tmix}]$$

where $P = \mathbf{par}\ \{P_1 \mid\mid P_2\}$ and $\Delta = \Delta_1, \Delta_2$.

Since $\mathbf{par}\ \{P_1 \mid\mid P_2\}$ is live, then either P_1 is live or P_2 is live.

Suppose w.l.o.g. that P_1 is live. By induction on $P_1 \vdash_\emptyset \Delta_1; \Gamma$ we conclude that either $P_1 \downarrow_x$ or P_1 reduces.

Case $P_1 \downarrow_x$

Then

$$\frac{P_1 \downarrow_x}{(\mathbf{par}\ \{P_1 \mid\mid P_2\}) \downarrow_x}\ [\text{mix}]$$

Case P_1 reduces

Then, $\mathbf{par}\ \{P_1 \mid\mid P_2\}$ reduces because of $\longrightarrow$ rule [cong].

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tcut].

We have

$$\frac{P_1 \vdash_\emptyset \Delta_1, x : \overline{A}; \Gamma \quad P_2 \vdash_\emptyset \Delta_2, x : A; \Gamma}{\mathbf{cut}\ \{P_1 \ \mid x \mid\ P_2\} \vdash_\emptyset \Delta_1, \Delta_2; \Gamma}\ [\text{cut}]$$

where $P = \mathbf{cut}\ \{P_1 \ \mid x \mid\ P_2\}$ and $\Delta = \Delta_1, \Delta_2$.

Since both P_1 and P_2 have a nonempty linear typing context, we conclude that both P_1 and P_2 are live (lemma 1).

By applying the i.h. to $P_1 \vdash_\emptyset \Delta_1, x : \overline{A}; \Gamma$ and $P_2 \vdash_\emptyset \Delta_2, x : A; \Gamma$ we conclude that

- $P_1 \downarrow_y$ or P_1 reduces, and
- $P_2 \downarrow_z$ or P_2 reduces

We have the following cases to consider

Case ($P_1 \downarrow_y$ and $y \neq x$) or ($P_2 \downarrow_z$ and $z \neq x$)

Suppose w.l.o.g. that $P_1 \downarrow_y$ and $y \neq x$.

Then

$$\frac{P_1 \downarrow_y \quad y \neq x}{(\mathbf{cut}\ \{P_1 \ \mid x \mid\ P_2\}) \downarrow_y}\ [\text{cut}]$$

Case $P_1 \downarrow_x$ and $P_2 \downarrow_x$

We have the following two cases

Case $P_1 \downarrow_{x:\text{fwd}}$ or $P_2 \downarrow_{x:\text{fwd}}$

Suppose w.l.o.g. that $P_1 \downarrow_{x:\text{fwd}}$.

Then, by lemma 3(3), we conclude that $\mathbf{cut}\ \{P_1\ |x|\ P_2\}$ reduces.

Case $P_1 \downarrow_{x:\text{act}}$ and $P_2 \downarrow_{x:\text{act}}$

Then, by lemma 3(2), we conclude that $\mathbf{cut}\ \{P_1\ |x|\ P_2\}$ reduces.

Case P_1 reduces or P_2 reduces

Because of $\longrightarrow$ rule [cong], $\mathbf{cut}\ \{P_1\ |x|\ P_2\}$ reduces.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tcut!].

We have

$$\frac{P_1 \vdash_\emptyset y : \overline{B}; \Gamma \quad P_2 \vdash_\emptyset \Delta; \Gamma, x : A}{\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\} \vdash_\emptyset \Delta; \Gamma}\ [\text{cut!}]$$

where $P = \mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}$.

Since $\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}$ is live, then P_2 is live.

By induction on $P_2 \vdash_\emptyset \Delta; \Gamma, x : A$ we conclude that either $P_2 \downarrow_z$ or P_2 reduces.

Case $P_2 \downarrow_z$ and $z \neq x$

Then

$$\frac{P_2 \downarrow_z \quad z \neq x}{(\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}) \downarrow_z}\ [\text{cut!}]$$

Case $P_2 \downarrow_x$

Then, $\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}$ reduces (lemma 3(4)).

Case P_2 reduces

Because of $\longrightarrow$ rule [cong], $\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}$ reduces.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [Tsh].

We have

$$\frac{P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_f\ A; \Gamma \quad P_2 \vdash_\emptyset \Delta_2, x : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ x\ \{P_1 \mid\mid P_2\} \vdash_\emptyset \Delta_1, \Delta_2, x : \mathbf{U}_f\ A; \Gamma}\ [\text{Tsh}]$$

where $P = \mathbf{share}\ x\ \{P_1 \mid\mid P_2\}$ and $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_f\ A$.

Since both P_1 and P_2 type with a nonempty linear context and an empty η, then both P_1 and P_2 are live (Lemma 1).

By applying the i.h. to $P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_f\ A; \Gamma$ and $P_2 \vdash_\emptyset \Delta_2, x : \mathbf{U}_f\ A; \Gamma$ we conclude both

- $P_1 \downarrow_y$ or P_1 reduces, and
- $P_2 \downarrow_z$ or P_2 reduces

We have the following cases to consider.

Case A $(P_1 \downarrow_y \text{ and } y \neq x)$ or $(P_2 \downarrow_z \text{ and } z \neq x)$
Suppose w.l.o.g. that $P_1 \downarrow_y$ and $y \neq x$.
Then

$$\frac{P_1 \downarrow_y \quad y \neq x}{(\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}) \downarrow_y}\ [\text{share}]$$

Case B $P_1 \downarrow_x$ and $P_2 \downarrow_x$
We have the following two cases.

Case B1 $P_1 \downarrow_{x:\text{fwd}}$ or $P_2 \downarrow_{x:\text{fwd}}$
Suppose w.l.o.g. that $P_1 \downarrow_{x:\text{fwd}}$.
Observe that x occurs typed by $\mathbf{U}_f\ A$ in the linear typing context of P_1.
Hence, we can apply Lemma 3(7) in order to conclude that either (i) $P_1 \downarrow_y$ for $y \neq x$ or (ii) P_1 reduces. If (i) go to case A. If (ii), go to case C.

Case B2 $P_1 \downarrow_{x:\text{act}}$ and $P_2 \downarrow_{x:\text{act}}$.
Then $(\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}) \downarrow_x$ (Lemma 3(1)).

Case C P_1 reduces or P_2 reduces
Because of $\longrightarrow$ rule [cong], $\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}$ reduces.

Case: The root rule of $P \vdash_\emptyset \Delta; \Gamma$ is [TshL].
We have

$$\frac{P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_e\ A; \Gamma \quad P_2 \vdash_\emptyset \Delta_2, x : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ x\ \{P_1 \mid\mid P_2\} \vdash_\emptyset \Delta_1, \Delta_2, x : \mathbf{U}_e\ A; \Gamma}\ [\text{TshL}]$$

where $P = \mathbf{share}\ x\ \{P_1 \mid\mid P_2\}$ and $\Delta = \Delta_1, \Delta_2, x : \mathbf{U}_e\ A$.

By applying the i.h. to $P_1 \vdash_\emptyset \Delta_1, x : \mathbf{U}_e\ A; \Gamma$ we conclude that either $P_1 \downarrow_y$ or P_1 reduces.

We have the following cases to consider.

Case A $P_1 \downarrow_y$ and $y \neq x$
Then

$$\frac{P_1 \downarrow_y \quad y \neq x}{(\mathbf{share}\ x\ \{P_1 \mid\mid P_2\}) \downarrow_y}\ [\text{share}]$$

Case B $P_1 \downarrow_x$
We have the following two cases.

Case B1 $P_1 \downarrow_{x:\text{fwd}}$
Suppose w.l.o.g. that $P_1 \downarrow_{x:\text{fwd}}$.
Observe that x occurs typed by $\mathbf{U}_e\ A$ in the linear typing context of P_1.
Hence, we can apply Lemma 3(7) in order to conclude that either (i) $P_1 \downarrow_y$ for $y \neq x$ or (ii) P_1 reduces. If (i) go to case A. If (ii), go to case C.

Case B2 $P_1 \downarrow_{x:\mathrm{act}}$.

Then (**share** x $\{P_1 \mid\mid P_2\}$) $\downarrow_x$ (Lemma 3(2)).

Case C P_1 reduces or P_2 reduces

Because of $\longrightarrow$ rule [cong], **share** x $\{P_1 \mid\mid P_2\}$ reduces.

Case: The root rule of $P \vdash_\emptyset \Delta;\Gamma$ is [TshR].

Similar to case [TshL].

Case: The root rule of $P \vdash_\emptyset \Delta;\Gamma$ is [Tsum].

We have

$$\frac{P_1 \vdash_\emptyset \Delta;\Gamma \quad P_2 \vdash_\emptyset \Delta;\Gamma}{P_1 + P_2 \vdash_\emptyset \Delta;\Gamma}\ [\mathrm{Tsum}]$$

where $P = P_1 + P_2$.

Since $P_1 + P_2$ is live, then either P_1 is live or P_2 is live.

Assume w.l.o.g. that P_1 is live. By induction on $P_1 \vdash_\emptyset \Delta;\Gamma$ we conclude that either $P_1 \downarrow_x$ or P_1 reduces.

Case $P_1 \downarrow_x$

Then

$$\frac{P_1 \downarrow_x}{(P_1 + P_2) \downarrow_x}\ [\mathrm{sum}]$$

Case P_1 reduces

Then, $P_1 + P_2$ reduces because of $\longrightarrow$ rule [cong].

□

C

Confluence

Diamond Property for $\rightarrow_d$

We prove Lemma 6 and Lemma 7.

Lemma 6 (Predex Decomposition)**.** *For each process $P \vdash_\eta \Delta;\Gamma$ there are predexes $R_1, \ldots, R_k$ and a static context C such that $P \equiv_d C[R_1, \ldots, R_k]$ and $C \nrightarrow$.*

Proof. By induction on the structure of the typing derivation tree for $P \vdash \Delta;\Gamma$. We perform case analysis on the root rule.

Case: [T0]

Then $C = -$ and $k = 0$.

Similar analysis for the rules which introduce an action or a forwarder or the recurion process variable.

Case: [Tmix]

Let $P = \mathbf{par}\ \{P_1 \parallel P_2\}$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

Let $C = \mathbf{par}\ \{C_1 \parallel C_2\}$.

We have $P \equiv_d C[R_1, \ldots R_p, S_1, \ldots, S_q]$, with $k = p + q$.

Furthermore, since $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$, then $C \nrightarrow_d$.

Case: [Tsum]

Let $P = P_1 + P_2$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

Let $C = C_1 + C_2$.

We have $P \equiv_d C[R_1, \ldots R_p, S_1, \ldots, S_q]$, with $k = p + q$.

Furthermore, $C \nrightarrow_d$.

Case: [Tsh]

Let $P = \textbf{share}\ x\ \{P_1 \mid\mid P_2\}$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

Let $C = \textbf{share}\ x\ \{C_1 \mid\mid C_2\}$.

We have $P \equiv_d C[R_1, \ldots R_p, S_1, \ldots, S_q]$, with $k = p + q$.

Furthermore, since $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$, then $C \nrightarrow_d$.

Cases: [TshL], [TshR]

Similar to case [Tsh].

Case: [Tcut]

Let $P = \textbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$.

By i.h., $P_1 \equiv_d C_1[R_1, \ldots, R_p]$ and $P_2 \equiv_d C_2[S_1, \ldots, S_q]$, where $C_1 \nrightarrow_d$ and $C_2 \nrightarrow_d$.

We have the following cases to consider.

Case: Either $C_1 = C_1'[\textbf{fwd}\ x\ z]$ for some $1 \leq i \leq p$ and static context C_1' or $C_2 = C_2'[\textbf{fwd}\ x\ z]$ for some $1 \leq j \leq q$ and static context C_2'.
Suppose w.l.o.g. that $C_1 = C_1'[\textbf{fwd}\ x\ z]$ for some $1 \leq i \leq p$ and static context C_1'
We have two cases to consider, depending on whether (i) $\textbf{fwd}\ x\ z$ is guarded by either a share or (ii) not.

If (i), then cut $\textbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex. So simply let $C = \textbf{cut}\ \{C_1\ \ |x|\ \ C_2\}$.

Suppose (ii). Then

$$\begin{aligned} P &= \textbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \textbf{cut}\ \{C_1'[R_1, \ldots, R_p, \textbf{fwd}\ x\ z]\ \ |x|\ \ C_2[S_1, \ldots, S_q]\} \\ &\equiv_d C_1'[R_1, \ldots, R_p, \textbf{cut}\ \{\textbf{fwd}\ x\ z\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}] \end{aligned}$$

with $k = p$.

Observe that $\textbf{cut}\ \{\textbf{fwd}\ x\ z\ \ |x|\ \ C_2[S_1, \ldots, S_q]\}$ is a predex.

Case: Either $R_i = \textbf{cut}\,\{\textbf{fwd}\,x\,z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}$ or $Q_j = \textbf{cut}\,\{\textbf{fwd}\,x\,z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}$ for some $1 \leq i \leq p, 1 \leq j \leq q$.

Suppose w.l.o.g. that $R_i = \textbf{cut}\ \{\textbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1, \ldots, M_n]\}$.

We have two cases to consider, depending on whether(i) $\textbf{fwd}\ x\ z$ is guarded by either a share in C_1 or (ii) not.

If (i), then cut $\textbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex. Let simply $C = \textbf{cut}\ \{C_1\ \ |x|\ \ C_2\}$.

Suppose (ii). Then

$$\begin{aligned} P &= \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \mathbf{cut}\ \{C_1[R_1,\ldots,\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1,\ldots,M_n]\},\ldots,R_p]\ \ |x|\ \ C_2[S_1,\ldots,S_q]\} \\ &\equiv_D C_1[R_1,\ldots,\mathbf{cut}\ \{\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1,\ldots,M_n]\}\ \ |x|\ \ C_2[S_1,\ldots,S_q]\},\ldots,R_p] \end{aligned}$$

with $k = p$.

Observe that $\mathbf{cut}\ \{\mathbf{cut}\ \{\mathbf{fwd}\ x\ z\ \ |z|\ \ \mathcal{D}[M_1,\ldots,M_n]\}\ \ |x|\ \ C_2[S_1,\ldots,S_q]\}$ is a predex.

Case: There is no C_1' s.t. $C_1 \equiv_d C_1'[\mathcal{A}]$, for some static context where $s(\mathcal{A}) = x$ and the hole in C_1' is not guarded by a share on x.
Then, the cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex.

Case: There is no C_2' s.t. $C_2 \equiv_d C_2'[\mathcal{A}]$, for some static context where $s(\mathcal{A}) = x$ and the hole in C_2' is not guarded by a share on a x.
Then, the cut $\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ does not yield a predex.

Case: $C_1 \equiv_d C_1'[\mathcal{A}]$, $C_2 \equiv_d C_2'[\mathcal{B}]$, $s(\mathcal{A}) = s(\mathcal{B}) = x$, C_1', C_2' are static contexts in which the holes are not guarded by share operations on x.
Then

$$\begin{aligned} P &= \mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \\ &\equiv_d \mathbf{cut}\ \{C_1'[\mathcal{A},R_1,\ldots,R_p]\ \ |x|\ \ C_2'[\mathcal{B},S_1,\ldots,S_q]\} \\ &\equiv_d C_1'[\mathbf{cut}\ \{\mathcal{A}\ \ |x|\ \ C_2'[\mathcal{B},S_1,\ldots,S_q]\},R_1,\ldots,R_p] \\ &\equiv_d C_1'[C_2'[\mathbf{cut}\ \{\mathcal{A}\ \ |x|\ \ \mathcal{B}\},S_1,\ldots S_q],R_1,\ldots,R_p] \end{aligned}$$

Case: [Tcut!]

Let $P = \mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}$.

By i.h., $P_2 \equiv_d C_2[S_1,\ldots,S_q]$, where $C_2 \nrightarrow_d$.

Observe that x occurs guarded in S_i, for all $1 \leq i \leq q$.

We have two cases to consider, depending on whether (i) x does not occur unguarded in C_2 or (ii) x occurs unguarded in C_2.

Suppose (i). Then

$$\begin{aligned} P &= \mathbf{cut!}\ \{y.P_1\ |x|\ P_2\} \\ &\equiv_d \mathbf{cut!}\ \{y.P_1\ |x|\ C_2[S_1,\ldots,S_q]\} \end{aligned}$$

Let $C = \mathbf{cut!}\ \{y.P_1\ |x|\ C_2\}$.

Suppose (ii). Then

$$\begin{aligned}
P &= \mathbf{cut!}\ \{y.P_1\ |x|\ P_2\} \\
&\equiv_d \mathbf{cut!}\ \{y.P_1\ |x|\ C_2[S_1, \ldots, S_q]\} \\
&= \mathbf{cut!}\ \{y.P_1\ |x|\ C_2'[\mathbf{call}\ x(y_1); Q_1, \ldots, \mathbf{call}\ x(y_n); Q_n, M_1, \ldots, M_m, S_1, \ldots, S_q]\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{where } x \notin \mathrm{n}(C_2')) \\
&\equiv_d C_2'[Q_1', \ldots, Q_n', M_1', \ldots, M_m', S_1', \ldots, S_q']
\end{aligned}$$

where $Q_i' = \mathbf{cut!}\ \{y.P_1\ |x|\ Q_i\}$, $M_i' = \mathbf{cut!}\ \{y.P_1\ |x|\ M_i\}$ and $S_i' = \mathbf{cut!}\ \{y.P_1\ |x|\ S_i\}$.

Let $C = \lambda x_1 \ldots x_n y_1 \ldots y_q.\, C_2'[x_1, \ldots, x_n, M_1', \ldots, M_m', \mathbf{cut!}\ \{y.P_1\ |x|\ y_1\}, \ldots, \mathbf{cut!}\ \{y.P_1\ |x|\ y_q\}]$.

□

Lemma 7 (Diamond Property for $\rightarrow -d$, Restricted to Predexes)**.** *Let P be a predex s.t. $P \rightarrow_d Q$ and $P \rightarrow_d R$. Either $Q \equiv_d R$ or there exists S s.t. both $Q \rightarrow_d S$ and $R \rightarrow_d S$.*

Proof. By induction on the structure of the predex P.

Case: $\mathbf{cut}\ \{\mathcal{A}\ |x|\ \mathcal{B}\}$.

There is an unique $\rightarrow_d$-reduction modulo structural congruence $\equiv_d$, hence the conclusion is immediate.

Case: $\mathbf{cut!}\ \{y.P\ |x|\ \mathbf{call}\ x(y); Q\}$.

There is an unique $\rightarrow_d$-reduction modulo structural congruence $\equiv_d$, hence the conclusion is immediate.

Case: $\mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_n]\}$.

Since $C \twoheadrightarrow_d$, the $\rightarrow_d$-reductions of $\mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_n]\}$ are either obtained by $\rightarrow_d$-reducing one of the predexes R_i or by $\rightarrow_d$-reducing the cut on name y with the forwarder $\mathbf{fwd}\ x\ y$. Consequently, we have the following cases to consider

Case: Q and R are obtained by $\rightarrow_d$-reducing distinct predexes R_i, R_j.

Suppose w.l.o.g. that $i < j$. We have

$$\begin{aligned}
Q &\equiv_d \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_i', \ldots, R_j, \ldots, R_n]\} \text{ and} \\
R &\equiv_d \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_i, \ldots, R_j', \ldots, R_n]\}
\end{aligned}$$

where $R_i \rightarrow_d R_i'$ and $R_j \rightarrow_d R_j'$.

Let $S = \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_i', \ldots, R_j', \ldots, R_n]\}$.

Then, both $Q \rightarrow_d S$ and $R \rightarrow_d S$.

Case: Q and R are obtained by $\rightarrow_d$-reducing the same predex R_i.

We have

$$Q \equiv_d \mathbf{cut}\ \{\mathbf{fwd}\ x\ y\ |y|\ C[R_1, \ldots, R_i', \ldots, R_n]\}$$

and

$$Q \equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i'', \ldots, R_n]\}$$

where $R_i \to_d R_i'$ and $R_i \to_d R_i''$.

