Propositions As Sessions Taken Apart

Wen Kokke

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

The foundations of functional programming are built on the λ-calculus, a powerful model of computation whose canonicity is affirmed by its correspondence with intuitionistic logic. The foundations of concurrent computation are built on less firm ground. There is a wide variety of process calculi, but none enjoy the same canonicity as the λ-calculus, nor an exact correspondence with logic. Since its inception by Girard [1987], Classical Linear Logic has been believed to have some relation to concurrent computation, which has spawned a wealth of research in both logic and programming language theory.

This thesis continues the work towards a foundation for concurrent computation, starting from the propositions-as-sessions correspondence proposed by Wadler [2012]. Drawing on the work of Abramsky [1994], Bellin and Scott [1994], and Caires and Pfenning [2010], among others, Wadler proposed the process calculus CP, which has an exact correspondence with Classical Linear Logic. Drawing on the work of Honda [1993], Honda et al. [1998], and Gay and Vasconcelos [2010], among others, Wadler proposed the session-typed functional language GV, which provides a practical foundation for session-typed concurrency in functional languages. Finally, Wadler connects GV and CP by means of an operational correspondence, and, thereby, connects practice of session-typed concurrency to Classical Linear Logic.

This thesis provides an in-depth study of the propositions-as-sessions correspondence proposed by Wadler, highlighting its strengths and repairing a number of its shortcomings. Due to its adoption of commuting conversions, Wadler's CP is non-confluent, and its semantics are difficult to realise using the abstractions of concurrent computation it purports to model. I address this shortcoming by removing the commuting conversions, and showing that the resulting process calculus is well-behaved and retains its connection to linear logic. Due to its lack of a parallel composition operator, Wadler's CP is ill-suited to an analysis using standard concurrency theory. Wadler's GV lacks an operational semantics and accounts of polymorphism and replication, which were provided by Lindley and Morris [2014; 2015; 2016b]. Unfortunately, Lindley and Morris' GV suffers from similar problems as CP with regard to its treatment of parallel composition, which complicates its metatheory. I address this shortcoming in both systems by adding a parallel composition operator and showing that the resulting calculus is well-behaved and correspond to a hypersequential variant of Classical Linear Logic. Finally, I study Priority CP, a variant of Wadler's propositions-as-sessions correspondence proposed by Dardha and Gay [2018], which increases its expressivity at the cost of its compositionality and correspondence to logic. As with CP, I remove the commuting conversions from Priority CP, and complete the correspondence with Priority GV, which is the analogous variant of GV.

Lay Summary

In recent decades, programming languages have widely adopted *data type checking*, which are a mechanism that helps programmers write safe and secure programs by ruling out data errors. An example of a *data error* would be if the program that manages bank accounts handled the account balance as text rather than as a number. If my balance is £50 and I deposit £20, I would expect my balance to become £70, not the text '5020'.

When a programming language is designed with care, it is possible to automatically determine if a program is free from data errors by checking the types of ways in which some data is handled (data types) against each other (e.g. the account balance is handled as a number here but as text there) or against the programmer's stated intention (e.g. the account balance is handled as text, but the programmer stated that it should always be handled as a number).

With the rise of the internet and the end of Moore's Law, *concurrent programs* are becoming increasingly important. A concurrent program consists of multiple processes that run at the same time and may share resources or communicate by passing messages. Examples include your browser communicating with a server, an ATM communicating with a bank, and the thousands of graphics processors involved in rendering your favourite game.

Concurrent programs are vulnerable to erroneous behaviours that cannot be prevented by data types. For an example of such an error, let us consider what would happen if the program that manages bank accounts handled each deposit or withdrawal in three steps: (1) read the current balance, (2) calculate the updated balance by adding the amount deposited or subtracting the amount withdrawn, and (3) write the updated account balance. If the account balance for the joint account that I share with my partner is £100, I use one ATM to deposit £60, and my partner uses another ATM to withdraw £50, then we would expect our balance to become £110. If the two interactions happen in sequence, this is what happens. However, each ATM reads and writes the account balance separately, and these reads and writes may be interleaved in any order:

- If my ATM reads the initial balance of £100, then my partner withdraws £50, and then my ATM writes the updated balance of £100 plus the deposited £60, the final balance becomes £160.
- If my partner's ATM reads the initial balance of £100, then I deposit £60, and then my partner's ATM writes the updated balance of £100 minus the withdrawn £50, the final balance becomes £50.

This kind of error is known as a *race condition*. Race conditions are difficult to diagnose since, while the program is fundamentally incorrect,

the erroneous behaviour might not happen most of the time. After all, how often do my partner and I use our joint account at exactly the same time? So if I, unaware of my partner's simultaneous withdrawal, call my bank to complain that "I deposited £60, but my balance went down!", an engineer might test the bank's program by depositing £60, see the account balance go up, and conclude that everything is working as intended.