By applying the i.h. to R_i we conclude that either (1) $R_i \equiv_d R_i''$ or (2) exists S_i s.t. both $R_i' \to_d S_i$ and $R_i'' \to_d S_i$.

Suppose (1). Then $Q \equiv_d R$.

Suppose (2). Let $S = \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, S_i, \ldots, R_n]\}$. Then, both $Q \to_d S$ and $R \to_d S$.

Case: One of Q, R is obtained by $\to_d$-reducing the cut on name y, the other by $\to_d$-reducing one predex R_i.

Suppose w.l.o.g. that Q is obtained by $\to_d$-reducing the cut on name y and that R is obtained by $\to_d$-reducing a predex R_i. Then

$$Q \equiv_d (\{x/y\}C)[\{x/y\}R_1, \ldots, \{x/y\}R_i, \ldots, \{x/y\}R_n] \text{ and}$$
$$R \equiv_d \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ C[R_1, \ldots, R_i', \ldots, R_n]\}$$

where $R_i \to_d R_i'$.

We have $\{x/y\}R_i \ \to_d \{x/y\}R_i'$.

Let $S = (\{x/y\}C)[\{x/y\}R_1, \ldots, \{x/y\}R_i', \ldots, \{x/y\}R_n]$. Both $Q \to_d S$ and $R \to_d S$.

Case: $\textbf{cut}\ \{C[R_1] \ldots [R_m]\ \ |x|\ \ \textbf{cut}\ \{\textbf{fwd}\ x\ y\ \ |y|\ \ D[R_1] \ldots [R_n]\}\}$.
Similar to previous case.

□

Factorisation of $\twoheadrightarrow$ through $\to_d$

We prove Lemma 9.

Lemma 9(1). *$\sum \mathcal{S}(P) \equiv P$, for all P.*

Proof. By structural induction on P and case analysis on its principal form.

Case: Let $P = \mathcal{X}$, where $\mathcal{X}$ is either the inaction process, a forwarder, an action or a process variable.
We have

$$\sum \mathcal{S}(\mathcal{X}) = \sum \{\mathcal{X}\} \qquad \text{(Def. 23, [S0])}$$
$$= \mathcal{X}$$

Case: $P = \textbf{par}\ \{P_1 \parallel P_2\}$.

Suppose $\mathcal{S}(P_1) = \{P_1^i\}_{i\in\mathcal{I}}$ and $\mathcal{S}(P_2) = \{P_2^j\}_{j\in\mathcal{J}}$.

Then

$$\begin{aligned}
\sum \mathcal{S}(\textbf{par}\ \{P_1 \parallel P_2\}) &= \sum_{i\in\mathcal{I}, j\in\mathcal{J}} (\textbf{par}\ \{P_1^i \parallel P_2^j\}) && \text{(Def. 23, [}\mathcal{S}\text{M])}\\
&\equiv \textbf{par}\ \{(\sum_{i\in\mathcal{I}} P_1^i) \parallel (\sum_{j\in\mathcal{J}} P_2^j)\} && (\equiv \text{rules of } +)\\
&\equiv \textbf{par}\ \{P_1 \parallel P_2\} && \text{(i.h. applied to } P_1 \text{ and } P_2)
\end{aligned}$$

Cases: $P = \textbf{cut}\ \{P_1\ \ |x|\ \ P_2\}$ and $P = \textbf{cut!}\ \{y.P_1\ |x|\ P_2\}$. Similar to case $P = \textbf{par}\ \{P_1 \parallel P_2\}$ above.

Case: $P = P_1 + P_2$.

Suppose $\mathcal{S}(P_1) = \{P_1^i\}_{i\in\mathcal{I}}$ and $\mathcal{S}(P_2) = \{P_2^j\}_{j\in\mathcal{J}}$.

Then

$$\begin{aligned}
\sum \mathcal{S}(P_1 + P_2) &= \sum(\mathcal{S}(P_1) \cup \mathcal{S}(P_2)) && \text{(Def. 23, [}\mathcal{S}\text{Sm])}\\
&\equiv (\sum_{i\in\mathcal{I}} P_1^i) + (\sum_{j\in\mathcal{J}} P_2^j) && (\equiv \text{rules of } +)\\
&\equiv P_1 + P_2 && \text{(i.h. applied to } P_1 \text{ and } P_2)
\end{aligned}$$

Case: $P = \textbf{share}\ x\ \{P_1 \parallel P_2\}$.

We prove first the following auxiliary hypothesis

(H) $\sum \mathcal{I}_x(P, Q) \equiv \textbf{share}\ x\ \{P \parallel Q\}$, for all P, Q.

The definition of the interleaving $\mathcal{I}_x(P, Q)$ map (Def. 23) is split into two cases, whether (i) P and Q both offer a take action with subject x or (ii) not.

If (ii), $\mathcal{I}_x(P, Q) = \{\textbf{share}\ x\ \{P \parallel Q\}\}$, hence (H) follows immediately.

Suppose (i). Then, $P \equiv_d \mathcal{C}[\textbf{take}\ x(y); P']$ and $Q \equiv_d \mathcal{D}[\textbf{take}\ x(z); Q']$, for some static contexts $\mathcal{C}, \mathcal{D}$, where the hole is not guarded by a sum or share on x.

Then

$$\begin{aligned}
&\sum \mathcal{I}_x(\mathcal{C}[\textbf{take}\ x(y); P'], \mathcal{D}[\textbf{take}\ x(z); Q'])\\
&= \mathcal{C} \circ \mathcal{D}[\textbf{take}\ x(y); \textbf{share}\ x\ \{P' \parallel Q\}] + \mathcal{C} \circ \mathcal{D}[\textbf{take}\ x(z); \textbf{share}\ x\ \{P \parallel Q'\}]\\
&\equiv \mathcal{C} \circ \mathcal{D}[\textbf{take}\ x(y); \textbf{share}\ x\ \{P' \parallel Q\} + \textbf{take}\ x(z); \textbf{share}\ x\ \{P \parallel Q'\}]\\
&\qquad\qquad (\equiv \text{distributive laws over sum})\\
&\equiv \mathcal{C} \circ \mathcal{D}[\textbf{share}\ x\ \{\textbf{take}\ x(y); P' \parallel \textbf{take}\ x(z); Q'\}] \qquad (\equiv \text{law [TSh]})\\
&\equiv \textbf{share}\ x\ \{\mathcal{C}[\textbf{take}\ x(y); P'] \parallel \mathcal{D}[\textbf{take}\ x(z); Q']\}\\
&\qquad\qquad (\equiv \text{laws [CSh], [ShC!] ,[ShM], [ShSh]})\\
&= \textbf{share}\ x\ \{P \parallel Q\}
\end{aligned}$$

We continue with the proof of our main Lemma. Suppose $\mathcal{S}(P_1) = \sum_{i\in\mathcal{I}} P_1^i$ and $\mathcal{S}(P_2) = \sum_{j\in\mathcal{J}} P_2^j$.

Then

$$\begin{aligned}
\sum \mathcal{S}(\textbf{share } x\ \{P_1 \mid\mid P_2\}) &= \sum(\bigcup_{i\in\mathcal{I},j\in\mathcal{J}} \mathcal{I}_x(P_1^i, P_2^j)) && \text{(Def. 23, [}\mathcal{S}\text{Sh])}\\
&\equiv \sum_{i\in\mathcal{I},j\in\mathcal{J}} (\sum \mathcal{I}_x(P_1^i, P_2^j))\\
&\equiv \sum_{i\in\mathcal{I},j\in\mathcal{J}} \textbf{share } x\ \{P_1^i \mid\mid P_2^j\} && \text{(hypothesis (H))}\\
&\equiv \textbf{share } x\ \{(\sum_{i\in\mathcal{I}} P_1^i) \mid\mid (\sum_{j\in\mathcal{J}} P_2^j)\} && (\equiv \text{ laws of sum})\\
&\equiv \textbf{share } x\ \{P_1 \mid\mid P_2\} && \text{(i.h. applied to } P_1 \text{ and } P_2)
\end{aligned}$$

□

We introduce the following auxiliary notation which will be useful in the proof of the following lemma.

Definition 37 ($\mathcal{S} \sim \mathcal{T}$). *Given two mutisets of processes $\mathcal{S}, \mathcal{T}$ we write $\mathcal{S} \sim \mathcal{T}$ iff*

(i) *For all $P \in \mathcal{S}$, there exists $Q \in \mathcal{T}$ s.t. $P \equiv_d Q$.*

(ii) *For all $P \in \mathcal{T}$, there exists $Q \in \mathcal{S}$ s.t. $P \equiv_d Q$.*

It is immediate to check that $\sim$ is an equivalence relation. Then, we can succinctly write the statement of Lemma 9(2) in the following way

Lemma 9(2). *If $P \equiv Q$, then $\mathcal{S}(P) \sim \mathcal{S}(Q)$.*

Proof. By induction on the structure of a derivation tree for $P \equiv Q$ and by case analysis on the root rule. Some cases of $\equiv$ immediately follow because they are also contained in $\equiv_d$ and therefore we can apply the conversion pointwise to the multiset $\mathcal{S}(P)$ (and symmetrically to $\mathcal{S}(Q)$), e.g. $\equiv$ law [M] that expresses the commutativity of the mix construct. Other cases follow because the two multisets $\mathcal{S}(P)$ and $\mathcal{S}(Q)$ are the same, e.g. $\equiv$ law [Sm]. We perform case analysis on the root rule of the derivation tree for $\equiv$.

Case: [refl] $P \equiv P$.
Follows because $\sim$ is reflexive.

Case: [symm] $P \equiv Q \supset Q \equiv P$.
Follows by i.h., applied to $P \equiv Q$, and symmetry of $\sim$.

Case: [trans] $P \equiv Q$ and $Q \equiv R \supset P \equiv R$.
Follows by i.h., applied to both $P \equiv Q$ and $Q \equiv R$, and transitivity of $\sim$.

Case: [cong] $P \equiv Q \supset C[P] \equiv C[Q]$.
Follows by i.h., applied to $P \equiv Q$, followed by induction on C.

Case: [M] $\mathbf{par}\ \{P \parallel Q\} \equiv \mathbf{par}\ \{Q \parallel P\}$.
We prove Def. 37(i). Proof of Def. 37(ii) is similar.

Suppose $M \in \mathcal{S}(\mathbf{par}\ \{P \parallel Q\})$.

Then, $M = \mathbf{par}\ \{P' \parallel Q'\}$ where $P' \in \mathcal{S}(P)$ and $Q' \in \mathcal{S}(Q)$.

By applying $\equiv_d$ rule [M] we obtain $\mathbf{par}\ \{P' \parallel Q'\} \equiv_d \mathbf{par}\ \{Q' \parallel P'\}$.

Furthermore, $\mathbf{par}\ \{Q' \parallel P'\} \in \mathcal{S}(\mathbf{par}\ \{Q' \parallel P'\})$.

Case: [CM] $\mathbf{cut}\ \{P\ \ |x : A|\ \ (\mathbf{par}\ \{Q \parallel R\})\} \equiv \mathbf{par}\ \{(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}) \parallel R\}$.
We prove Def. 37(i). Proof of Def. 37(ii) is similar.

Suppose $M \in \mathcal{S}(\mathbf{cut}\ \{P\ \ |x : A|\ \ (\mathbf{par}\ \{Q \parallel R\})\})$.

Then, $M = \mathbf{cut}\ \{P'\ \ |x : A|\ \ (\mathbf{par}\ \{Q' \parallel R'\})\}$, where $P' \in \mathcal{S}(P)$, $Q' \in \mathcal{S}(Q')$ and $R' \in \mathcal{S}(R)$.

By applying $\equiv_d$ rule [CM] we obtain $\mathbf{cut}\ \{P'\ |x : A|\ (\mathbf{par}\ \{Q' \parallel R'\})\} \equiv_d \mathbf{par}\ \{(\mathbf{cut}\ \{P'\ |x : A|\ \ Q'\}) \parallel R'\}$.

Furthermore, $\mathbf{par}\ \{(\mathbf{cut}\ \{P'\ \ |x : A|\ \ Q'\}) \parallel R'\} \in \mathcal{S}(\mathbf{par}\ \{(\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}) \parallel R\})$.

Cases: [C], [fwd], [0M], [MM], [CC], [CC!], [C!M], [C!C!], [D-C!M], [D-C!C], [D-C!C!] follow pointwise application of corresponding $\equiv_d$ rule. Similar to case [CM] above.

Case: [0Sm] $\mathbf{0} + \mathbf{0} \equiv \mathbf{0}$.
We have $\mathcal{S}(\mathbf{0} + \mathbf{0}) = \{\mathbf{0}, \mathbf{0}\}$ and $\mathcal{S}(\mathbf{0}) = \{\mathbf{0}\}$. Follows immediately.

Case: [Sm] $P + Q \equiv Q + P$.
Follows by commutativity of multiset union

$$\mathcal{S}(P + Q) = \mathcal{S}(P) \cup \mathcal{S}(Q) = \mathcal{S}(Q) \cup \mathcal{S}(P) = \mathcal{S}(Q + R)$$

Case: [SmSm] $P + (Q + R) \equiv (P + Q) + R$.
Follows by associativity of multiset union

$$\mathcal{S}(P + (Q + R)) = \mathcal{S}(P) \cup \mathcal{S}(Q) \cup \mathcal{S}(R) = \mathcal{S}((P + Q) + R).$$

Case: [Sh] $\mathbf{share}\ x\ \{P \parallel Q\} \equiv \mathbf{share}\ x\ \{Q \parallel P\}$.
Follows because $\mathcal{I}_x(P', Q') \sim \mathcal{I}_x(Q', P')$, for all P', Q'.

Case: [CSh] $\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{share}\ y\ \{Q \parallel R\}\} \equiv \mathbf{share}\ y\ \{\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\} \parallel R\}$.

Follows because

$$\{\mathbf{cut}\ \{P'\ \ |x|\ \ M\} \mid M \in \mathcal{I}_y(Q', R')\} \sim \mathcal{I}_y(\mathbf{cut}\ \{P'\ \ |x|\ \ Q'\}, R')$$

for all P', Q', R'.

Cases: [ShC!], [ShM], [ShSh] are similar to case [CSh] above.

Case: [MSm] $\mathbf{par}\ \{P \parallel (Q + R)\} \equiv (\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\})$.
Suppose $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$, $\mathcal{S}(Q) = \{Q_j\}_{j\in\mathcal{J}}$ and $\mathcal{S}(R) = \{R_k\}_{k\in\mathcal{K}}$.

Then

$$\mathcal{S}(\mathbf{par}\ \{P \parallel (Q + R)\}) =$$
$$\{\mathbf{par}\ \{P_i \parallel Q_j\}\}_{i\in\mathcal{I},j\in\mathcal{J}} \cup \{\mathbf{par}\ \{P_i \parallel R_k\}\}_{i\in\mathcal{I},k\in\mathcal{K}} = \mathcal{S}((\mathbf{par}\ \{P \parallel Q\}) + (\mathbf{par}\ \{P \parallel R\}))$$

Cases: [CSm], [C!Sm], [ShSm] are similar to case [MSm] above.

Case: [TSh]

$$\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel \mathbf{take}\ x(y_2); P_2\}$$
$$\equiv \mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \parallel \mathbf{take}\ x(y_2); P_2\} + \mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel P_2\}$$

Follows immediately since

$$\mathcal{S}(\mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel \mathbf{take}\ x(y_2); P_2\})$$
$$= \mathcal{S}(\mathbf{take}\ x(y_1); \mathbf{share}\ x\ \{P_1 \parallel \mathbf{take}\ x(y_2); P_2\} + \mathbf{take}\ x(y_2); \mathbf{share}\ x\ \{\mathbf{take}\ x(y_1); P_1 \parallel P_2\})$$

Case: [RSh] $\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel P\} \equiv P$.
We prove Def. 37(i). Proof of Def. 37(ii) is similar.

Suppose $M \in \mathcal{S}(\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel P\})$.

Then, $M = \mathbf{share}\ x\ \{\mathbf{release}\ x \parallel P'\}$ where $P' \in \mathcal{S}(P)$.

By applying $\equiv_d$ rule [FSh] we obtain $\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel P'\} \equiv_d P'$.

Case: [PSh] $\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P); Q \parallel R\} \equiv \mathbf{put}\ x(y.P); \mathbf{share}\ x\ \{Q \parallel R\}$.
Similar to case [RSh] above.

□

Lemma 9(3). *Let $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$. If $P \twoheadrightarrow Q$, then exists $\{Q_{ij}\}_{i\in\mathcal{I},j\in\mathcal{J}_i}$ s.t. $Q \equiv \sum_{i\in\mathcal{I},j\in\mathcal{J}_i} Q_{ij}$ and for all $i \in \mathcal{I}$, $j \in \mathcal{J}_i$, either $P_i \equiv_d Q_{ij}$ or $P_i \rightarrow_d Q_{ij}$.*

Proof. By induction on the structure of a derivation tree for $P \twoheadrightarrow Q$. The principal cut reductions are handled straightforwardly since they do not involve sums, hence in those cases $\mathcal{S}(P)$ is simply the singleton $\{P\}$. Furthermore, all the principal cut reductions are also valid rules of $\rightarrow_d$. Case [cong] is handled by applying pointwise the congruence rule. Case [+par] is straightforward by invoking the inductive hypothesis on both summands P_1 and P_2 of $P = P_1 + P_2$. Case [≡] follows by Lemma 9(2). We perform case analysis on the root rule of $\twoheadrightarrow$.

Case: [**1**⊥] $\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x|\ \ \mathbf{wait}\ x; P'\} \twoheadrightarrow P'$.

We have $\mathcal{S}(\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x|\ \ \mathbf{wait}\ x; P'\}) = \{\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x|\ \ \mathbf{wait}\ x; P'\}\}$.

By $\rightarrow_d$ rule [**1**⊥] we obtain$\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x|\ \ \mathbf{wait}\ x; P'\} \rightarrow_d P'$.

Cases: [fwd], [⊗⅋], [&⊕$_l$], [&⊕$_r$], [!?], [call], [∃∀],[$\mu\nu$], [corec], [∧∨u], [$\mathbf{S}_f\ \mathbf{U}_f$ f], [$\mathbf{S}_f\ \mathbf{U}_f$ t],[$\mathbf{S}_e\ \mathbf{U}_e$] are handled similarly to case [**1**⊥] above.

Case: [cong].

By convenience we consider a distinct but equivalent axiomatisation of $\twoheadrightarrow$ in which rule [cong] is replaced by an explicit congruence law for each static constructor. We illustrate for the mix, left argument:

$$\frac{P \twoheadrightarrow P'}{\mathbf{par}\ \{P \ || \ Q\} \twoheadrightarrow \mathbf{par}\ \{P' \ || \ Q\}}$$

Suppose $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$ and $\mathcal{S}(Q) = \{Q_j\}_{j\in\mathcal{J}}$.

I.h. applied to $P \twoheadrightarrow P'$ yields the existence of $\{P'_{ik}\}_{i\in\mathcal{I},k\in\mathcal{K}_i}$ s.t. $P' \equiv \sum_{i\in\mathcal{I},k\in\mathcal{K}_i} P'_{ik}$ and for all $i \in \mathcal{I}$, $k \in \mathcal{K}_i$, $P_i\ (\rightarrow_d \cup \equiv_d)\ P'_{ik}$.

Consider the set $\{\mathbf{par}\ \{P'_{ik} \ || \ Q_j\}\}_{i\in\mathcal{I},k\in\mathcal{K}_i,j\in\mathcal{J}}$.

Notice that

$$\sum_{i\in\mathcal{I},k\in\mathcal{K}_i,j\in\mathcal{J}} \mathbf{par}\ \{P_{ik} \ || \ Q_j\} \equiv \mathbf{par}\ \{(\sum_{i\in\mathcal{I},k\in\mathcal{K}_i} P_{ik}) \ || \ (\sum_{j\in\mathcal{J}} Q_j)\}$$
$$\equiv \mathbf{par}\ \{P' \ || \ Q\}$$

Furthermore, by applying either $\rightarrow_d$ rule [cong] or $\equiv_d$ rule [cong] pointwise we conclude that for all $i \in \mathcal{I}$, $k \in \mathcal{K}_i$, $j \in \mathcal{J}$, $\mathbf{par}\ \{P_i \ || \ Q_j\}\ \ (\rightarrow_d \cup \equiv_d)\ \mathbf{par}\ \{P'_{ik} \ || \ Q_j\}$.

Case [+par] $P \twoheadrightarrow P'$, $Q \twoheadrightarrow Q'$ $\supset$ $P + Q \twoheadrightarrow P' + Q'$.

Suppose $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$ and $\mathcal{S}(Q) = \{Q_j\}_{j\in\mathcal{J}}$.

By applying i.h. on $P \twoheadrightarrow P'$ we conclude that exists $\{P'_{ik}\}_{i\in\mathcal{I},k\in\mathcal{K}_i}$ s.t. $P' \equiv \sum_{i\in\mathcal{I},k\in\mathcal{K}_i} P'_{ik}$ and $P_i\ (\equiv_d \cup \rightarrow_d)\ P'_{ik}$, for all $i \in \mathcal{I}$, $k \in \mathcal{K}_i$.

By applying i.h. on $Q \twoheadrightarrow Q'$ we conclude that exists $\{Q'_{jl}\}_{j\in\mathcal{J},l\in\mathcal{L}_j}$ s.t. $Q' \equiv \sum_{j\in\mathcal{J},l\in\mathcal{L}_j} Q'_{jl}$ and $Q_j\ (\equiv_d \cup \rightarrow_d) Q'_{jl}$, for all $j \in \mathcal{J}$, $l \in \mathcal{L}_j$.

Furthermore

$$(\sum_{i\in\mathcal{I},k\in\mathcal{K}_i} P'_{ik}) + (\sum_{j\in\mathcal{J},l\in\mathcal{L}_j} Q'_{jl}) \equiv P' + Q'$$

Case $[\equiv]$ $P \equiv P' \twoheadrightarrow Q' \equiv Q \supset P \twoheadrightarrow Q$.

Suppose $\mathcal{S}(P) = \{P_i\}_{i\in\mathcal{I}}$ and $\mathcal{S}(P') = \{P'_{i'}\}_{i'\in\mathcal{I}'}$.

By applying i.h. on $P' \twoheadrightarrow Q'$ we conclude that exists $\{Q'_{i'j'}\}_{i'\in\mathcal{I}',j'\in\mathcal{J}'_{i'}}$ s.t. $Q' \equiv \sum_{i'\in\mathcal{I}',j'\in\mathcal{J}'_{i'}} Q'_{i'j'}$ and $P'_i\ (\rightarrow_d \cup \equiv_d)\ Q'_{i'j'}$, for all $i' \in \mathcal{I}'$, $j' \in \mathcal{J}'_{i'}$.

Define $\mathcal{K}_i = \{i' \in \mathcal{I}' \mid P_i \equiv_d P'_{i'}\}$ and $\mathcal{K}_{i'}^{-1} = \{i \in \mathcal{I} \mid i' \in \mathcal{K}_i\}$.

By definition, $K_{i'}^{-1} = \{i \in \mathcal{I} \mid P_i \equiv_d P'_{i'}\}$.

Since $P \equiv P'$, Lemma 9(2) implies that both $\mathcal{K}_i$ and $\mathcal{K}_i^{-1}$ are nonempty.

Define the multiset $\{R_{ii'j'}\}_{i\in\mathcal{I},i'\in\mathcal{K}_i,j'\in\mathcal{J}'_{i'}}$ by $R_{ii'j'} \triangleq Q'_{i'j'}$.

Notice that for all $i \in \mathcal{I}$, $i' \in \mathcal{K}_i$, $j' \in \mathcal{J}'_{i'}$ we have $P_i \equiv_d P'_i$ and $P'_i\ (\rightarrow_d \cup \equiv_d)\ Q'_{i'j'} = R_{ii'j'}$, hence $P_i\ (\rightarrow_d \cup \equiv_d)\ R_{ii'j'}$.

Furthermore

$$\begin{aligned}
\sum_{i\in\mathcal{I},i'\in\mathcal{K}_i,j'\in\mathcal{J}'_{i'}} R_{ii'j'} &= \sum_{i'\in\mathcal{I}',i\in\mathcal{K}_{i'}^{-1},j'\in\mathcal{J}'_{i'}} R_{ii'j'} \\
&= \sum_{i'\in\mathcal{I}',j'\in\mathcal{J}'_{i'}} \Big(\sum_{i\in\mathcal{K}_{i'}^{-1}} R_{ii'j'}\Big) \\
&\equiv \sum_{i'\in\mathcal{I}',j'\in\mathcal{J}'_{i'}} Q'_{i'j'} && \text{(idempotency of sum)} \\
&\equiv Q' \equiv Q
\end{aligned}$$

□

D

Cut Normalisation

We prove the Share Expansion Lemma 14 and the Cut Normalisation Lemma 15. But before, some useful notation: we annotate (by subscripting) the typing rules [TX] that type an action with the subject x of the introduced action $[\mathrm{TX}]_x$. So, for example, for the action release

$$\frac{}{\textbf{release}\ x \vdash x : \mathbf{U}_f\ A; \Gamma}\ [\mathrm{Trelease}]_x$$

Lemma 3 (Share Expansion). *Let P and Q be processes s.t. $Q \vdash \Delta', x : \boldsymbol{U}_f\ A; \Gamma$ and either $P \vdash \Delta, \boldsymbol{U}_e\ A; \Gamma$ or $P \vdash \Delta, x : \boldsymbol{U}_f\ A; \Gamma$. Then, one of the following hypothesis hold*

(i) *There is a process R s.t.* **share** $x\ \{P \mid\mid Q\} \approx R$ *and* $\#(\textbf{share}\ x\ \{P \mid\mid Q\}) > \#R$.

(ii) *There are processes R, S s.t* **share** $x\ \{P \mid\mid Q\} \approx R + S$ *and* $\#(\textbf{share}\ x\ \{P \mid\mid Q\}) > \#R, \#S$.

(iii) *There is a process R s.t.* **share** $x\ \{P \mid\mid Q\} \approx \textbf{share}\ x\ \{\textbf{fwd}\ x\ y \mid\mid R\}$.

Proof. The proof is by induction on the sum of depths of the type derivation trees of P and Q. We perform case analysis on the root rule of each tree.

Case Exclusions. Typing rules [T**0**], [T**1**], [T!], [Tdiscard], [Tempty] and $[\mathrm{Trelease}]_y$, $[\mathrm{Ttake}]_y$, $[\mathrm{Tput}]_y$ where $y \neq x$, as well as $[\mathrm{T}\bot]_x$, $[\mathrm{T}\otimes]_x$, $[\mathrm{T}\parr]_x$, $[\mathrm{T}\oplus_l]_x$, $[\mathrm{T}\oplus_r]_x$, $[\mathrm{T}\&]_x$, $[\mathrm{T?}]_x$, $[\mathrm{Tcall}]_x$, $[\mathrm{Tcell}]_x$ are all excluded because they type a process with a linear typing context that cannot possibly contain $x : \mathbf{U}_f\ A$ or $x : \mathbf{U}_e\ A$.

The remaining cases are organised in the following groups

Group A: Both derivations have as root rule $[\mathrm{Ttake}]_x$.

Group B: One of the root rules is $[\mathrm{Tput}]_x$.

Group C: One of the root rules is $[\mathrm{Trelease}]_x$.

Group D: One of the root rules is either $[\mathrm{T}\bot]_y$, $[\mathrm{T}\otimes]_y$, $[\mathrm{T}\parr]_y$, $[\mathrm{T}\oplus_l]_y$, $[\mathrm{T}\oplus_r]_y$, $[\mathrm{T}\&]_y$, $[\mathrm{T?}]_y$, $[\mathrm{Tcall}]_y$, $[\mathrm{Taffine}]_y$, $[\mathrm{Tuse}]_y$, $[\mathrm{Tcell}]_y$, $[\mathrm{Ttake}]_y$ or $[\mathrm{Tput}]_y$, where $y \neq x$.

Group E: One of the root rules is either [Tmix], [Tcut], [Tcut!], [Tsh], [TshL] or [TshR].

Group F: One of the root rules is [Tsum].

Group G: One of the root rules is [Tfwd].

We consider w.l.o.g. (since $\mathbf{share}\ x\ \{P \parallel Q\} \approx \mathbf{share}\ x\ \{Q \parallel P\}$) that Groups C-G apply to P.

Group A: Suppose $P = \mathbf{take}\ x(y); P'$ and $Q = \mathbf{take}\ x(y); Q'$.

Applying $\equiv$ rule [TSh] yields

$$\begin{aligned}&\mathbf{share}\ x\ \{\mathbf{take}\ x(y); P' \parallel \mathbf{take}\ x(z); Q'\}\\ &\quad\equiv \mathbf{take}\ x(y); \mathbf{share}\ x\ \{P' \parallel \mathbf{take}\ x(z); Q'\} + \mathbf{take}\ x(z); \mathbf{share}\ x\ \{\mathbf{take}\ x(y); P' \parallel Q'\}\end{aligned}$$

We have

$$\begin{aligned}\#\mathbf{share}\ x\ \{\mathbf{take}\ x(y); P' \parallel Q\} &= (1 + \#P') \times \#Q\\ &= \#Q + \#P' \times \#Q\\ &> 1 + \#P' \times \#Q \qquad \text{(Lemma 13 implies } \#Q > 1)\\ &= \#\mathbf{take}\ x(y); (\mathbf{share}\ x\ \{P' \parallel Q\})\end{aligned}$$

Similarly, one may derive

$$\#\mathbf{share}\ x\ \{P \parallel \mathbf{take}\ x(z); Q'\} < \#\mathbf{take}\ x(z); Q'(\mathbf{share}\ x\ \{P \parallel Q'\})$$

Therefore, (ii) holds.

Group B: Only P can type with an empty usage, hence $P = \mathbf{put}\ x(y.P_1); P_2$.

Applying $\equiv$ rule [PSh] yields

$$\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P_1); P_2 \parallel Q\} \approx \mathbf{put}\ x(y.P_1); \mathbf{share}\ x\ \{P_1 \parallel Q\}$$

And

$$\begin{aligned}\#\mathbf{share}\ x\ \{\mathbf{put}\ x(y.P_1); P_2 \parallel Q\} &= (1 + \#P_1 \times \#P_2) \times \#Q\\ &= \#Q + \#P_1 \times \#P_2 \times \#Q\\ &> 1 + \#P_1 \times \#P_2 \times \#Q \quad \text{(Lemma 13 implies } \#Q > 1)\\ &= \#\mathbf{put}\ x(y.P_1); \mathbf{share}\ x\ \{P_1 \parallel Q\}\end{aligned}$$

Therefore, (i) holds.

Group C: Applying $\equiv$ rule [RSh] yields

$$\mathbf{share}\ x\ \{\mathbf{release}\ x \parallel Q\} \approx Q$$

And

$$\begin{aligned}\#\textbf{share}\ x\ \{\textbf{release}\ x \parallel Q\} &= \#\textbf{release}\ x\ \times \#Q \\ &= 2 \times \#Q \\ &> \#Q \qquad \text{(Lemma 13 implies } \#Q > 0)\end{aligned}$$

Therefore, (i) holds.

Group D: We illustrate with cases [T⊥] and [T&].

Case [T⊥]

Suppose $P = \textbf{wait}\ y; P'$. Applying ≈ rule [Sh⊥] yields

$$\textbf{share}\ x\ \{\textbf{wait}\ y; P' \parallel Q\} \approx \textbf{wait}\ y; \textbf{share}\ x\ \{P' \parallel Q\}$$

. And

$$\begin{aligned}\#(\textbf{share}\ x\ \{\textbf{wait}\ y; P' \parallel Q\}) &= (1 + \#P') \times \#Q \\ &= \#Q + \#P' \times \#Q \\ &> 1 + \#P' \times \#Q \qquad \text{(Lemma 13 implies } \#Q > 1) \\ &= \#(\textbf{wait}\ y; (\textbf{share}\ x\ \{P' \parallel Q\}))\end{aligned}$$

Case [T&]

Suppose $P = \textbf{case}\ y\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}$. Applying ≈ rule [Sh&] yields

$$\textbf{share}\ x\ \{\textbf{case}\ y\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\} \parallel Q\} \approx \textbf{case}\ y\ \{|\mathsf{inl} : \textbf{share}\ x\ \{P_1 \parallel Q\} \mid \mathsf{inr} : \textbf{share}\ x\ \{P_2 \parallel Q\}\}$$

. And

$$\begin{aligned}&\#(\textbf{share}\ x\ \{\textbf{case}\ y\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\} \parallel Q\}) \qquad = (1 + \#P_1 + \#P_2) \times \#Q \\ &= \#Q + \#P_1 \times \#Q + \#P_2 \times \#Q \\ &> 1 + \#P_1 \times \#Q + \#P_2 \times \#Q \qquad \text{(Lemma 13 implies } \#Q > 1) \\ &= \#(\textbf{case}\ y\ \{|\mathsf{inl} : \textbf{share}\ x\ \{P_1 \parallel Q\} \mid \mathsf{inr} : \textbf{share}\ x\ \{P_2 \parallel Q\}\})\end{aligned}$$

Group E: We illustrate with case [Tmix].

Suppose $P = \textbf{par}\ \{P_1 \parallel P_2\}$ for some P_1, P_2.We assume w.l.o.g. that $x \in \mathsf{fn}(P_1)$ instead of $x \in \mathsf{fn}(P_2)$.

Applying ≡ rule [ShM] yields

$$\textbf{share}\ x\ \{(\textbf{par}\ \{P_1 \parallel P_2\}) \parallel Q\} \approx \textbf{par}\ \{\textbf{share}\ x\ \{P_1 \parallel Q\} \parallel P_2\}$$

By induction we conclude that one of the following hypothesis hold

Case (i): There is R s.t. $\mathbf{share}\ x\ \{P_1 \parallel Q\} \approx R$ and $\#(\mathbf{share}\ x\ \{P_1 \parallel Q\}) > \#R$

Then

$$\mathbf{share}\ x\ \{(\mathbf{par}\ \{P_1 \parallel P_2\}) \parallel Q\} \approx \mathbf{par}\ \{R \parallel P_2\}$$

and

$$\begin{aligned}\#(\mathbf{share}\ x\ \{\mathbf{par}\ \{P_1 \parallel P_2\} \parallel Q\}) &= \#(\mathbf{par}\ \{P_1 \parallel P_2\}) \times \#Q \\ &= \#P_1 \times \#P_2 \times \#Q \\ &= \#P_1 \times \#Q \times \#P_2 \\ &= \#(\mathbf{share}\ x\ \{P_1 \parallel Q\}) \times \#P_2 \\ &> \#R \times \#P_2 \quad (\text{since } \#(\mathbf{share}\ x\ \{P_1 \parallel Q\}) > \#R) \\ &= \#(\mathbf{par}\ \{R \parallel P_2\})\end{aligned}$$

So hypothesis (i) holds.

Case (ii): There are R, S s.t. $\mathbf{share}\ x\ \{P_1 \parallel Q\} \approx R + S$ and $\#(\mathbf{share}\ x\ \{P_1 \parallel Q\}) > \#R, \#S$.

Then

$$\begin{aligned}\mathbf{share}\ x\ \{(\mathbf{par}\ \{P_1 \parallel P_2\}) \parallel Q\} &\approx \mathbf{par}\ \{(R + S) \parallel P_2\} \\ &\approx (\mathbf{par}\ \{R \parallel P_2\}) + (\mathbf{par}\ \{S \parallel P_2\})\end{aligned}$$

We can obtain (derivation similar to Case (i) above)

$$\#(\mathbf{share}\ x\ \{(\mathbf{par}\ \{P_1 \parallel P_2\}) \parallel Q\}) > \#(\mathbf{par}\ \{R \parallel P_2\}), \#(\mathbf{par}\ \{S \parallel P_2\})$$

So hypothesis (ii) holds.

Case (iii): There is R s.t. $\mathbf{share}\ x\ \{P_1 \parallel Q\} \approx \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ y \parallel R\}$.

Then

$$\begin{aligned}&\mathbf{share}\ x\ \{(\mathbf{par}\ \{P_1 \parallel P_2\}) \parallel Q\} \approx \\ &\mathbf{par}\ \{\mathbf{share}\ x\ \{\mathbf{fwd}\ x\ y \parallel R\} \parallel P_2\} \approx \mathbf{share}\ x\ \{\mathbf{fwd}\ x\ y \parallel (\mathbf{par}\ \{R \parallel P_2\})\}\end{aligned}$$

So hypothesis (iii) holds.

Group F: We have $P = P_1 + P_2$.

Applying ≡ rule [ShSm] yields

$$\mathbf{share}\ x\ \{P_1 + P_2 \parallel Q\} \approx \mathbf{share}\ x\ \{P_1 \parallel Q\} + \mathbf{share}\ x\ \{P_2 \parallel Q\}$$

We have

$$\begin{aligned}\#(\mathbf{share}\ x\ \{P_1 + P_2 \parallel Q\}) &= \#(P_1 + P_2) \times \#Q \\ &= (\#P_1 + \#P_2) \times \#Q \\ &= \#P_1 \times \#Q + \#P_2 \times \#Q \\ &> \#P_1 \times \#Q \qquad (\#P_2, \#Q > 0) \\ &= \#(\mathbf{share}\ x\ \{P_1 \parallel Q\})\end{aligned}$$

Similarly, one can derive

$$\#(\textbf{share}\ x\ \{P_1 + P_2 \mid\mid Q\}) > \#(\textbf{share}\ x\ \{P_2 \mid\mid Q\})$$

Therefore, hypothesis (ii) holds.

Group G: We have $P = \textbf{fwd}\ x\ y$. Hypothesis (iii) holds.

□

Lemma 32 (Cut Normalisation)**.** *The following two hypothesis hold*

$H_1(A, P, Q)$**:** *Suppose* $P \vdash \Delta', x : A; \Gamma$, $Q \vdash \Delta, x : \overline{A}; \Gamma$ *are normal processes. There exists a normal process* R *s.t.* **cut** $\{P \ \ |x : A|\ \ Q\} \approx R$.

$H_2(A, P, Q)$ *Suppose* $P \vdash y : A; \Gamma$ *and* $Q \vdash \Delta; \Gamma, x : \overline{A}$ *are normal processes. There exists a normal process* R *s.t.* **cut!** $\{y.P \ |x : A|\ Q\} \approx R$.

Proof. The proof is by mutual induction. More precisely, by lexicographical induction: first on A (ordered by $<$, Def. 28), then on the hypothesis being proved - we consider $H_1 < H_2$ - and finally on the #-measure of the cut.

Since $H_1 < H_2$, we can appeal to $H_1(A, P, Q)$ when proving $H_2(A, P, Q)$. To justify the inductive call we use the following notation

$$(A, H_i, P, Q) \rightarrow (B, H_j, R, S)$$

which means that we are proving hypothesis $H_i(A, P, Q)$ and that somewhere during the proof we appeal to hypothesis $H_j(B, R, S)$. The inductive call must respect the lexicographical order.

Proof of $H_1(A, P, Q)$**:** We perform case analysis on the root rules of derivation trees for $P \vdash \Delta', x : A; \Gamma$ and $Q \vdash \Delta, x : \overline{A}; \Gamma$.