Behavioural type checking is a mechanism which aims to rule out race conditions and other such erroneous behaviours. Whereas data types describe the way that a program must always handle some data (e.g. the account balance must always be handled as a number), behavioural types describe the evolving way a program must interact with its environment over time (e.g. usually, the bank's program should accept any request for a transaction, but while a transaction is ongoing, it should not accept requests for other transactions on the same account).

Most mainstream programming languages do not support behavioural types and, as such, it is not generally possible to automatically determine if a program is free from behavioural errors. In practice, writing correct concurrent programs comes down to the programmer correctly reasoning about how multiple processes might interact and correctly diagnosing and repairing errors, both of which are incredibly difficult tasks.

One reason that behavioural types have not yet seen widespread mainstream adoption is that the mathematical foundation underlying them is several decades younger and less well understood than the mathematical foundation underlying data types.

In this thesis, I investigate several proposed theories that could become the mathematical foundation underlying behavioural types for message-passing communication in concurrent programs. I identify several shortcomings in the proposed theories and, on the basis of my findings, propose a new mathematical foundation for behavioural types. I develop the theory necessary to integrate the proposed behavioural types into existing programming languages and describe a proof of concept implementation.

Acknowledgements

I am grateful to my supervisor, Philip Wadler, and my secondary supervisors, Sam Lindley and J. Garrett Morris, for their support throughout my Ph.D.

I am grateful to my examiners, James Cheney and Peter Thiemann, for accepting to act as my examiners and for their feedback on this thesis.

I am grateful to the proverbial village—to all past members of the ABCD research group, to all past and present members of the Scottish Programming Languages Community, and especially to Robert Atkey, Ornela Dardha, Simon Fowler, and James McKinna.

I am grateful to Fabrizio Montesi and Marco Peressotti for the kindness they showed me upon discovering our parallel developments, and for exposing me to concurrency theory.

I am grateful to Jonathan MacBride and the lovely folks at the Staff Pride Network for making the University of Edinburgh a kinder and safer place.

I am grateful to Sabine Weber for founding and organising the Informatics Open Art Space with me, to all the wonderful folks who joined us every week, and to Tekevwe Kwakpovwe and Marine Demarty for continuing to organise the Open Art Space.

I am grateful to Andrew McLeod and Mihaela Dragomir for organising my first-year trip to the Firbush Outdoor Centre, to the Firbush staff for starting my obsession with windsurfing and sailing, and to the many, many folks who helped me organise and attend nearly a dozen trips to Firbush between 2016 and 2020.

I am grateful to the many, many organisers of CDT Pizza, and especially to Kate McCurdy, Aidan Marnane, Sigrid Passano Hellan, and Siobhán Carroll for organising the 2019 series with me, to the EPSRC Centre for Doctoral Training in Data Science for funding CDT Pizza, and to the CDT in Data Science community for welcoming outsiders with an interest in data science and free pizza.

I am grateful to my mother, Marieke Helsen, and my secondary parents, Odile Jansen and Louis Kolkman, for their support throughout my Ph.D.

I am grateful to my partners, pals, and pets, who have greatly supported me throughout my Ph.D., and especially to April Gonçalves and Dora Gonçalves-Kokke, Grae Rose, Inari Listenmaa, and Zaza Helsen.

The EPSRC Centre for Doctoral Training in Pervasive Parallelism (EP/L01503X/1) supported my studies at the University of Edinburgh.

The Cost Action CA15123 (EUTypes) supported my visit to Fabrizio Montesi and Marco Peressotti at the University of Southern Denmark.

The SIGPLAN Professional Activities Committee (PAC) supported my attendance of the 23rd and 24th International Conference on Functional Programming (ICFP'18 and ICFP'19) and the 46th Symposium on Principles of Programming Languages (POPL'19).

Table of Contents

Chapter 1

Introduction

The foundations of functional programming are built on the λ-calculus, a powerful Model of Computation whose canonicity is affirmed by its correspondence with intuitionistic logic. The foundations of concurrent computation are built on less firm ground. There is a wide variety of process calculi, but none enjoy the same canonicity as the λ-calculus, nor an exact correspondence with logic.

Since its inception by Girard [1987], Classical Linear Logic (CLL) has been believed to have some relation to concurrent computation, which has spawned a wealth of research in both logic and programming language theory.

This thesis continues the work towards a foundation for concurrent computation, starting from the propositions-as-sessions correspondence proposed by Wadler [2012].