Case Exclusions. Typing rules $[\text{Tcall}]_x$ and $[\text{T}\mathbf{0}]$, $[\text{T}\mathbf{1}]_y$, $[\text{T!}]_y$,$[\text{Tdiscard}]_y$, $[\text{Tempty}]_y$, and $[\text{Trelease}]_y$, where $y \neq x$, are excluded because they type a process with a linear typing context that cannot possibly contain $x : \overline{A}$ or $x : A$. Typing rule [Tcut!] is excluded since we are assuming that both P and Q are in normal form. The remaining cases are organised in the following groups

Group A: Each root rule is either $[\text{T}\mathbf{1}]_x$, $[\text{T}\bot]_x$, $[\text{T}\otimes]_x$, $[\text{T}\parr]_x$, $[\text{T}\oplus_l]_x$, $[\text{T}\oplus_r]_x$, $[\text{T}\&]_x$, $[\text{T!}]_x$, $[\text{T?}]_x$, $[\text{Taffine}]_x$, $[\text{Tuse}]_x$, $[\text{Tdiscard}]_x$, $[\text{Tcell}]_x$, $[\text{Trelease}]_x$, $[\text{Ttake}]_x$ or $[\text{Tput}]_x$.

Group B: One of the root rules is either $[\text{T}\bot]_y$ $[\text{T}\otimes]_y$ $[\text{T}\parr]_y$, $[\text{T}\oplus_l]_y$, $[\text{T}\oplus_r]_y$ $[\text{T}\&]_y$ $[\text{T?}]_y$, $[\text{Tcall}]_y$,$[\text{Taffine}]_y$, $[\text{Tuse}]_y$, $[\text{Tcell}]_y$, $[\text{Ttake}]_y$ or $[\text{Tput}]_y$, with $y \neq x$.

Group C: One of the root rules is either [Tmix], [Tsum] or $[\text{Tsh}]_y$, $[\text{TshL}]_y$ or $[\text{TshR}]_y$ with $y \neq x$.

Group D : One of the root rules is either $[\text{Tcell}]_x$ or $[\text{Tempty}]_x$, the other is either $[\text{Tsh}]_x$, $[\text{TshL}]_x$ or $[\text{TshR}]_x$.

Group E : One of the root rules is [Tcut].

Group A Since the name x is typed by $\overline{A}$ in P and by A in Q we conclude that the introduced actions must be dual. Therefore, we are able to apply one of the principal cut reductions. We perform case analysis on A. We illustrate with some representative cases.

Case $A = \mathbf{1}$.

We have

$$\frac{}{\mathbf{wait}\ x; \vdash x : \mathbf{1}; \Gamma} \qquad \frac{Q_0 \vdash \Delta; \Gamma}{\mathbf{wait}\ x; Q_0 \vdash \Delta, x : \bot; \Gamma}$$

where $P = \mathbf{close}\ x$, $\Delta = \emptyset$ and $Q = \mathbf{wait}\ x; Q_0$.

Then

$$\mathbf{cut}\ \{\mathbf{close}\ x \ \ |x : \mathbf{1}|\ \ \mathbf{wait}\ x; Q_0\} \to Q_0 \qquad (\to [\mathbf{1}\bot])$$

By hypothesis, Q, and hence Q_0, is normal.

Case $A = A_1 \otimes A_2$.

We have

$$\frac{P_1 \vdash \Delta_1', y : A_1; \Gamma \quad P_2 \vdash \Delta_2', x : A_2; \Gamma}{\mathbf{send}\ x(y.P_1); P_2 \vdash \Delta_1', \Delta_2', x : A_1 \otimes A_2; \Gamma} \qquad \frac{Q' \vdash \Delta, x : \overline{A}_2, y : \overline{A}_1; \Gamma}{\mathbf{recv}\ x(y); Q' \vdash \Delta, x : \overline{A}_1 \parr \overline{A}_2}$$

where $P = \mathbf{send}\ x(y.P_1); P_2$, $\Delta' = \Delta_1', \Delta_2'$ and $Q = \mathbf{recv}\ x(z); Q'$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{send}\ x(y.P_1); P_2 \ \ |x : A_1 \otimes A_2|\ \ \mathbf{recv}\ x(z); Q'\} \\
&\to \mathbf{cut}\ \{P_2 \ \ |x : A_2|\ \ (\mathbf{cut}\ \{P_1 \ \ |y : A_1|\ \ \{y/z\}Q'\})\} && (\to [\otimes\parr]) \\
&\approx \mathbf{cut}\ \{P_2 \ \ |x : A_2|\ \ R_1\}, \text{for some normal } R_1 \\
& && (\text{i.h. } (A_1 \otimes A_2, H_1, -, -) \to (A_1, H_1, -, -)) \\
&\approx R, \text{for some normal } R && (\text{i.h. } (A_1 \otimes A_2, H_1, -, -) \to (A_2, H_1, -, -))
\end{aligned}$$

Case $A = !B$

We have

$$\frac{P' \vdash y : B; \Gamma}{!x(y); P' \vdash x :!B; \Gamma} \qquad \frac{Q' \vdash \Delta; \Gamma, x : \overline{B}}{?x; Q' \vdash \Delta, x :?\overline{B}; \Gamma}$$

where $P = !x(y); P'$, $\Delta' = \emptyset$ and $Q = ?x; Q'$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{!x(y); P' \ \ |x :!B|\ \ ?x; Q'\} \\
&\to \mathbf{cut!}\ \{y.P' \ |x : B|\ Q'\} && (\to [!?]) \\
&\approx R, \text{for some normal } R && (\text{induction } (!B, H_1, -, -) \to (B, H_2, -, -))
\end{aligned}$$

Case $A = \mathbf{S}_f\ B$.

We have

$$\frac{P' \vdash \Delta', y : \wedge B; \Gamma}{\mathbf{cell}\ x(y.P')\ \vdash \Delta', x : \mathbf{S}_f\ B; \Gamma}$$

where $P = \mathbf{cell}\ x(y.P')$.

The $\mathbf{U}_f\ B$ modality can be introduced by two possible rules: $[\text{Trelease}]_x$ or $[\text{Ttake}]_x$.

Case $[\text{Trelease}]_x$.

We have

$$\mathbf{release}\ x \vdash x : \mathbf{U}_f\ \overline{B}; \Gamma$$

where $Q = \mathbf{release}\ x$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{cell}\ x(y.P')\ \ |x : \mathbf{S}_f\ B|\ \ \mathbf{release}\ x\} \\
&\to \mathbf{cut}\ \{P\ \ |y : \wedge B|\ \ \mathbf{discard}\ y\} && (\to [\mathbf{S}_f\ \mathbf{U}_f\ \text{r}]) \\
&\approx R, \text{for some normal } R \quad (\text{induction } (\mathbf{S}_f\ B, -, -, -) \to (\wedge B, -, -, -))
\end{aligned}$$

Case $[\text{Ttake}]_x$.

We have

$$\frac{Q' \vdash \Delta, z : \vee\overline{B}, x : \mathbf{U}_e\ \overline{B}; \Gamma}{\mathbf{take}\ x(z); Q'\ \vdash \Delta, x : \mathbf{U}_f\ \overline{B}; \Gamma}$$

where $Q = \mathbf{take}\ x(z); Q'$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{cell}\ x(y.P')\ \ |x : \mathbf{S}_f\ B|\ \ \mathbf{take}\ x(z); Q'\} \\
&\to \mathbf{cut}\ \{\{z/y\}P'\ \ |z : \wedge B|\ \ (\mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x : \mathbf{S}_e\ B|\ \ Q'\})\} \\
& && (\to [\mathbf{S}_f\ \mathbf{U}_f\ \text{t}]) \\
&\approx \mathbf{cut}\ \{\{z/y\}P'\ \ |z : \wedge B|\ \ R'\}, \text{for some normal } R' \\
& && (\text{i.h. } (\mathbf{S}_f\ B, H_1, P, Q) \to (\mathbf{S}_e\ B, H_1, \mathbf{empty}\ x, Q')) \\
&\approx R, \text{for some normal } R \quad (\text{induction } (\mathbf{S}_f\ B, -, -, -) \to (\wedge B, -, -, -))
\end{aligned}$$

Case $A = \mathbf{S}_e\ B$.

We have

$$\frac{}{\mathbf{empty}\ x \vdash x : \mathbf{S}_e\ B; \Gamma} \quad \frac{Q_1 \vdash \Delta_1, z : \wedge B; \Gamma \quad Q_2 \vdash \Delta, x : \mathbf{S}_f\ \overline{B}; \Gamma}{\mathbf{put}\ x(y.Q_1); Q_2 \vdash \Delta_1, \Delta_2, x : \mathbf{S}_e\ \overline{B}; \Gamma}$$

where $P = \mathbf{empty}\ x$, $\Delta' = \emptyset$,$Q = \mathbf{put}\ x(z.Q_1); Q_2$ and $\Delta = \Delta_1, \Delta_2$.

Then

$$\begin{aligned}
&\mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x : \mathbf{S}_e\ B|\ \ \mathbf{put}\ x(z.Q_1); Q_2\}\\
&\to \mathbf{cut}\ \{\mathbf{cell}\ x(z.Q_1)\ \ |x : \mathbf{S}_f\ B|\ \ Q_2\} \qquad (\to [\mathbf{S}_e\ \mathbf{U}_e\ ]])\\
&\approx R, \text{ for some normal } R\\
&(\text{induction } (\mathbf{S}_e\ B, H_1, \mathbf{empty}\ x, \mathbf{put}\ x(z.Q_1); Q_2) \to (\mathbf{S}_f\ B, H_1, \mathbf{cell}\ x(z.Q_1), Q_2))
\end{aligned}$$

The inductive call is justified by

$$\begin{aligned}
&\#(\mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ \mathbf{wrt}\ x(z.Q_1); Q_2\})\\
&= \#\mathbf{empty}\ x \times \#\mathbf{wrt}\ x(z.Q_1); Q_2\\
&= 2 \times (1 + \#Q_1 \times \#Q_2)\\
&= 2 + 2 \times \#Q_1 \times \#Q_2\\
&> (2 \times \#Q_1) \times \#Q_2\\
&= (\#Q_1 + \#Q_1) \times \#Q_2\\
&> (1 + \#Q_1) \times \#Q_2 \qquad (\text{Lemma 13 implies } \#Q_1 > 1)\\
&= \#(\mathbf{cut}\ \{\mathbf{cell}\ x(z.Q_1)\ \ |x|\ \ Q_2\})
\end{aligned}$$

Group B All of these cases introduce an action with subject $y \neq x$ and are handled by applying one of the $\approx$ conversions of Fig. 8.2, which commute a cut with an action. We obtain a cut which is #-smaller than the initial cut for which the hypothesis is being proved, which allow us to make an inductive call. We will illustrate with case $[\mathrm{T}\bot]_y$.

Case $[\mathrm{T}\bot]_y$.

We have

$$\frac{Q' \vdash \Delta_0, x : \overline{A}; \Gamma}{\mathbf{wait}\ y; Q' \vdash \Delta_0, x : \overline{A}, y : \bot; \Gamma}$$

where $Q = \mathbf{wait}\ y; Q'$ and $\Delta = \Delta_0, y : \bot$.

Then

$$\begin{aligned}
\mathbf{cut}\ \{P\ \ |x : A|\ \ \mathbf{wait}\ y; Q'\} &\approx \mathbf{wait}\ y; (\mathbf{cut}\ \{P\ \ |x : A|\ \ Q'\}) \quad (\text{figure 8.2, } [\bot])\\
&\approx \mathbf{wait}\ y; (R), \text{ for some normal } R\\
&\quad (\text{i.h. } (A, H_1, P, \mathbf{wait}\ y; Q') \to (A, H_1, P, Q'))
\end{aligned}$$

Group C Cases of this group are handled by applying one of the $\equiv$ conversions [CM], [CSh] or [CSm] that commute a cut with the static constructs mix, share and sum. We obtain a #-smaller cut (two, in case of [Tsum]) which allow us to make an inductive call. We illustrate with cases [Tmix] and [Tsum]. Case $[\mathrm{Tshare}]_y$ is handled similarly to [Tmix].

Case [Tmix].

We have

$$\frac{P_1 \vdash \Delta_1', x : A; \Gamma \quad P_2 \vdash \Delta_2'; \Gamma}{\mathbf{par}\ \{P_1 \mid\mid P_2\} \vdash \Delta_1', \Delta_2', x : A; \Gamma}$$

where $P = \mathbf{par}\ \{P_1 \mid\mid P_2\}$ and $\Delta' = \Delta_1', \Delta_2'$. We assume w.l.o.g., since $\mathbf{par}\ \{P_1 \mid\mid P_2\} \equiv \mathbf{par}\ \{P_2 \mid\mid P_1\}$, that $x \in \mathsf{fn}(P_1)$.

Then

$$\begin{aligned}\mathbf{cut}\ \{(\mathbf{par}\ \{P_1 \mid\mid P_2\})\ \ |x : A|\ \ Q\} &\equiv \mathbf{par}\ \{(\mathbf{cut}\ \{P_1\ \ |x : A|\ \ Q\}) \mid\mid P_2\} \\ &\qquad (\equiv \text{[CM]}) \\ &\approx \mathbf{par}\ \{R \mid\mid P_2\}, \text{ for some normal } R \\ &\qquad (\text{i.h } (A, H_1, P, Q) \longrightarrow (A, H_1, P_1, Q))\end{aligned}$$

Case [Tsum]. We have

$$\frac{P_1 \vdash \Delta', x : A; \Gamma \quad P_2 \vdash \Delta', x : A; \Gamma}{P_1 + P_2 \vdash \Delta', x : A; \Gamma}$$

where $P = P_1 + P_2$.

Then

$$\begin{aligned}\mathbf{cut}\ \{(P_1 + P_2)\ \ |x : A|\ \ Q\} &\equiv (\mathbf{cut}\ \{P_1\ \ |x : A|\ \ Q\}) + (\mathbf{cut}\ \{P_2\ \ |x : A|\ \ Q\}) \\ &\qquad (\equiv \text{[CSm]}) \\ &\approx R_1 + (\mathbf{cut}\ \{P_2\ \ |x : A|\ \ Q\}), \text{ for some normal } R_1 \\ &\qquad (\text{i.h. } (A, H_1, P, Q) \longrightarrow (A, H_1, P_1, Q)) \\ &\approx R_1 + R_2, \text{ for some normal } R_1 \\ &\qquad (\text{induction } (A, H_1, P, Q) \longrightarrow (A, H_1, P_2, Q))\end{aligned}$$

Group D Suppose that one of the root rules is $[\text{Tcell}]_x$ and the other $[\text{Tshare}]_x$. The other cases are handled similarly.

We have

$$\frac{P' \vdash \Delta, y : \wedge B; \Gamma}{\mathbf{cell}\ x(y.P') \vdash x : \mathbf{S}_f\ B; \Gamma} \qquad \frac{Q_1 \vdash \Delta_1', x : \mathbf{U}_f\ \overline{B}; \Gamma \quad Q_2 \vdash \Delta_2', x : \mathbf{U}_f\ \overline{B}; \Gamma}{\mathbf{share}\ x\ \{Q_1 \mid\mid Q_2\}\ \vdash \Delta_1', \Delta_2', x : \mathbf{U}_f\ \overline{B}; \Gamma}$$

where $P = \mathbf{cell}\ x(y.P')$, $Q = \mathbf{share}\ x\ \{Q_1 \mid\mid Q_2\}$, $\Delta' = \Delta_1', \Delta_2'$ and $A = \mathbf{S}_f\ B$.

By a applying Lemma 14 to hypothesis $Q_1 \vdash \Delta_1', x : \mathbf{U}_f\ B; \Gamma$ and $Q_2 \vdash \Delta_2', x : \mathbf{U}_f\ B; \Gamma$ we conclude that one of the following cases hold

Case (i): There is Q' s.t. $Q \approx Q'$ and $\#Q > \#Q'$.

Then

$$\begin{aligned}\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\ &\approx \mathbf{cut}\ \{P\ \ |x : A|\ \ Q'\} \\ &\approx R, \text{ for some normal } R \\ &\qquad (\text{i.h. } (A, H_1, P, Q) \longrightarrow (A, H_1, P, Q'))\end{aligned}$$

Case (ii): Exists R_1, R_2 s.t. $Q \approx R_1 + R_2$ and $\#Q > \#R_1, \#R_2$.
Then

$$\begin{aligned}
\textbf{cut}\ \{P\ \ |x : A|\ \ Q\} &\approx \textbf{cut}\ \{P\ \ |x : A|\ \ (R_1 + R_2)\} \\
&\approx (\textbf{cut}\ \{P\ \ |x : A|\ \ R_1\}) + (\textbf{cut}\ \{P\ \ |x : A|\ \ R_2\}) \\
&\approx S_1 + (\textbf{cut}\ \{P\ \ |x : A|\ \ R_2\}), \text{ for some normal } S_2 \\
&\qquad\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P, R_1)) \\
&\approx S_1 + S_2, \text{ for some normal } S_2 \\
&\qquad\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P, R_2))
\end{aligned}$$

Case (iii): Exists Q' s.t. $Q \approx \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \parallel Q'\}$
Then

$$\textbf{cut}\ \{P\ \ |x : A|\ \ Q\} \approx \textbf{cut}\ \{\textbf{cell}\ x(y.P')\ \ |x : A|\ \ \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \parallel Q'\}\}$$

and $\textbf{cut}\ \{\textbf{cell}\ x(y.P')\ \ |x : A|\ \ \textbf{share}\ x\ \{\textbf{fwd}\ x\ z \parallel Q'\}\}$ is an open cell.

Observe that the transformations of Lemma 14 neither introduce newer cuts nor do they interfere with the existing ones. As as consequence, the processes obtained by applying the lemma are still normal.

Group E [Tcut]

The cut must be an open cell. Suppose it is a full open cell, the strategy is similar if it is an empty open cell.

We have

$$P = \textbf{cut}\ \{\textbf{cell}\ y(z.P_1)\ \ |y : B|\ \ \textbf{share}\ y\ \{\textbf{fwd}\ y\ w \parallel P_2\}\}$$

There are two hypothesis to consider, depending on whether (i) $w \neq x$ or (ii) $w = x$.

Case (i) $w \neq x$
Then

$$\begin{aligned}
&\textbf{cut}\ \{P\ \ |x : A|\ \ Q\} \\
&= \textbf{cut}\ \{(\textbf{cut}\ \{\textbf{cell}\ y(z.P_1)\ \ |y : B|\ \ \textbf{share}\ y\ \{\textbf{fwd}\ y\ w \parallel P_2\}\})\ \ |x : A|\ \ Q\} \\
&\approx \textbf{cut}\ \{\textbf{cell}\ y(z.P_1)\ \ |y : B|\ \ \textbf{share}\ y\ \{\textbf{fwd}\ y\ w \parallel \textbf{cut}\ \{P_2\ \ |x : A|\ \ Q\}\}\} \\
&\approx \textbf{cut}\ \{\textbf{cell}\ y(z.P_1)\ \ |y : B|\ \ \textbf{share}\ y\ \{\textbf{fwd}\ y\ w \parallel R\}\}, \text{ for some normal } R \\
&\qquad\qquad (\text{i.h. } (A, H_1, P, Q) \rightarrow (A, H_1, P_2, Q))
\end{aligned}$$

Case (ii) $w = x$

Since y and x occur in a forwarder, we have $B = A$. Then

$$\begin{aligned}
&\mathbf{cut}\ \{P\ \ |x : A|\ \ Q\}\\
&= \mathbf{cut}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ y(z.P_1)\ \ |y : A|\ \ \mathbf{share}\ y\ \{\mathbf{fwd}\ y\ x \parallel P_2\}\})\ \ |x : A|\ \ Q\}\\
&\approx \mathbf{cut}\ \{\mathbf{cell}\ y(z.P_1)\ \ |y : A|\ \ \mathbf{share}\ y\ \{\{y/x\}Q \parallel P_2\}\}\\
&\approx R, \text{ for some normal } R\\
&\qquad \text{(i.h. } (A, H_1, P, Q) \longrightarrow (A, H_1, \mathbf{cell}\ y(z.P_1), \mathbf{share}\ y\ \{\{y/x\}Q \parallel P_2\}))
\end{aligned}$$

The following justifies the inductive call

$$\begin{aligned}
\#(\mathbf{cut}\ \{P\ \ |x|\ \ Q\}) &= \#P \times \#Q\\
&= \#(\mathbf{cut}\ \{\mathbf{cell}\ y(z.P_1)\ \ |y|\ \ \mathbf{share}\ y\ \{\mathbf{fwd}\ y\ x \parallel P_2\}\}) \times \#Q\\
&= \#\mathbf{cell}\ y(z.P_1) \times \#\mathbf{fwd}\ y\ x \times \#P_2 \times \#Q\\
&> \#\mathbf{cell}\ y(z.P_1) \times \#P_2 \times \#Q \qquad \text{(since } \#\mathbf{fwd}\ y\ x = 2)\\
&= \#\mathbf{cell}\ y(z.P_1) \times \#P_2 \times \#(\{y/x\}Q)\\
&= \#(\mathbf{cut}\ \{\mathbf{cell}\ y(z.P_1)\ \ |y|\ \ \mathbf{share}\ y\ \{\{y/x\}Q \parallel P_2\}\})
\end{aligned}$$

Proof of H_2

The proof is by case analysis on the root rule of derivation tree for $Q \vdash \Delta; \Gamma, x : \overline{A}$. Case [Tcut!] is excluded since, by hypothesis, Q is normal. The remaining cases are organised in the following groups

Group A: The root rule is either [T**0**], [T**1**], [Tfwd], [Tdiscard], [Trelease] or [Tempty].

Group B: The root rule is $[\text{Tcall}]_x$.

Group C: The root rule is $[\text{Tcall}]_y$, with $y \neq x$ or [T$\bot$], [T$\otimes$], [T$\parr$], [T$\oplus_l$], [T$\oplus_r$], [T$\&$], [T!], [T?], [Taffine], [Tuse], [Tcell], [Ttake] or [Tput].

Group D: The root rules is either [Tmix], [Tsh], [TshL], [TshR] or [Tsum].

Group E: The root rule is [Tcut].

Group A These cases are handled by applying one of the $\approx$ discarding principles [C!**0**], [C!**1**], [C!fwd] or [C!free], [C!discard], [C!release] or [C!empty]. We illustrate for case [T**0**].

Case [T**0**].

We have

$$\frac{}{\mathbf{0} \vdash \emptyset; \Gamma}$$

Applying $\approx$ [C!**0**] yields $\mathbf{cut!}\ \{y.P\ |x : A|\ \mathbf{0}\} \approx \mathbf{0}$ and, by Def. 25, **0** is normal.

Group B We have

$$\frac{Q' \vdash \Delta, z : \overline{A}; \Gamma, x : \overline{A}}{\textbf{call}\ x(z); Q' \ \vdash \Delta; \Gamma, x : \overline{A}}$$

where $Q = \textbf{call}\ x(z); Q'$.

Then

$$\begin{aligned}
\textbf{cut!}\ \{y.P\ |x : A|\ \textbf{call}\ x(z); Q'\} &\to \textbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ (\textbf{cut!}\ \{y.P\ |x : A|\ Q'\})\} \\
&\qquad (\to [\text{call}]) \\
&\approx \textbf{cut}\ \{\{z/y\}P\ \ |z : A|\ \ R'\}, \text{for some normal } R' \\
&\qquad (\text{i.h. } (A, H_2, P, Q) \to (A, H_2, P, Q')) \\
&\approx R, \text{for some normal } R \\
&\qquad (\text{i.h. } (A, H_2, -, -) \to (A, H_1, -, -))
\end{aligned}$$

Group C The cases of this group are handled by applying one $\approx$ conversions of Fig. 8.3 that commute an unrestricted cut with an action. We illustrate with case [T$\bot$].

Case [T$\bot$].

We have

$$\frac{Q' \vdash \Delta'; \Gamma, x : \overline{A}}{\textbf{wait}\ y; Q' \vdash \Delta', y : \bot; \Gamma, x : \overline{A}}$$

where $Q = \textbf{wait}\ y; Q'$ and $\Delta = \Delta', y : \bot$.

Then

$$\begin{aligned}
\textbf{cut!}\ \{y.P\ |x : A|\ \textbf{wait}\ y; Q'\} &\approx \textbf{wait}\ y; (\textbf{cut!}\ \{y.P\ |x : A|\ Q'\}) \quad (\approx [\text{C!}\bot]) \\
&\approx \textbf{wait}\ y; R, \text{for some normal } R \\
&\qquad (\text{i.h. } (A, H_2, P, Q) \to (A, H_2, P, Q'))
\end{aligned}$$

Group D These cases are handled by applying one of the $\equiv$ conversions [C!M], [C!Sh] or [C!Sm] that distribute an unrestricted cut over either a mix, a share or a sum. Then we make two inductive calls. We illustrate with case [Tmix].

Case [Tmix].

We have

$$\frac{Q_1 \vdash \Delta_1; \Gamma, x : \overline{A} \quad Q_2 \vdash \Delta_2; \Gamma, x : \overline{A}}{\textbf{par}\ \{Q_1 \mid\mid Q_2\}\ \vdash \Delta_1, \Delta_2; \Gamma, x : \overline{A}}$$

where $Q = \textbf{par}\ \{Q_1 \mid\mid Q_2\}$ and $\Delta = \Delta_1, \Delta_2$.

Then

$$\begin{aligned}
\textbf{cut!}\ \{y.P\ |x : A|\ (\textbf{par}\ \{Q_1\ ||\ Q_2\})\} &\equiv \textbf{par}\ \{(\textbf{cut!}\ \{y.P\ |x : A|\ Q_1\})\ ||\ (\textbf{cut!}\ \{y.P\ |x : A|\ Q_2\})\} \\
&\qquad (\equiv \text{[C!M]}) \\
&\approx \textbf{par}\ \{R_1\ ||\ (\textbf{cut!}\ \{y.P\ |x : A|\ Q_2\})\}, \text{ for some normal } R_1 \\
&\qquad (\text{i.h. } (A, H_2, P, Q) \rightarrow (A, H_2, P, Q_1)) \\
&\approx \textbf{par}\ \{R_1\ ||\ R_2\}, \text{ for some normal } R_2 \\
&\qquad (\text{i.h. } (A, H_2, P, Q) \rightarrow (A, H_2, P, Q_1))
\end{aligned}$$

Group E [Tcut]

We have

$$Q = \textbf{cut}\ \{\textbf{cell}\ z(w.Q_1)\ \ |z|\ \ \textbf{share}\ z\ \{\textbf{fwd}\ z\ u\ ||\ Q_2\}\}$$

Then

$$\begin{aligned}
&\textbf{cut!}\ \{y.P\ |x : A|\ Q\} \\
&= \textbf{cut!}\ \{y.P\ |x : A|\ (\textbf{cut}\ \{\textbf{cell}\ z(w.Q_1)\ \ |z|\ \ \textbf{share}\ z\ \{\textbf{fwd}\ z\ u\ ||\ Q_2\}\})\} \\
&\approx \textbf{cut}\ \{\textbf{cell}\ z(w.(\textbf{cut!}\ \{y.P\ |x : A|\ Q_1\}))\ \ |z|\ \ \textbf{share}\ z\ \{\textbf{fwd}\ z\ u\ ||\ (\textbf{cut!}\ \{y.P\ |x : A|\ Q_2\})\}\} \\
&\approx \textbf{cut}\ \{R_1\ \ |z|\ \ \textbf{share}\ z\ \{\textbf{fwd}\ z\ u\ ||\ (\textbf{cut!}\ \{y.P\ |x : A|\ Q_2\})\}\}, \text{ for some normal } R_1 \\
&\qquad (\text{i.h. } (A, H_2, P, Q) \rightarrow (A, H_2, P, Q_1)) \\
&\approx \textbf{cut}\ \{R_1\ \ |z|\ \ \textbf{share}\ z\ \{\textbf{fwd}\ z\ u\ ||\ R_2\}\}, \text{ for some normal } R_2 \\
&\qquad (\text{i.h.}(A, H_2, P, Q) \rightarrow (A, H_2, P, Q_2))
\end{aligned}$$

□

E

Strong Normalisation

Relations $\equiv_c$ and $\rightarrow_c$

Definition 38 ($\equiv_c$)**.** *The relation $\equiv_c$ is the least relation in* CLASS *s.t. the following holds*

- $\equiv_c$ *satisfies all the rules listed in Fig. 2.2.*
- $\equiv_c$ *satisfies all the rules listed in Fig. 3.2 with the exception of [Sm], [0Sm], [SmSm], [MSm], [CSm], [C!Sm], [ShSm] and [TSh].*
- $\equiv_c$ *satisfies all the rules listed in Figs. 8.1, 8.2, 8.3 and E.1.*
- $\equiv_c$ *satisfies the congruence rule [TShC]*

  $$\textbf{share}\ x\ \{\textbf{take}\ x(y_1); P_1 \parallel \textbf{take}\ x(y_2); P_2\} \equiv_c \textbf{take}\ x(y_1); \textbf{share}\ x\ \{P_1 \parallel \textbf{take}\ x(y_2); P_2\}$$

Definition 39 (Collapsing Reduction $\rightarrow_c$)**.** *Reduction $\rightarrow_c$ is the least relation in* CLASS *s.t. the following holds*

- $\rightarrow_c$ *satisfies all the rules listed in Fig. 2.3, except [≡].*
- $\rightarrow_c$ *satisfies all the rules listed in Fig. 3.3.*
- $\rightarrow_c$ *satisfies rule [$\equiv_c$]*

$$\frac{P \equiv_c P' \quad P' \rightarrow_c Q' \quad Q' \equiv_c Q}{P \rightarrow_c Q}$$

Interference-Sensitive Cells, Simulation Properties

Lemma 18. *The following properties hold*

$\mathbf{par}\ \{P \parallel \mathbf{wait}\ y; Q\} \equiv_c \mathbf{wait}\ y; \mathbf{par}\ \{P \parallel Q\}$ [M$\bot$]

$\mathbf{par}\ \{P \parallel \mathbf{send}\ y(z.Q_1); Q_2\} \equiv_c \mathbf{send}\ y(z.(\mathbf{par}\ \{P \parallel Q_1\})); Q_2$[M$\otimes_1$]

$\mathbf{par}\ \{P \parallel \mathbf{send}\ y(z.Q_1); Q_2\} \equiv_c \mathbf{send}\ y(z.Q_1); \mathbf{par}\ \{P \parallel Q_2\}$[M$\otimes_2$]

$\mathbf{par}\ \{P \parallel \mathbf{recv}\ y(z); Q\} \equiv_c \mathbf{recv}\ y(z); \mathbf{par}\ \{P \parallel Q\}$ [M$\parr$]

$\mathbf{par}\ \{P \parallel y.\mathbf{inl}; Q\} \equiv_c y.\mathbf{inl}; \mathbf{par}\ \{P \parallel Q\}$ [M$\oplus_l$]

$\mathbf{par}\ \{P \parallel y.\mathbf{inr}; Q\} \equiv_c y.\mathbf{inr}; \mathbf{par}\ \{P \parallel Q\}$ [M$\oplus_r$]

$\mathbf{par}\ \{P \parallel \mathbf{case}\ y\ \{|\mathsf{inl} : Q_1 \mid \mathsf{inr} : Q_2\}\}$
$\quad \equiv_c \mathbf{case}\ y\ \{|\mathsf{inl} : \mathbf{par}\ \{P \parallel Q_1\} \mid \mathsf{inr} : \mathbf{par}\ \{P \parallel Q_2\}\}$ [M&]

$\mathbf{par}\ \{P \parallel ?y; Q\} \equiv_c ?y; \mathbf{par}\ \{P \parallel Q\}$ [M?]

$\mathbf{par}\ \{P \parallel \mathbf{call}\ y(z); Q\} \equiv_c \mathbf{call}\ y(z); \mathbf{par}\ \{P \parallel Q\}$ [MCall]

$\mathbf{par}\ \{P \parallel \mathbf{sendty}\ y\ A; Q\} \equiv_c \mathbf{sendty}\ y\ A; \mathbf{par}\ \{P \parallel Q\}$ [M$\exists$]

$\mathbf{par}\ \{P \parallel \mathbf{recvty}\ y(X); Q\} \equiv_c \mathbf{recvty}\ y(X); \mathbf{par}\ \{P \parallel Q\}$ [M$\forall$]

$\mathbf{par}\ \{P \parallel \mathbf{unfold}_\mu\ y; Q\} \equiv_c \mathbf{unfold}_\mu\ y; \mathbf{par}\ \{P \parallel Q\}$ [Mμ]

$\mathbf{par}\ \{P \parallel \mathbf{unfold}_\nu\ y; Q\} \equiv_c \mathbf{unfold}_\nu\ y; \mathbf{par}\ \{P \parallel Q\}$ [Mν]

$\mathbf{par}\ \{P \parallel \mathbf{affine}\ y; Q\} \equiv_c \mathbf{affine}\ y; \mathbf{par}\ \{P \parallel Q\}$ [MAffine]

$\mathbf{par}\ \{P \parallel \mathbf{use}\ y; Q\} \equiv_c \mathbf{use}\ y; \mathbf{par}\ \{P \parallel Q\}$ [MUse]

$\mathbf{par}\ \{P \parallel \mathbf{cell}\ y(z.Q)\} \equiv_c \mathbf{cell}\ y(z.\mathbf{par}\ \{P \parallel Q\})$ [MCell]

$\mathbf{par}\ \{P \parallel \mathbf{take}\ y(z); Q\} \equiv_c \mathbf{take}\ y(z); \mathbf{par}\ \{P \parallel Q\}$ [MTake]

$\mathbf{par}\ \{P \parallel \mathbf{put}\ y(z.Q_1); Q_2\} \equiv_c \mathbf{put}\ y(z.(\mathbf{par}\ \{P \parallel Q_1\})); Q_2$[MPut1]

$\mathbf{par}\ \{P \parallel \mathbf{put}\ y(z.Q_1); Q_2\} \equiv_c \mathbf{put}\ y(z.Q_1); \mathbf{par}\ \{P \parallel Q_2\}$[MPut2]

Figure E.1: Mix-action commuting conversions $\equiv_c$.

(1) *Let* $S \subseteq \{R \mid R \vdash_\eta y : \wedge A\}$, $P \in S$, $Q \vdash_\eta x : \boldsymbol{U}_f\ \overline{A}$ *and suppose* Q *is* S*-preserving on* x*. Then,* $\mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q\}$ *is simulated by* $\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\}$.

(2) *Let* $S \subseteq \{R \mid R \vdash_\eta y : \wedge A\}$, $Q \vdash_\eta x : \boldsymbol{U}_e\ \overline{A}$ *and* Q *suppose* Q *is* S*-preserving on* x*. Then,* $\mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ Q\}$ *is simulated by* $\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ Q\}$.

Proof. Define

$$\mathcal{S} \triangleq \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$$

where

$$\begin{array}{rcl}
\mathcal{S}_1 & \triangleq & \{(M,N) \mid \exists P \in S, \exists Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}.\ Q \text{ is } S\text{-preserving on } x \text{ and} \\
& & M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q\} \text{ and } N \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\}\} \\
\mathcal{S}_2 & \triangleq & \{(M,N) \mid \exists Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}.\ Q \text{ is } S\text{-preserving on } x \text{ and} \\
& & M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ Q\}\ \text{ and } N \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ Q\}\} \\
\mathcal{S}_3 & \triangleq & \{(M,N) \mid M \equiv_{\mathsf{c}} N\}
\end{array}$$

We prove that $\mathcal{S}$ is a simulation. Suppose $(M,N) \in \mathcal{S}$ and $M \to_{\mathsf{c}} M'$. We perform first case analysis on $(M,N) \in \mathcal{S}$.

Case: $(M,N) \in \mathcal{S}_1$. Then

$$M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q\}$$

and

$$N \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\}$$

where $P \in S$ and $Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}$.

We perform case analysis on the reduction $M \to_{\mathsf{c}} M'$.

Case: Internal reduction of Q.

Then

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.P)\ \ |x|\ \ Q'\}$$

Let

$$N' \triangleq \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q'\}$$

Then, $N \to_{\mathsf{c}} N'$ and $(M',N') \in \mathcal{S}_1$.

Case: Cell-take interaction on session x.

Then, $Q \equiv_{\mathsf{c}} \mathbf{take}\ x(y); Q'$ and

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x\ \ |x|\ \ (\mathbf{cut}\ \{R\ \ |y|\ \ Q'\})\}$$

where $R \in S$.

Since, by hypothesis, $Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}$ and $Q \equiv_{\mathsf{c}} \mathbf{take}\ x(y); P'$, then $Q' \vdash_\eta x : \mathbf{U}_e\ \overline{A}, y : \vee\overline{A}$. Since $R \in S$, then $R \vdash_\eta y : \wedge A$, hence $\mathbf{cut}\ \{R\ \ |y|\ \ P'\} \vdash_\eta x : \mathbf{U}_e\ \overline{A}$ is S-preserving (Def. 31(a)).

Let

$$N' \triangleq \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ (\mathbf{cut}\ \{R\ \ |y|\ \ Q'\})\}$$

Then, $N \to_{\mathsf{c}} N'$ and $(M',N') \in \mathcal{S}_2$.

Case: Cell-release interaction on session x.

Then, $Q \equiv_c C[\textbf{release}\ x]$ and

$$\begin{aligned} M &\equiv_c \textbf{cut}\ \{\textbf{cell}\ x(y.P)\ \ |x|\ \ C[\textbf{release}\ x]\} \\ &\to_c C[\textbf{cut}\ \{P\ \ |y|\ \ \textbf{discard}\ y\}] \end{aligned}$$

Let

$$N' \triangleq C[\textbf{cut}\ \{P\ \ |y|\ \ \textbf{discard}\ y\}]$$

Then, since $P \in S$:

$$\begin{aligned} N &\equiv_c \textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ C[\textbf{release}\ x]\} \\ &\to_c C[\textbf{cut}\ \{P\ \ |y|\ \ \textbf{discard}\ y\}] = N' \end{aligned}$$

and $(M', N') \in \mathcal{S}_3$.

Case: $(M, N) \in \mathcal{S}_2$. Then

$$M \equiv_c \textbf{cut}\ \{\textbf{empty}\ x\ \ |x|\ \ Q\}$$

and

$$N \equiv_c \textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ Q\}$$

where $Q \vdash_\eta x : \mathbf{U}_f\ \overline{A}$.

We perform case analysis on the reduction $M \to_c M'$.

Case: Internal reduction of Q.

Then

$$M' \equiv_c \textbf{cut}\ \{\textbf{empty}\ x\ \ |x|\ \ Q'\}$$

Let

$$N' \triangleq \textbf{cut}\ \{\textbf{empty}\ x(y.S)\ \ |x|\ \ Q'\}$$

Then, $N \to_c N'$ and $(M', N') \in \mathcal{S}_2$.

Case: Cell-put interaction on session x.

Then, $Q \equiv_c \textbf{put}\ x(y.Q_1); Q_2$.

By hypothesis, $Q \vdash_\eta x : \mathbf{U}_e\ \overline{A}$, hence $Q_2 \vdash_\eta x : \mathbf{U}_f\ \overline{A}$.

Furthermore, since Q is S-preserving on x, then $Q_1 \in S$ and Q_2 is S-preserving on x (Def. 31(b)).

Then

$$M' \equiv_c \textbf{cut}\ \{\textbf{cell}\ x(y.Q_1)\ \ |x|\ \ Q_2\}$$

Let

$$N' \triangleq \textbf{cut}\ \{\textbf{cell}\ x(y.S)\ \ |x|\ \ Q_2\}$$

Then, $N \to_c N'$ and $(M', N') \in \mathcal{S}_1$.

Case: $(M, N) \in \mathcal{S}_3$.

Trivial since $M \equiv_{\mathsf{c}} N$.