- Drawing on the work of Abramsky [1994], Bellin and Scott [1994], and Caires and Pfenning [2010], among others, Wadler proposed the process calculus Classical Processes (CP), which has an exact correspondence with Classical Linear Logic.
- Drawing on the work of Honda [1993], Honda et al. [1998], and Gay and Vasconcelos [2010], among others, Wadler proposed the session-typed functional language Good Variation (GV), which provides a practical foundation for session-typed concurrency in functional languages.
- Finally, Wadler connects GV and CP by means of an operational correspondence, and, thereby, connects practice of session-typed concurrency to Classical Linear Logic.

The names CP and GV are unstated homages to the authors of the work that inspired them: Classical Processes was based on πDILL by Caires and Pfenning [2010; later Caires et al., 2016]; and Good Variation was based on LAST by Gay and Vasconcelos [2010; the name LAST was given by Lindley and Morris, 2015].

Let us start by discussing Wadler's propositions-as-sessions correspondence at a glance. The paper contains two separate correspondences:

1. CP demonstrates that the original and most well-known sequent calculus for full CLL, exactly as presented by Girard [1987, pp. 22 and 26-27], can be interpreted as something that syntactically resembles a process calculus. I intentionally avoid calling CP a process calculus—whether or not it *is* is debatable—but it can hardly be denied that CP *looks* like a process calculus.

2. GV, together with the translations from GV to CP [Wadler, 2012, § 3.1; Lindley and Morris, 2015, § 3.2] and from CP to GV [Lindley and Morris, 2015, § 3.1], demonstrates an operational correspondence between CLL and a session-typed concurrent λ-calculus.

Only the first—the correspondence between CLL and CP—is one that I would characterize as a Curry-Howard correspondence:

- It relates a logic and a typed Model of Computation.
- It is complete. Propositions correspond to types, proofs correspond to processes, and cut elimination corresponds to computation.[1]
- It is exact. Going from the logic to the Model of Computation, and vice versa, requires no hard work. It is simply a change of notation.

The correspondence is not a *profound coincidence*. CP was *intentionally constructed* to correspond to CLL. Moreover, its inspiration, πDILL, was intentionally constructed to correspond to DILL. A constructed correspondence undermines what is arguably the most important role of Curry-Howard correspondences. It does nothing to reassure us that either system is profound, in the sense of a correspondence between two systems that were independently devised.

One could argue that these correspondences relate the π-calculus to linear logic—in which case they would be profound—but I believe CP and πDILL are too far removed from the π-calculus for this to be true.

Having rejected these correspondences as profound coincidences, let us take a different view: CP *anticipates* a Curry-Howard correspondence. The typed λ-calculus, the most popular typed Model of Computation, corresponds exactly to intuitionistic logic. Linear logic has been said to be relevant to concurrent computation since its inception. So why not cut out the middle-man of independent discovery and directly employ linear logic as a typed Model of Concurrent Computation (MoCC)?

[1] The correspondence between CP and CLL is actually incomplete, in the sense that the structural congruence of CP does not correspond to anything in CLL. The equations of the structural congruence are merely valid rewrite rules for proofs. This is easily missed: λ-calculus does not have a structural congruence, and as such the list of correspondences for CP looks as complete as the list for the Curry-Howard correspondence.

To evaluate whether or not CP is successful under this particular view, we must examine what we want from a typed MoCC:

- What is the intended purpose for our typed MoCC?
- Do we want our typed MoCC to be as expressive as the π-calculus?
- What properties do we want our typed MoCC to have?

To me, the purpose of a type system is to rule out erroneous programs. The price of a type system is *necessarily* a reduction in expressivity—for the silly reason that we can no longer express erroneous programs, and for the serious reason that the exact boundary between erroneous and non-erroneous is difficult to capture.

- I do not want to evaluate programs with data-type errors such as `"hello" + true`. The price paid is more difficulty with code reuse [see, e.g. McBride, 2010, Dagand and McBride, 2014, Stump, 2017]
- I do not want to evaluate programs that loop indefinitely, such as `x = x`. The price paid is more difficulty with recursive algorithms due to the undecidability of the halting problem [Davis, 1952, Kleene, 1952, Davis, 1958].
- I do not want to evaluate programs that deadlock. The price paid is more difficulty with certain concurrent communication structures.

It seems unreasonable to expect our typed MoCC to be as expressive as, say, the π-calculus, simply because the π-calculus includes all of the above erroneous behaviors. However, this raises an important question: What behaviors are erroneous?