□

Lemma 20. *The following pair of simulations hold*

(1) *Let* $P \vdash_\eta x : \boldsymbol{U}_f\ A$, $Q \vdash_\eta x : \boldsymbol{U}_f\ A$ *and* $S \subseteq \{R \mid R \vdash_\eta y : \wedge\overline{A}\}$. *Then,*

$$\begin{array}{c}\mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\} \\ \textit{simulates} \\ \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\}\end{array}$$

(2) *Let* $P \vdash_\eta x : \boldsymbol{U}_e\ A$, $Q \vdash_\eta x : \boldsymbol{U}_f\ A$ *and* $S \subseteq \{R \mid R \vdash_\eta y : \wedge\overline{A}\}$. *Then,*

$$\begin{array}{c}\mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\} \\ \textit{simulates} \\ \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\}\end{array}$$

Proof. Define

$$\mathcal{S} \triangleq \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$$

where

$$\begin{array}{rcl}\mathcal{S}_1 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta x : \mathbf{U}_f\ A, \exists Q \vdash_\eta x : \mathbf{U}_f\ A.\ M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\} \\ & & \text{and } N \equiv_{\mathsf{c}} \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}\} \\ \mathcal{S}_2 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta x : \mathbf{U}_e\ A, \exists Q \vdash_\eta x : \mathbf{U}_f\ A.\ M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\} \\ & & \text{and } N \equiv_{\mathsf{c}} \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}\} \\ \mathcal{S}_3 & \triangleq & \{(M,N) \mid \exists P \vdash_\eta \emptyset;\emptyset, \exists \mathcal{C} \exists \mathcal{D}.\ M \equiv_{\mathsf{c}} \mathcal{C} \circ \mathcal{D}[P] \text{ and } N \equiv_{\mathsf{c}} \mathbf{par}\ \{\mathcal{C}[P]\ ||\ \mathcal{D}[P]\}\}\end{array}$$

We prove that $\mathcal{S}$ is a simulation. Suppose $(M, N) \in \mathcal{S}$ and $M \to_{\mathsf{c}} M'$. We perform first case analysis on $(M, N) \in \mathcal{S}$.

Case: $(M, N) \in \mathcal{S}_1$. Then

$$M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\}$$

and

$$N \equiv_{\mathsf{c}} \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{S\ \ |x|\ \ Q\})\}$$

where $P \vdash_\eta x : \mathbf{U}_f\ A$ and $Q \vdash_\eta x : \mathbf{U}_f\ A$.

We perform case analysis on the reduction $M \to_{\mathsf{c}} M'$.

Case: Internal reduction of either P or Q.

Suppose w.l.o.g. that $M \to_{\mathsf{c}} M'$ is obtained by an internal reduction $P \to_{\mathsf{c}} P'$.

Then

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P'\ ||\ Q\}\}$$

Let

$$N' \triangleq \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ P'\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}$$

Then, $N \rightarrow_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_1$.

Case: Cell-take interaction on session x.

Suppose w.l.o.g. that the interaction occurs between the cell and P.

Then, $P \equiv_{\mathsf{c}} \mathbf{take}\ x(y); P'$ and

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathbf{cut}\ \{R\ \ |y|\ \ P'\}\ ||\ Q\}\}$$

where $R \in S$.

Since, by hypothesis, $P \vdash_\eta x : \mathbf{U}_f\ A$ and $P \equiv_{\mathsf{c}} \mathbf{take}\ x(y); P'$, then $P' \vdash_\eta x : \mathbf{U}_e\ A, y : \vee A$. Since $R \in S$, then $R \vdash_\eta y : \wedge\overline{A}$, hence $\mathbf{cut}\ \{R\ \ |y|\ \ P'\} \vdash_\eta x : \mathbf{U}_e\ A$.

Let

$$N' \triangleq \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ (\mathbf{cut}\ \{R\ \ |y|\ \ P'\})\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}$$

Then, $N \rightarrow_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_2$.

Case: Cell-release interaction on session x.

Both $P \equiv_{\mathsf{c}} \mathbf{release}\ x$ and $Q \equiv_{\mathsf{c}} \mathbf{release}\ x$, i.e. $P \equiv \mathcal{C}[\mathbf{release}\ x]$ and $Q \equiv_{\mathsf{c}} \mathcal{D}[\mathbf{release}\ x]$, for some static contexts $\mathcal{C}, \mathcal{D}$.

Then

$$\begin{aligned} M \ &\equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{\mathcal{C}[\mathbf{release}\ x]\ ||\ \mathcal{D}[\mathbf{release}\ x]\}\} \\ &\equiv_{\mathsf{c}} \mathcal{C} \circ \mathcal{D}[\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{release}\ x\}] \\ &\rightarrow_{\mathsf{c}} \mathcal{C} \circ \mathcal{D}[\mathbf{cut}\ \{R\ \ |y|\ \ \mathbf{discard}\ y\}] \end{aligned}$$

where $R \in S$.

Let

$$N' \triangleq \mathbf{par}\ \{\mathcal{C}[\mathbf{cut}\ \{R\ \ |y|\ \ \mathbf{discard}\ y\}]\ ||\ \mathcal{D}[\mathbf{cut}\ \{R\ \ |y|\ \ \mathbf{discard}\ y\}]\}$$

Then $N \xrightarrow{2}_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_3$.

Case: $(M, N) \in \mathcal{S}_2$. Then

$$M \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P\ ||\ Q\}\}$$

and

$$N \equiv_{\mathsf{c}} \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ P\})\ ||\ (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}$$

where $P \vdash_\eta x : \mathbf{U}_e\ A$ and $Q \vdash_\eta x : \mathbf{U}_f\ A$.

We perform case analysis on the reduction $M \rightarrow_{\mathsf{c}} M'$.

Case: Internal reduction of either P or Q.

Suppose w.l.o.g. that $M \to_{\mathsf{c}} M'$ is obtained by an internal reduction $P \to_{\mathsf{c}} P'$.

Then

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P' \parallel Q\}\}$$

Let

$$N' \triangleq \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{empty}\ x(y.S)\ \ |x|\ \ P'\}) \parallel (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}$$

Then, $N \to_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_2$.

Case: Cell-put interaction on session x.

Then, $P \equiv_{\mathsf{c}} \mathbf{put}\ x(y.P_1); P_2$.

By hypothesis, $P \vdash_\eta x : \mathbf{U}_e\ A$, hence $P_2 \vdash_\eta x : \mathbf{U}_f\ A$.

Then

$$M' \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ \mathbf{share}\ x\ \{P_2 \parallel Q\}\}$$

Let

$$N' \triangleq \mathbf{par}\ \{(\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ P_2\}) \parallel (\mathbf{cut}\ \{\mathbf{cell}\ x(y.S)\ \ |x|\ \ Q\})\}$$

Then, $N \to_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_1$.

Case: $(M, N) \in \mathcal{S}_3$.

Then

$$M \equiv_{\mathsf{c}} C \circ \mathcal{D}[P]$$

and

$$N \equiv_{\mathsf{c}} \mathbf{par}\ \{C[P] \parallel \mathcal{D}[P]\}$$

where $P \vdash_\eta \emptyset; \emptyset$.

We perform case analysis on the reduction $M \to_{\mathsf{c}} M'$.

Case: Internal reduction of either C or $\mathcal{D}$.

Suppose w.l.o.g. that $C \to_{\mathsf{c}} C'$. Then

$$M' \equiv_{\mathsf{c}} C' \circ \mathcal{D}[P]$$

Let

$$N' \triangleq \mathbf{par}\ \{C'[P] \parallel \mathcal{D}[P]\}$$

Then, $N \to_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_3$.

Case: Internal reduction of P.

Suppose $P \to_{\mathsf{c}} P'$.

Then

$$M \equiv_{\mathsf{c}} C \circ \mathcal{D}[P']$$

Let $N' \triangleq \mathbf{par}\,\{C[P'] \mid\mid \mathcal{D}[P']\}$.

Then, $N \xrightarrow{2}_{\mathsf{c}} N'$ and $(M', N') \in \mathcal{S}_3$.

□

Fundamental Lemma

Lemma Fundamental Lemma 27. *If $P \vdash_\eta \Delta;\Gamma$, then $P \in \mathcal{L}[\![\vdash_\eta \Delta;\Gamma]\!]_\sigma$.*

Proof. By induction on the structure of a typing derivation for $P \vdash_\eta \Delta;\Gamma$. Cases [Tcut], [Tfwd], [Tcut!] follow immediately because $[\![x : A]\!] = [\![x : \overline{A}]\!]^\perp$. Case [T**0**] follows because **0** is SN and case [Tmix] follows because $\mathbf{par}\,\{P \mid\mid Q\}$ is SN whenever P and Q are SN. For the positive types A, the logical predicate $[\![x : A]\!]_\sigma$ is defined as the biorthogonal of some set S, hence for the typing rules that introduce a positive type A the strategy is to show that the introduced action P lies in $S \subseteq S^{\perp\perp}$. For the negative types $\overline{A}$: $[\![x : \overline{A}]\!]_\sigma = S^{\perp\perp\perp} = S^\perp$, hence the strategy for the typing rules that that introduce an action Q that types with a negative type $x : \overline{A}$ is to show that $\mathbf{cut}\,\{P \;\; |x : A| \;\; Q\}$ is SN, for all $P \in S$. Particularly, for rule [Tcorec], where $A = \mu X.\, B$, we proceed by induction on the depth n of unfolding, since $S \bigcup_{n\in\mathbb{N}} \phi_B^n(\emptyset^{\perp\perp})$. Cases [Tcell] and [Tempty] follow by applying the simulations Lemma 18(1)-(2). Cases [Tsh], [TshL], [TshR] follows after applying the *decomposition of the share as a mix* as given by Lemma 20(1)-(2). We illustrate the proof with some cases. In the cases in which the recursive map η that annotates the typing judgments $P \vdash_\eta \Delta;\Gamma$ plays no role and is essentially propagated from the conclusion to the premises of the typing rule we omit it, working as if the process P did not have any free recursion variable X. Similarly for the map σ which annotates the logical predicates $[\![x : A]\!]_\sigma$.

Case: [T**0**]:

$$\frac{}{\mathbf{0} \vdash \cdot;\Gamma}$$

Let $C_! \in [\![\Gamma]\!]^!$.

Then, $C_![\mathbf{0}]$ is SN.

Case [Tmix]:

$$\frac{P_1 \vdash \Delta_1;\Gamma \quad P_2 \vdash \Delta_2;\Gamma}{\mathbf{par}\,\{P_1 \mid\mid P_2\} \vdash \Delta_1,\Delta_2;\Gamma}$$

Let $C \in [\![\Delta_1,\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$C \circ \mathcal{D}[\mathbf{par}\ \{P_1 \mid\mid P_2\}] \equiv_c \mathbf{par}\ \{(C_1 \circ \mathcal{D}[P_1]) \mid\mid (C_2 \circ \mathcal{D}[P_2])\}$$

where $C_1 \in [\![\Delta_1]\!]$ and $C_2 \in [\![\Delta_2]\!]$.

I.h. applied to $P_1 \vdash \Delta_1; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1]$ is SN.

I.h. applied to $P_2 \vdash \Delta_2; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2]$ is SN.

By applying Lemma 21(4) we conclude that $\mathbf{par}\ \{(C_1 \circ \mathcal{D}[P_1]) \mid\mid (C_2 \circ \mathcal{D}[P_2])\}$ is SN.

Hence, $C \circ \mathcal{D}[\mathbf{par}\ \{P_1 \mid\mid P_2\}]$ is SN.

Case [Tfwd]:

$$\frac{}{\mathbf{fwd}\ x\ y \vdash x : A, y : \overline{A}; \Gamma}$$

Let $C \in [\![x : A, y : \overline{A}]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$C \circ \mathcal{D}[\mathbf{fwd}\ x\ y] \equiv_c \mathcal{D}[\mathbf{cut}\ \{P\ \ |x|\ \ (\mathbf{cut}\ \{Q\ \ |y|\ \ \mathbf{fwd}\ x\ y\})\}]$$

where $P \in [\![x : \overline{A}]\!]$ and $Q \in [\![y : A]\!]$.

We prove that (H) $\mathcal{D}[\mathbf{cut}\ \{P\ \ |x|\ \ (\mathbf{cut}\ \{Q\ \ |y|\ \ \mathbf{fwd}\ x\ y\})\}]$ is SN, by induction on $N(P) + N(Q)$. Suppose that $\mathcal{D}[\mathbf{cut}\ \{P\ \ |x|\ \ (\mathbf{cut}\ \{Q\ \ |y|\ \ \mathbf{fwd}\ x\ y\})\}] \to_c R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either P or Q.

Case: (ii) R is obtained by an interaction with with the forwarder $\mathbf{fwd}\ x\ y$ on either session x or y.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Suppose w.l.o.g. that R is obtained by an interaction with the forwarder $\mathbf{fwd}\ x\ y$ on session y. Then $R \equiv_c \mathcal{D}[\mathbf{cut}\ \{P\ \ |x|\ \ \{x/y\}Q\}]$.

By Lemma 23(1), $\{x/y\}Q \in [\![x : A]\!]$.

By Lemma 23(4), $\mathbf{cut}\ \{P\ \ |x|\ \ \{x/y\}Q\}$ is SN.

By Lemma 21(3), $\mathbf{cut}\ \{P\ \ |x|\ \ (\mathbf{cut}\ \{Q\ \ |y|\ \ \mathbf{fwd}\ x\ y\})\}$ is SN.

Hence, $C \circ \mathcal{D}[\mathbf{fwd}\ x\ y]$ is SN.

Case [Tcut]:

$$\frac{P_1 \vdash \Delta_1, x : \overline{A}; \Gamma \quad P_2 \vdash \Delta_2, x : A; \Gamma}{\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\} \vdash \Delta_1, \Delta_2; \Gamma}$$

Let $C_1 \in [\![\Delta_1]\!]$, $C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$C_1 \circ C_2 \circ \mathcal{D}[\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}] \equiv_{\mathsf{c}} \mathbf{cut}\ \{(C_1 \circ \mathcal{D}[P_1])\ \ |x|\ \ (C_2 \circ \mathcal{D}[P_2])\}$$

I.h. and Lemma 26 applied to $P_1 \vdash \Delta_1, x : \overline{A}; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![x : \overline{A}]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash \Delta_2, x : A; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2] \in [\![x : A]\!]$.

By applying Lemma 23(4) we conclude that $\mathbf{cut}\ \{(C_1 \circ \mathcal{D}[P_1])\ \ |x|\ \ (C_2 \circ \mathcal{D}[P_2])\}$ is SN.

Hence, $C \circ \mathcal{D}[\mathbf{cut}\ \{P_1\ \ |x|\ \ P_2\}]$ is SN.

Case [Tcut!]:

$$\frac{P_1 \vdash y : \overline{A}; \Gamma \quad P_2 \vdash \Delta; \Gamma, x : A}{\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\} \vdash \Delta; \Gamma}$$

Let $C \in [\![\Delta]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$C \circ \mathcal{D}[\mathbf{cut!}\ \{y.P_1\ |x|\ P_2\}] \equiv_{\mathsf{c}} C \circ (\mathbf{cut!}\ \{y.\mathcal{D}[P_1]\ |x|\ \mathcal{D}\})[P_2]$$

I.h. and Lemma 26 applied to $P_1 \vdash y : \overline{A}; \Gamma$ yields $\mathcal{D}[P_1] \in [\![y : \overline{A}]\!]$.

Then, $\mathbf{cut!}\ \{y.\mathcal{D}[P_1]\ |x|\ \mathcal{D}\} \in [\![\Gamma, x : A]\!]^!$.

I.h. applied to $P_2 \vdash \Delta; \Gamma, x : A$ yields $C \circ (\mathbf{cut!}\ \{y.\mathcal{D}[P_1]\ |x|\ \mathcal{D}\})[P_2]$ is SN.

Hence, $C \circ C_!\mathbf{[cut!}\ \{y.P_1\ |x|\ P_2\}]$ is SN.

Case [Tvar]:

$$\frac{\eta = \eta', X(x, \vec{y})\ \mapsto \Delta, x : Y; \Gamma}{X(z, \vec{w}) \vdash_\eta \{\vec{w}/\vec{y}\}(\Delta, z : Y; \Gamma)}$$

Let $\rho \in [\![\eta]\!]_\sigma$. Then, $\rho = \rho', X(x, \vec{y}) \mapsto Q$ where $Q \in \mathcal{L}[\![\vdash_\emptyset \Delta, x : Y; \Gamma]\!]_\sigma$ and $\rho' \in [\![\eta']\!]_\sigma$.

We have

$$\rho(X(z, \vec{w})) = \{z/x\}\{\vec{w}/\vec{y}\}Q$$

Since $Q \in \mathcal{L}[\![\vdash_\emptyset \Delta, x : Y; \Gamma]\!]_\sigma$, then $\{z/x\}\{\vec{w}/\vec{y}\}Q \in \mathcal{L}[\![\vdash_\emptyset \{\vec{w}/\vec{y}\}(\Delta, z : Y; \Gamma)]\!]$.

Hence, $X(z, \vec{w}) \in \mathcal{L}[\![\vdash_\eta \{\vec{w}/\vec{y}\}(\Delta, z : Y; \Gamma)]\!]$.

Case [Tsh]:

$$\frac{P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_f\ A; \Gamma \quad P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ c\ \{P\ ||\ Q\}\ \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_f\ A; \Gamma}$$

Let $C_1 \in [\![\Delta_1]\!]$, $C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$.

We have

$$\begin{aligned} &C_1 \circ C_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1\ ||\ P_2\}] \\ &\equiv_{\mathsf{c}} \mathbf{share}\ c\ \{C_1 \circ \mathcal{D}[P_1]\ ||\ C_2 \circ \mathcal{D}[P_2]\} \end{aligned}$$

I.h. and Lemma 26 applied to $P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_f\ A; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![c : \mathbf{U}_f\ A]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2] \in [\![c : \mathbf{U}_f\ A]\!]$.

By applying Lemma 24(1) we conclude that $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1 \mid\mid P_2\}] \in [\![c : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\mathbf{share}\ c\ \{P_1 \mid\mid P_2\} \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_f\ A; \Gamma]\!]$.

Case: [TshL]

$$\frac{P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_e\ A; \Gamma \quad P_2 \vdash_\eta \Delta, c : \mathbf{U}_f\ A; \Gamma}{\mathbf{share}\ c\ \{P_1 \mid\mid P_2\} \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma}$$

Let $C_1 \in [\![\Delta_1]\!]$, $C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$\begin{array}{l} C_1 \circ C_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1 \mid\mid P_2\}] \\ \equiv_{\mathsf{c}} \mathbf{share}\ c\ \{C_1 \circ \mathcal{D}[P_1] \mid\mid C_2 \circ \mathcal{D}[P_2]\} \end{array}$$

I.h. and Lemma 26 applied to $P_1 \vdash_\eta \Delta_1, c : \mathbf{U}_e\ A; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![c : \mathbf{U}_e\ A]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2] \in [\![c : \mathbf{U}_f\ A]\!]$.

By applying Lemma 24(2) we conclude that $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{share}\ c\ \{P_1 \mid\mid P_2\}] \in [\![c : \mathbf{U}_e\ A]\!]$.

By Lemma 26, $\mathbf{share}\ c\ \{P_1 \mid\mid P_2\} \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma]\!]$.

Case: [TshL]. Similarly to case [TshR].

Case: [T**1**]

$$\frac{}{\mathbf{close}\ x \vdash_\eta x : \mathbf{1}; \Gamma}$$

By def. 32

$$\begin{array}{lcl} [\![x : \mathbf{1}]\!] & \triangleq & S^{\perp\perp}, \text{ where} \\ S & = & \{Q \vdash x : \mathbf{1} \mid Q \equiv_{\mathsf{c}} \mathbf{close}\ x\}. \end{array}$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$. We have $\mathcal{D}[\mathbf{close}\ x] \approx \mathbf{close}\ x$. Hence, $\mathcal{D}[\mathbf{close}\ x] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $\mathcal{D}[\mathbf{close}\ x] \in [\![x : \mathbf{1}]\!]$.

Lemma 26 implies that $\mathbf{close}\ x \in \mathcal{L}[\![x : \mathbf{1}; \Gamma]\!]$.

Case: [T⊗]

$$\frac{P_1 \vdash_\eta \Delta_1, y : A; \Gamma \quad P_2 \vdash_\eta \Delta_2, x : B; \Gamma}{\mathbf{send}\ x(y.P_1); P_2 \vdash_\eta \Delta_1, \Delta_2, x : A \otimes B; \Gamma}$$

By def. 32, $[\![x : A \otimes B]\!] = S^{\perp\perp}$, where

$$S = \{Q \mid \exists Q_1, Q_2.\ Q \equiv_{\mathsf{c}} \mathbf{send}\ x(y.Q_1); Q_2 \text{ and } Q_1 \in [\![y : A]\!] \text{ and } Q_2 \in [\![x : B]\!]\}.$$

Let $C \in [\![\Delta_1, \Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$C \circ \mathcal{D}[\mathbf{send}\ x(y.P_1); P_2] \equiv_{\mathsf{c}} \mathbf{send}\ x(y.C_1 \circ \mathcal{D}[P_1]); C_2 \circ \mathcal{D}[P_2]$$

where $C_1 \in [\![\Delta_1]\!]$ and $C_2 \in [\![\Delta_2]\!]$.

I.h. and Lemma 26 applied to $P_1 \vdash_\eta \Delta_1, y : A; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![y : A]\!]$.

I.h. and Lemma 26 applied to $P_2 \vdash_\eta \Delta_2, x : B; \Gamma$ yields $C_2 \circ \mathcal{D}[P_2] \in [\![x : B]\!]$.

Hence, $C \circ \mathcal{D}[\mathbf{send}\ x(y.P_1); P_2] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $C \circ \mathcal{D}[\mathbf{send}\ x(y.P_1); P_2] \in [\![x : A \otimes B]\!]$.

Lemma 26 implies that $\mathbf{send}\ x(y.P_1); P_2 \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, x : A \otimes B; \Gamma]\!]$.

Case: [T$\oplus_l$]

$$\frac{P_1 \vdash_\eta \Delta', x : A; \Gamma}{x.\mathbf{inl}; P_1 \ \vdash_\eta \Delta', x : A \oplus B; \Gamma}$$

By def. 32, $[\![x : A \oplus B]\!] = S^{\perp\perp}$, where

$$S = \{Q \mid \exists Q'.\, (Q \equiv_c x.\mathbf{inl}; Q' \text{ and } Q' \in [\![x : A]\!]) \text{ or } (Q \equiv_c x.\mathbf{inr}; Q' \text{ and } Q' \in [\![x : B]\!])\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$C \circ \mathcal{D}[x.\mathbf{inl}; P_1] \equiv_c x.\mathbf{inl}; C \circ \mathcal{D}[P_1]$$

I.h. and Lemma 26 applied to $P_1 \vdash_\eta \Delta', x : A; \Gamma$ yields $C \circ \mathcal{D}[P_1] \in [\![x : A]\!]$.

Hence, $C \circ \mathcal{D}[x.\mathbf{inl}; P_1] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $C \circ \mathcal{D}[x.\mathbf{inl}; P_1] \in [\![x : A \oplus B]\!]$.

Lemma 26 implies that $x.\mathbf{inl}; P_1 \in \mathcal{L}[\![\vdash_\eta \Delta', x : A \oplus B; \Gamma]\!]$.

Case: [T$\oplus_r$]. Similarly to case [T$\oplus_l$].

Case: [T!]

$$\frac{P' \vdash_\eta y : A; \Gamma}{!x(y); P' \ \vdash_\eta x : !A; \Gamma}$$

By def. 32, $[\![x :!A]\!] = S^{\perp\perp}$, where

$$S = \{Q \mid \exists Q'.\, Q \equiv_c !x(y); Q' \text{ and } Q' \in [\![y : A]\!]\}.$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$\mathcal{D}[!x(y); P'] \equiv_c !x(y); \mathcal{D}[P']$$

I.h. and Lemma 26 applied to $P' \vdash_\eta y : A; \Gamma$ yields $\mathcal{D}[P'] \in [\![y : A]\!]$.

Hence, $\mathcal{D}[P'] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $\mathcal{D}[P'] \in [\![x :!A]\!]$.

Lemma 26 implies that $!x(y); P' \in \mathcal{L}[\![\ \vdash_\eta x : !A; \Gamma]\!]$.

Case: [T∃]

$$\frac{P' \vdash_\eta \Delta, x : \{B/X\}A; \Gamma}{\mathbf{sendty}\ x\ B; P' \vdash_\eta \Delta, x : \exists X.A; \Gamma}\ [\mathrm{T}\exists]$$

By def. 32, $[\![x : \exists X.A]\!] = S^{\perp\perp}$, where

$$S = \{Q \mid \exists Q', S' \in \mathcal{R}[- : B].\ Q \equiv_{\mathsf{c}} \mathbf{sendty}\ x\ B; Q' \text{ and } Q' \in [\![x : A]\!]_{\sigma[X \mapsto S']}\}.$$

Let $\mathcal{C} \in [\![\Delta]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$\mathcal{C} \circ \mathcal{D}[\mathbf{sendty}\ x\ B; P'] \equiv_{\mathsf{c}} \mathbf{sendty}\ x\ B; \mathcal{C} \circ \mathcal{D}[P']$$

I.h. and Lemma 26 applied to $P' \vdash_\eta \Delta, x : \{B/X\}A; \Gamma$ yields $\mathcal{C} \circ \mathcal{D}[P'] \in [\![x : \{B/X\}A]\!]$.

By Lemma 23(5), $\mathcal{C} \circ \mathcal{D}[P'] \in [\![x : A]\!]_{\sigma[X \mapsto [\![x:B]\!]]}$.

Hence, $\mathcal{C} \circ \mathcal{D}[\mathbf{sendty}\ x\ B; P'] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $\mathcal{C} \circ \mathcal{D}[\mathbf{sendty}\ x\ B; P'] \in [\![x : \exists X.A]\!]$.

Lemma 26 implies that $\mathbf{sendty}\ x\ B; P' \in \mathcal{L}[\![\vdash_\eta \Delta, x : \exists X.A; \Gamma]\!]$.

Case: [Tμ]

$$\frac{P' \vdash_\eta \Delta', x : \{\mu X.\ A/X\}A; \Gamma}{\mathbf{unfold}_\mu\ x; P' \vdash_\eta \Delta', x : \mu X.\ A; \Gamma}$$

Let $\mathcal{C} \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$\mathcal{C} \circ \mathcal{D}[\mathbf{unfold}_\mu\ x; P'] \equiv_{\mathsf{c}} \mathbf{unfold}_\mu\ x; \mathcal{C} \circ \mathcal{D}[P']$$

I.h. and Lemma 26 applied to $P' \vdash_\eta \Delta', x : \{\mu X.\ A/X\}A; \Gamma$ yields $\mathcal{C} \circ \mathcal{D}[P'] \in [\![x : \{\mu X.\ A/X\}A]\!]$.

By Lemma 25(2), $[\![x : \mu X.\ A]\!] = \mathbf{unfold}_\mu\ x; [\![x : \{\mu X.\ A/X\}A]\!]_\sigma$, hence $\mathcal{C} \circ \mathcal{D}[\mathbf{unfold}_\mu\ x; P'] \in [\![x : \mu X.\ A]\!]_\sigma$.

Lemma 26 implies that $\mathbf{unfold}_\mu\ x; P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : \mu X.\ A; \Gamma]\!]$.

Case: [Tν]

$$\frac{P' \vdash_\eta \Delta', x : \{\mu X.\ A/X\}A; \Gamma}{\mathbf{unfold}_\nu\ x; P' \vdash_\eta \Delta', x : \nu X.\ A; \Gamma}$$

Let $\mathcal{C} \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$\mathcal{C} \circ \mathcal{D}[\mathbf{unfold}_\nu\ x; P'] \equiv_{\mathsf{c}} \mathbf{unfold}_\nu\ x; \mathcal{C} \circ \mathcal{D}[P']$$

I.h. and Lemma 26 applied to $P' \vdash_\eta \Delta', x : \{\mu X.\ A/X\}A; \Gamma$ yields $\mathcal{C} \circ \mathcal{D}[P'] \in [\![x : \{\nu X.\ A/X\}A]\!]$.

By Lemma 25(5), $\mathbf{unfold}_\nu\ x; [\![x : \{\nu X.\ A/X\}A]\!]_\sigma \subseteq [\![x : \nu X.\ A]\!]_\sigma$, hence $\mathcal{C} \circ \mathcal{D}[\mathbf{unfold}_\nu\ x; P'] \in [\![x : \nu X.\ A]\!]_\sigma$.

Lemma 26 implies that $\mathbf{unfold}_\nu\ x; P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : \mu X.\ A; \Gamma]\!]$.

Case: [Taffine]

$$\frac{P' \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, x : A; \Gamma}{\mathbf{affine}\ x; P' \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, a : \wedge A; \Gamma}$$

By def. 32, $[\![x : \wedge A]\!] = S^{\perp\perp}$, where

$$S = \{Q \mid \exists Q'.\ Q \equiv_c \mathbf{affine}\ x; Q' \text{ and } Q' \in [\![x : A]\!]\}.$$

Let $C \in [\![\vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$. We have

$$C \circ \mathcal{D}[\mathbf{affine}\ x; P'] \equiv_c \mathbf{affine}\ x; C \circ \mathcal{D}[P']$$

I.h. and Lemma 26 applied to $P' \vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, x : A; \Gamma$ yields $C \circ \mathcal{D}[P'] \in [\![x : A]\!]$.

Hence, $C \circ \mathcal{D}[\mathbf{affine}\ x; P'] \in S$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, thus $C \circ \mathcal{D}[\mathbf{affine}\ x; P'] \in [\![x : \wedge A]\!]$.

Lemma 26 implies that $\mathbf{affine}\ x; P' \in \mathcal{L}[\![\vdash_\eta \vec{c} : \mathbf{U}_f\ \vec{B}, \vec{a} : \vee\vec{C}, x : A; \Gamma]\!]$.

Case: [Tcell]

$$\frac{P' \vdash_\eta \Delta', a : \wedge A; \Gamma}{\mathbf{cell}\ c(a.P') \vdash_\eta \Delta', c : \mathbf{S}_f\ A; \Gamma}$$

Let $C \in [\![\Delta']\!]$, $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!]$.

I.h. and Lemma 26 applied to $P' \vdash_\eta \Delta', a : \wedge A; \Gamma$ yields $C \circ \mathcal{D}[P'] \in [\![a : \wedge A]\!]$.

Since $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!]$, then Q is $[\![a : \wedge A]\!]$-preserving.

Hence, by Lemma 18(1), $\mathbf{cut}\ \{\mathbf{cell}\ c(a.C \circ \mathcal{D}[P'])\ |c|\ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge A]\!])\ |c|\ Q\}$.

Since $Q \in [\![c : \mathbf{U}_f\ \overline{A}]\!] = S^\perp$ where $S = \{R \mid R \equiv_c \mathbf{cell}\ c(a.[\![a : \wedge A]\!])\}$, then $\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge A]\!])\ |c|\ Q\}$ is SN.

Hence, $\mathbf{cut}\ \{C \circ \mathcal{D}[\mathbf{cell}\ c(a.P')]\ |c|\ Q\}$ is SN.

Then, $\mathbf{cell}\ c(a.P') \in \mathcal{L}[\![\vdash_\eta \Delta', c : \mathbf{S}_f\ A; \Gamma]\!]$.

Case: [Tempty]

$$\frac{}{\mathbf{empty}\ c \vdash_\eta c : \mathbf{S}_e\ A; \Gamma}$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!]$.

Since $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!]$, then Q is $[\![a : \wedge A]\!]$-preserving.

Hence, by Lemma 18(2), $\mathbf{cut}\ \{\mathbf{empty}\ c\ \ |c|\ \ Q\}$ is simulated by $\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ Q\}$.

Since $Q \in [\![c : \mathbf{U}_e\ \overline{A}]\!] = S^\perp$ where $S = \{R \mid R \equiv_c \mathbf{empty}\ c([\![a : \wedge A]\!].)\}$, then $\mathbf{cut}\ \{\mathbf{empty}\ c([\![a : \wedge A]\!].)\ |c|\ Q\}$ is SN.

Hence, $\mathbf{cut}\ \{\mathcal{D}[\mathbf{empty}\ c]\ |c|\ Q\}$ is SN.

Then, $\mathbf{empty}\ c \in \mathcal{L}[\![\vdash_\eta c : \mathbf{S}_e\ A; \Gamma]\!]$.

Case: [T⊥]

$$\frac{P' \vdash_\eta \Delta'; \Gamma}{\mathbf{wait}\ x; P' \vdash_\eta \Delta', x : \bot; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \bot]\!] = S^\bot$, where

$$S = \{Q \vdash x : \mathbf{1} \mid Q \equiv_c \mathbf{close}\ x\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_c \mathbf{close}\ x$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{wait}\ x; P']\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{wait}\ x; P']\} \to_c R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{wait}\ x; P']\} \equiv_c \mathbf{cut}\ \{\mathbf{close}\ x \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{wait}\ x; P']\} \to_c C \circ \mathcal{D}[P'] = R$$

Applying i.h. to $P' \vdash_\eta \Delta'; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{wait}\ x; P']\}$ is SN.

Therefore, $C \circ \mathcal{D}[\mathbf{wait}\ x; P'] \in [\![x : \bot]\!]$.

By Lemma 26, $\mathbf{wait}\ x; P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : \bot; \Gamma]\!]$.

Case: [T⅋]

$$\frac{P' \vdash_\eta \Delta', z : A, x : B; \Gamma}{\mathbf{recv}\ x(z); P' \vdash_\eta \Delta', x : A \parr B; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : A \parr B]\!] = S^\bot$, where

$$S = \{Q \mid \exists Q_1, Q_2.\ Q \equiv_c \mathbf{send}\ x(y.Q_1); Q_2 \text{ and } Q_1 \in [\![y : \overline{A}]\!] \text{ and } Q_2 \in [\![x : \overline{B}]\!]\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_c \mathbf{send}\ x(y.Q_1); Q_2$ and $Q_1 \in [\![y : \overline{A}]\!]$ and $Q_2 \in [\![x : \overline{B}]\!]$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recv}\ x(z); P']\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recv}\ x(z); P']\} \to_c R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\begin{aligned}\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recv}\ x(z); P']\} \quad &\equiv_c \mathbf{cut}\ \{\mathbf{send}\ x(y.Q_1); Q_2\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recv}\ x(z); P']\} \\ &\to_c \mathbf{cut}\ \{Q_2\ \ |x|\ \ (\mathbf{cut}\ \{Q_1\ \ |y|\ \ C \circ \mathcal{D}[\{y/z\}P']\})\} \ = R\end{aligned}$$

Applying i.h. to $\{y/z\}P' \vdash_\eta \Delta', y : A, x : B; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recv}\ x(z); P']\}$ is SN.

Therefore, $C \circ \mathcal{D}[\mathbf{recv}\ x(z); P'] \in [\![x : A \parr B]\!]$.

By Lemma 26, $\mathbf{recv}\ x(z); P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : A \parr B; \Gamma]\!]$.

Case: [T&]

$$\frac{P_1 \vdash_\eta \Delta', x : A; \Gamma \quad P_2 \vdash_\eta \Delta', x : B; \Gamma}{\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}\ \vdash_\eta \Delta', x : A \,\&\, B; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : A \,\&\, B]\!] = S^\perp$, where

$$S = \{Q \mid \exists Q'.(Q \equiv_c x.\mathbf{inl}; Q' \text{ and } Q' \in [\![x : \overline{A}]\!]) \text{ or } (Q \equiv_c x.\mathbf{inr}; Q' \text{ and } Q' \in [\![x : \overline{B}]\!])\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Suppose that $Q \equiv_c x.\mathbf{inl}; Q'$ and $Q' \in [\![x : \overline{A}]\!]$. The case in which choice is right is handled similarly.

We prove that (H) $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}]\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}]\} \to_c R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii).

$$\begin{aligned}&\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}]\} \\ &\equiv_c \mathbf{cut}\ \{x.\mathbf{inl}; Q_1\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}]\} \\ &\to_c \mathbf{cut}\ \{Q_1\ \ |x|\ \ C \circ \mathcal{D}[P_1]\} = R\end{aligned}$$

Applying i.h. to $P_1 \vdash_\eta \Delta', x : A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}]\}$ is SN.

Therefore, $C \circ \mathcal{D}[\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\}] \in [\![x : A \,\&\, B]\!]$.

By Lemma 26, $\mathbf{case}\ x\ \{|\mathsf{inl} : P_1 \mid \mathsf{inr} : P_2\} \in \mathcal{L}[\![\vdash_\eta \Delta', x : A \,\&\, B; \Gamma]\!]$.

Case: [T?]

$$\frac{P' \vdash_\eta \Delta'; \Gamma, x : A}{?x; P' \vdash_\eta \Delta', x : ?A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : ?A]\!] = S^\perp$, where

$$S = \{Q \mid \exists Q'.\ Q \equiv_{\mathsf{c}} !x(y); Q' \text{ and } Q' \in [\![y : \overline{A}]\!]\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} !x(y); Q'$ and $Q' \in [\![y : \overline{A}]\!]$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[?x; P']\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[?x; P']\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\begin{aligned}
&\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[?x; P']\} \\
&\equiv_{\mathsf{c}} \mathbf{cut}\ \{!x(y); Q' \ \ |x|\ \ C \circ \mathcal{D}[?x; P']\} \\
&\to_{\mathsf{c}} \mathbf{cut!}\ \{y.Q' \ |x|\ C \circ \mathcal{D}[P']\} = R
\end{aligned}$$

Applying i.h. to $P' \vdash_\eta \Delta'; \Gamma, x : A$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |x|\ \ C \circ \mathcal{D}[?x; P']\}$ is SN.

Therefore, $C \circ \mathcal{D}[?x; P'] \in [\![x : ?A]\!]$.

By Lemma 26, $?x; P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : ?A; \Gamma]\!]$.

Case: [Tcall]

$$\frac{P' \vdash_\eta \Delta, z : A; \Gamma', x : A}{\mathbf{call}\ x(z); P' \vdash_\eta \Delta; \Gamma', x : A}$$

Let $C \in [\![\Delta]\!]$ and $\mathcal{D} \in [\![\Gamma', x : A]\!]^!$. We prove that (H) $C \circ \mathcal{D}[\mathbf{call}\ x(z); P']$ is SN, by induction on $N(C)$.

Suppose that $C \circ \mathcal{D}[\mathbf{call}\ x(z); P'] \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of$C$.

Case: (ii) R is obtained by an interaction on session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\begin{aligned}&C \circ \mathcal{D}[\mathbf{call}\ x(z); P']\\ &\equiv_{\mathsf{c}} \mathbf{cut!}\ \{y.Q\ |x|\ C \circ \mathcal{D}'[\mathbf{call}\ x(z); P']\}\\ &\rightarrow_{\mathsf{c}} (\mathbf{cut}\ \{\{z/y\}Q\ \ |z|\ \ C\}) \circ (\mathbf{cut!}\ \{y.Q\ |x|\ \mathcal{D}'\})[P'] = R\end{aligned}$$

Since $\mathcal{D} \in [\![\Gamma', x : A]\!]^!$, then $\mathcal{D}' \in [\![\Gamma']\!]^!$ and $Q \in [\![y : \overline{A}]\!]$.

By Lemma 23(1), $\{z/y\}Q \in [\![z : \overline{A}]\!]$.

Then, $\mathbf{cut}\ \{\{z/y\}Q\ \ |z|\ \ C\} \in [\![\Delta, z : A]\!]$ and $\mathbf{cut!}\ \{y.Q\ |x|\ \mathcal{D}'\} \in [\![\Gamma', x : A]\!]^!$.

Applying i.h. to $P' \vdash_\eta \Delta, z : A; \Gamma', x : A$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $C \circ \mathcal{D}[\mathbf{call}\ x(z); P']$ is SN.

Thus, $\mathbf{call}\ x(z); P'\ \in \mathcal{L}[\![\vdash_\eta \Delta; \Gamma', x : A]\!]$.

Case: [T$\forall$]

$$\frac{P' \vdash_\eta \Delta', x : A; \Gamma}{\mathbf{recvty}\ x(X); P' \vdash_\eta \Delta', x : \forall X.A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \forall X.A]\!] = S^\perp$, where

$$S = \{Q \mid \exists Q', S'\ \in \mathcal{R}[- : B].\ Q \equiv_{\mathsf{c}} \mathbf{sendty}\ x\ B; Q' \text{ and } Q' \in [\![x : \overline{A}]\!]_{\sigma[X \mapsto S']}\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} \mathbf{sendty}\ x\ B; Q'$ and $Q' \in [\![x : \overline{A}]\!]_{\sigma[X \mapsto S']}$, for some $S' \in \mathcal{R}[- : B]$.

We prove that (H) $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recvty}\ x(X); P']\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recvty}\ x(X); P']\} \rightarrow_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\begin{aligned}\mathbf{cut}\ \{Q\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recvty}\ x(X); P']\}\ &\equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{sendty}\ x\ B; Q'\ \ |x|\ \ C \circ \mathcal{D}[\mathbf{recvty}\ x(X); P']\}\\ &\rightarrow_{\mathsf{c}} \mathbf{cut}\ \{Q'\ \ |x|\ \ C \circ \mathcal{D}[\{B/X\}P']\}\ = R\end{aligned}$$

Applying i.h. to $\{B/X\}P' \vdash_\eta \Delta', x : \{B/X\}A; \Gamma$ and Lemma 26 yields $C \circ \mathcal{D}[\{B/X\}P'] \in [\![x : \{B/X\}A]\!]$.

By Lemma 23(5), $\mathcal{C} \circ \mathcal{D}[\{B/X\}P'] \in [\![x : A]\!]_{\sigma[X \mapsto S']}$.

Since $Q' \in [\![x : \overline{A}]\!]_{\sigma[X \mapsto S']}$ and $\mathcal{C} \circ \mathcal{D}[\{B/X\}P'] \in [\![x : A]\!]_{\sigma[X \mapsto S']}$, Lemma 23(4) yields that R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |x| \ \ \mathcal{C} \circ \mathcal{D}[\mathbf{recvty}\ x(X); P']\}$ is SN.

Therefore, $\mathcal{C} \circ \mathcal{D}[\mathbf{recvty}\ x(X); P'] \in [\![x : \forall X.A]\!]$.

By Lemma 26, $\mathbf{recvty}\ x(X); P' \in \mathcal{L}[\![\vdash_\eta \Delta', x : \forall X.A; \Gamma]\!]$.

Case: [Tcorec]

$$\frac{\{x/z\}\{\vec{y}/\vec{w}\}P' \vdash_{\eta'} \Delta', x : A; \Gamma \quad \eta' = \eta, Y(x, \vec{y}) \mapsto \Delta', x : X; \Gamma}{\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}] \vdash_\eta \Delta', x : \nu X.\ A; \Gamma}$$

Let $\rho \in [\![\eta]\!]_\sigma$, $\mathcal{C} \in [\![\Delta']\!]_\sigma$ and $\mathcal{D} \in [\![\Gamma]\!]^!_\sigma$.

We prove that $\mathcal{C} \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in [\![x : \nu X.\ A]\!]_\sigma$.

By Lemma 26, this implies that $\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}] \in \mathcal{L}[\![\vdash_\eta \Delta', x : \nu X.\ A; \Gamma]\!]_\sigma$.

By Lemma 25(5), we have

$$[\![x : \nu X.\ A]\!]_\sigma = \bigcap_{n \in \mathbb{N}} \phi^n_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp$$

where $\phi_{\{\overline{X}/X\}\overline{A}}(S) \triangleq \mathbf{unfold}_\mu\ x; [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto S]}$.

We prove (H1):

$$\forall n \in \mathbb{N},\ \forall \rho \in [\![\eta]\!]_\sigma,\ \forall \mathcal{C} \in [\![\Delta']\!]_\sigma,\ \forall \mathcal{D} \in [\![\Gamma]\!]^!_\sigma.\\ \mathcal{C} \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in \phi^n_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp$$

Proof of (H1) is by induction on $n \in \mathbb{N}$:

Case: $n = 0$.

Follows because $\mathcal{C} \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \in \emptyset^\perp$ and since $\phi^0_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^\perp = \emptyset^{\perp\perp\perp} = \emptyset^\perp$ (Lemma 22(5)).

Case: $n = m + 1$.

Let $Q \in \phi^{m+1}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})$.

Then $Q \equiv_{\mathsf{c}} \mathbf{unfold}_\mu\ x; Q'$, where $Q' \in [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto \psi^m_A(\emptyset^{\perp\perp})]}$.

We prove (H2)

$$\mathbf{cut}\ \{\mathcal{C} \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \ \ |x| \ \ Q\} \text{ is SN}$$

by induction on $N(\mathcal{C}) + N(\rho) + N(Q)$.

Suppose that $\mathbf{cut}\ \{\mathcal{C} \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z, \vec{w}); P'\ [x, \vec{y}])] \ \ |x| \ \ Q\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either C, ρ or Q.

Case: (ii) R is obtained by an interaction on session x.

Case (i) follows by inner inductive hypothesis (H2).

So let us consider case (ii). Then

$$\begin{aligned}
&\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}])]\ \ |x|\ \ Q\} \\
&\equiv_{\mathsf{c}} \mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}])]\ \ |x|\ \ \mathbf{unfold}_{\mu}\ x; Q'\} \\
&\to_{\mathsf{c}} \mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\{x/z\}\{\vec{y}/\vec{w}\}\{\mathbf{corec}\ Y(z,\vec{w}); P'/Y\}P')]\ \ |x|\ \ Q'\} \\
&= \mathbf{cut}\ \{C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')]\ \ |x|\ \ Q'\} = R
\end{aligned}$$

where $\rho' = \rho, Y(x,\vec{y}) \mapsto \rho(\mathbf{corec}\ Y(z,\vec{w}); P')$.

I.h. (H1) applied to m yields

$$\begin{aligned}
&\forall C \in [\![\Delta']\!],\ \forall \mathcal{D} \in [\![\Gamma]\!]^{!}. \\
&\quad C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}])] \in \phi^{m}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}
\end{aligned}$$

Hence, by Lemma 26, we obtain

$$\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}]) \in \mathcal{L}[\![\vdash_{\emptyset} \Delta', x : X; \Gamma]\!]_{\sigma[X \mapsto \psi^{m}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}]}$$

Therefore, $\rho' \in [\![\eta']\!]_{\sigma}$.

Applying i.h. (outer i.h., fundamental lemma) to $\{x/z\}\{\vec{y}/\vec{w}\}P' \vdash_{\eta'} \Delta', x : A; \Gamma$ and Lemma 26 yields $C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')] \in [\![x : A]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})^{\perp}]}$.

Lemma 23(6) implies $C \circ \mathcal{D}[\rho'(\{x/z\}\{\vec{y}/\vec{w}\}P')] \in [\![x : \{\overline{X}/X\}A]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})]}$.

By hypothesis, $Q' \in [\![x : \{\overline{X}/X\}\overline{A}]\!]_{\sigma[X \mapsto \psi^{m}_{A}(\emptyset^{\perp\perp})]}$, hence by Lemma 23(3) we obtain that R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}])]\ \ |x|\ \ Q\}$ is SN.

Therefore, $C \circ \mathcal{D}[\rho(\mathbf{corec}\ Y(z,\vec{w}); P'\ [x,\vec{y}])]\ \in \phi^{m+1}_{\{\overline{X}/X\}\overline{A}}(\emptyset^{\perp\perp})^{\perp}$.

Case: [Tdiscard]

$$\frac{}{\mathbf{discard}\ a \vdash_{\eta} a : \vee A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \vee A]\!] = S^{\perp}$, where

$$S = \{Q \mid \exists Q'.\ Q \equiv_{\mathsf{c}} \mathbf{affine}\ a; Q' \text{ and } Q' \in [\![a : \overline{A}]\!]\}.$$

Let $\mathcal{D} \in [\![\Gamma]\!]^{!}$ and $Q \in S$.

Then, $Q \equiv_{\mathsf{c}} \mathbf{affine}\ a; Q'$ and $Q' \in [\![a : \overline{A}]\!]$.

We have $\mathbf{cut}\ \{Q\ \ |a|\ \ \mathcal{D}[\mathbf{discard}\ a]\} \equiv_{\mathsf{c}} \mathbf{cut}\ \{Q\ \ |a|\ \ \mathbf{discard}\ a\}$.

We prove that (H) $\mathbf{cut}\ \{Q\ \ |a|\ \ \mathbf{discard}\ a\}$ is SN, by induction on $N(Q)$.

Suppose that $\mathbf{cut}\ \{Q\ \ |a|\ \ \mathbf{discard}\ a\} \to_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q.

Case: (ii) R is obtained by an interaction on cut session a.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\mathbf{cut}\ \{Q \ \ |a|\ \ \mathbf{discard}\ a\} \equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{affine}\ a; Q' \ \ |a|\ \ \mathbf{discard}\ a\} \rightarrow_{\mathsf{c}} \mathbf{0} = R$$

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |a|\ \ \mathbf{discard}\ a\}$ is SN.

Therefore, $\mathbf{discard}\ a \ \in [\![x : \vee A]\!]$, hence $\mathcal{D}[\mathbf{discard}\ a] \in [\![x : \vee A]\!]$ (Lemma 23(2)).

By Lemma 26, $\mathbf{discard}\ a \in \mathcal{L}[\![\vdash_\eta a : \vee A; \Gamma]\!]$.

Case: [Tuse]

$$\frac{P' \vdash_\eta \Delta', a : A; \Gamma}{\mathbf{use}\ a; P' \vdash_\eta \Delta', a : \vee A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \bot]\!] = S^\bot$, where

$$S = \{Q \mid \exists Q'.\ Q \equiv_{\mathsf{c}} \mathbf{affine}\ a; Q' \text{ and } Q' \in [\![a : \overline{A}]\!]\}.$$

Let $C \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then $Q \equiv_{\mathsf{c}} \mathbf{affine}\ a; Q'$, where $Q' \in [\![a : \overline{A}]\!]$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |a|\ \ C \circ \mathcal{D}[\mathbf{use}\ a; P']\}$ is SN, by induction on $N(Q) + N(C)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |a|\ \ C \circ \mathcal{D}[\mathbf{use}\ a; P']\} \rightarrow_{\mathsf{c}} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or C.

Case: (ii) R is obtained by an interaction on cut session x.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\begin{aligned}\mathbf{cut}\ \{Q \ \ |a|\ \ C \circ \mathcal{D}[\mathbf{use}\ a; P']\} \ &\equiv_{\mathsf{c}} \mathbf{cut}\ \{\mathbf{affine}\ a; Q' \ \ |a|\ \ C \circ \mathcal{D}[\mathbf{use}\ a; P']\} \\ &\rightarrow_{\mathsf{c}} (\mathbf{cut}\ \{Q' \ \ |a|\ \ C\}) \circ \mathcal{D}[P'] \ = R\end{aligned}$$

Applying i.h. to $P' \vdash_\eta \Delta', a : A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |a|\ \ C \circ \mathcal{D}[\mathbf{use}\ a; P']\}$ is SN.

Therefore, $C \circ \mathcal{D}[\mathbf{use}\ a; P'] \ \in [\![a : \vee A]\!]$.

By Lemma 26, $\mathbf{use}\ a; P' \in \mathcal{L}[\![\vdash_\eta \Delta', a : A; \Gamma]\!]$.

Case: [Trelease]

$$\frac{}{\mathbf{release}\ c \vdash_\eta c : \mathbf{U}_f\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![x : \mathbf{U}_f\ A]\!] = S^\perp$, where

$$S = \{Q \mid Q \equiv_\mathsf{c} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\}.$$

Let $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_\mathsf{c} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\}$ is SN, by induction on $N(Q)$.

Suppose that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\} \to_\mathsf{c} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H).

So let us consider case (ii). Then

$$\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\} \equiv_\mathsf{c} \mathcal{D}[\mathbf{cut}\ \{\mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!]) \ \ |c|\ \ \mathbf{release}\ c\}] \xrightarrow{*}_\mathsf{c} \mathcal{D}[\mathbf{0}] = R$$

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{D}[\mathbf{release}\ c]\}$ is SN.

Furthermore, $\mathbf{release}\ c$ is vacuously $[\![y : \wedge\overline{A}]\!]$-preserving, for any y.

Therefore, $\mathcal{D}[\mathbf{release}\ c] \in [\![x : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\mathbf{release}\ c \in \mathcal{L}[\![\vdash_\eta a : \mathbf{U}_f\ A; \Gamma]\!]$.

Case: [Ttake]

$$\frac{P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma}{\mathbf{take}\ c(a); P' \vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![c : \mathbf{U}_f\ A]\!] = S^\perp$, where

$$S = \{Q \mid Q \equiv_\mathsf{c} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\}.$$

Let $\mathcal{C} \in [\![\Delta']\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_\mathsf{c} \mathbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])$.

We prove that (H) $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{C} \circ \mathcal{D}[\mathbf{take}\ c(a); P']\}$ is SN, by induction on $N(Q)+N(\mathcal{C})$.

Suppose that $\mathbf{cut}\ \{Q \ \ |c|\ \ \mathcal{C} \circ \mathcal{D}[\mathbf{take}\ c(a); P']\} \to_\mathsf{c} R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q or $\mathcal{C}$.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H). So let us consider case (ii). Then

$$\begin{aligned}\textbf{cut}\ \{Q\ \ |c|\ \ C \circ \mathcal{D}[\textbf{take}\ c(a); P']\} &\equiv_c \textbf{cut}\ \{\textbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ C \circ \mathcal{D}[\textbf{take}\ c(a); P']\} \\ &\to_c \textbf{cut}\ \{\textbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ (\textbf{cut}\ \{Q'\ \ |a|\ \ C \circ \mathcal{D}[P']\})\} = R\end{aligned}$$

where $Q' \in [\![a : \wedge\overline{A}]\!]$.

By Def. 32, $[\![c : \mathbf{S}_f\ \overline{A}]\!] = S^{\perp\perp}$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, hence $\textbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!]) \in [\![c : \mathbf{S}_f\ \overline{A}]\!]$.

Applying i.h. to $P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that $\textbf{cut}\ \{Q\ \ |c|\ \ C \circ \mathcal{D}[\textbf{take}\ c(a); P']\}$ is SN.

Now, we prove that $C \circ \mathcal{D}[\textbf{take}\ c(a); P']$ is $[\![a : \wedge\overline{A}]\!]$-preserving, for any a. Let $R \in [\![a : \wedge\overline{A}]\!]$. Applying i.h. to $P' \vdash_\eta \Delta', a : \vee A, c : \mathbf{U}_e\ A; \Gamma$ we conclude that $\textbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\} \in [\![c : \mathbf{U}_e\ A]\!]$, which implies that $\textbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\} \in [\![c : \mathbf{U}_e\ A]\!]$ and hence $\textbf{cut}\ \{R\ \ |a|\ \ C \circ \mathcal{D}[P']\}$ is $[\![a : \wedge\overline{A}]\!]$-preserving.

Therefore, $C \circ \mathcal{D}[\textbf{take}\ c(a); P'] \in [\![c : \mathbf{U}_f\ A]\!]$.

By Lemma 26, $\textbf{take}\ c(a); P' \in \mathcal{L}[\![\vdash_\eta \Delta', c : \mathbf{U}_f\ A; \Gamma]\!]$.

Case: [Tput]

$$\frac{P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma \qquad P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma}{\textbf{put}\ c(a.P_1); P_2\ \vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma}$$

By Def. 32 and Lemma 22(5) we have $[\![c : \mathbf{U}_e\ A]\!] = S^{\perp}$, where

$$S = \{Q \mid Q \equiv_c \textbf{empty}\ c([\![a : \wedge\overline{A}]\!].\}).$$

Let $C_1 \in [\![\Delta_1]\!], C_2 \in [\![\Delta_2]\!]$ and $\mathcal{D} \in [\![\Gamma]\!]^!$ and $Q \in S$.

Then, $Q \equiv_c \textbf{empty}\ c([\![a : \wedge\overline{A}]\!].)$.

We prove that (H) $\textbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\textbf{put}\ c(a.P_1); P_2]\}$ is SN, by induction on $N(Q) + N(C_1) + N(C_2)$.

Suppose that $\textbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\textbf{put}\ c(a.P_1); P_2]\} \to_c R$. There are two cases to consider:

Case: (i) R is obtained by an internal reduction of either Q, C_1 or C_2.

Case: (ii) R is obtained by an interaction on cut session c.

Case (i) follows by inner inductive hypothesis (H).So let us consider case (ii). Then

$$\begin{aligned}&\textbf{cut}\ \{Q\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\textbf{put}\ c(a.P_1); P_2]\} \\ &\equiv_c \textbf{cut}\ \{\textbf{empty}\ c([\![a : \wedge A]\!].)\ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\textbf{put}\ c(a.P_1); P_2]\} \\ &\equiv_c \textbf{cut}\ \{\textbf{empty}\ c([\![a : \wedge A]\!].)\ \ |c|\ \ \textbf{put}\ c(a.C_1 \circ \mathcal{D}[P_1]); C_2 \circ \mathcal{D}[P_2]\} \\ &\to_c \textbf{cut}\ \{\textbf{cell}\ c(a.[\![a : \wedge\overline{A}]\!])\ \ |c|\ \ C_2 \circ \mathcal{D}[P_2]\} = R\ (*)\end{aligned}$$

I.h. applied to $P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma$ yields $C_1 \circ \mathcal{D}[P_1] \in [\![a : \wedge\overline{A}]\!]$, hence reduction step (*).

By Def. 32, $[\![c : \mathbf{S}_f\ \overline{A}]\!] = S^{\perp\perp}$.

By Lemma 22(4), $S \subseteq S^{\perp\perp}$, hence **cell** $c(a.[\![a : \wedge\overline{A}]\!]) \in [\![c : \mathbf{S}_f\ \overline{A}]\!]$.

Applying i.h. to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ yields R is SN.

In either case (i)-(ii), R is SN.

By applying Lemma 21(3) we conclude that **cut** $\{Q \ \ |c|\ \ C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]\}$ is SN.

Now, we prove that $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2]$ is $[\![a : \wedge\overline{A}]\!]$-preserving, for any a. Applying i.h. to $P_1 \vdash_\eta \Delta_1, a : \wedge\overline{A}; \Gamma$ we conclude that $C_1 \circ \mathcal{D}[P_1] \in [\![a : \wedge\overline{A}]\!]$. Applying i.h. to $P_2 \vdash_\eta \Delta_2, c : \mathbf{U}_f\ A; \Gamma$ we conclude that $C_2 \circ \mathcal{D}[P_2] \in [\![c : \mathbf{U}_f\ A]\!]$, which implies that $C_2 \circ \mathcal{D}[P_2]$ is $[\![a : \wedge\overline{A}]\!]$-preserving

Therefore, $C_1 \circ C_2 \circ \mathcal{D}[\mathbf{put}\ c(a.P_1); P_2] \in [\![c : \mathbf{U}_e\ A]\!]$.

By Lemma 26, $\mathbf{put}\ c(a.P_1); P_2 \in \mathcal{L}[\![\vdash_\eta \Delta_1, \Delta_2, c : \mathbf{U}_e\ A; \Gamma]\!]$.

□

2022 CLASS: A Logical Foundation for Typeful Programming with Shared State Pedro Rocha