Let us consider Mazza [2018], who paints a pessimistic picture of the "proofs as processes" program. They argue that neither CLL nor DiLL[2] [Differential Linear Logic, Ehrhard and Regnier, 2006, Ehrhard, 2018] can satisfactorily encode even an elementary process calculus, as neither can express *confusion*—a kind of non-local non-determinism in which two communications mutually exclude each other. This raises an immediate question: *Do we want confusion?* The question alone elicits an immediate and visceral "No!" from the audience, but once our gut reaction subsides, we should evaluate the question formally.

Mazza [2018] demonstrates that confusion is present in even the most elementary MoCCs, and that confusion is preserved by any encoding that preserves the degree of distribution—i.e. any encoding that is a homomorphism on parallel composition. Therefore, there exists a class of processes for which we must choose (a) to allow confusion, or (b) to accept a reduction in the degree of distribution. Networking algorithms commonly allow for confusion—e.g. *leader election* only requires that

[2]Note that DiLL refers to Differentiable Linear Logic, whereas DILL refers to Dual Intuitionist Linear Logic. The only difference in writing is the capitalization of the "i". Presumably, both are pronounced like the herb.

some leader is elected. Therefore, I do not feel comfortable rejecting confusion outright. However, accepting confusion means rejecting strong normalization and the Church-Rosser property. To me, those properties are synonymous with safety, and I cannot reject them outright either.

We may want to allow our programming languages to write programs with local non-determinism or confusion, but we should never allow the programmer to write such a program *by accident*. This seems to require that we capture programs with these properties using some modality. CLL captures non-linearity using the exponentials. DiLL captures failure and local non-determinism using the co-exponentials. I choose not to commit to the wholesale acceptance or rejection of non-determinism and confusion. Rather, my stance is that all is not lost if our typed MoCC does not support confusion. We have simply taken the first step on our journey of trying to claw back the steep price we paid for types.

Dependently-typed λ-calculus recovers a lot of the expressivity of the λ-calculus by allowing output data to depend on input data. For instance, it can express any recursive function whose termination can be proved within the type system of the calculus itself. It stands to reason that a typed process calculus could likewise recover expressivity by allowing the present communication structure of a program to depend on the past communication structure.

I would argue that to build a dependently-typed process calculus, we need the solid foundation of a simply-typed process calculus. I believe—and intend to convince you in this thesis—that the simply-typed fragment of CP is not yet the solid foundation we are looking for.

- The correspondence between CP and process calculus is lacking, because CP does not easily admit a behavioural theory, e.g. a labelled-transition system and a behavioural equivalence.
- The correspondence between CP and CLL is lacking, because the sequent calculus for CLL is not the canonical proof theory. Proof nets are the canonical representation of CLL proofs! This is an issue with CP as a typed MoCC, because the sequent calculus has a lower degree of distribution than the proof nets.
- CLL itself is lacking as a foundation for a typed MoCC. It does not describe non-determinism, unlike DiLL [Ehrhard, 2018]. It describes parallelism as independence, and its dual as dependence, but cannot describe sequential dependence, unlike Pomset Logic [Retoré, 1997] and BV [Guglielmi, 2007].

The first assignment Philip Wadler gave me as my Ph.D. supervisor was to add support for non-determinism to CP. I did, albeit poorly [Kokke, 2017]. However, while doing so, I discovered more ways in which CP is lacking as a process calculus. Since CP does not have a standalone process construct for parallel composition, the extension I

introduced has *three different* process constructs that contain a parallel composition (two inherited from CP, and a third that permits non-determinism). Parallel composition is the most crucial connective in a process calculus. Not having it as a standalone process construct in CP is untenable. Consequently, I pivoted to restructuring CP to include parallel composition as a standalone process construct and tighten its correspondence with the π-calculus, while retaining its tight correspondence with CLL. This work is the principal focus of this thesis and is presented in Chapter 3 and Chapter 4. At times, I will allude to the second and third points, and arguably some of the work on priorities in Chapter 5 applies to the third point. However, I consider them out of scope for this thesis, as I believe that each could correspond to several theses in their own right.

The result should, at least at its core, be a conservative extension of CP. There are three main reasons for wanting this.

1. It retains some correspondence with Classical Linear Logic, even if the correspondence is no longer as obvious as in Wadler's case.
2. It ensures that any previous work on CP, such as Wadler's work on polymorphism and unrestricted usage and Lindley and Morris' work on fixed points, carries over without too much trouble.
3. It eases the future integration of the features offered by DiLL, Pomset Logic, and BV, since these logics are all extensions of CLL.

As I mentioned, my first assignment was to add support for non-determinism to CP, which I did, though in a manner I have always found unsatisfactory [Kokke, 2017]. Later, Qian et al. [2021] found the exact structure I was looking for. In hindsight, the answer was there all along, in logic! DiLL extends CLL with co-exponentials, which make cut elimination non-deterministic [Ehrhard, 2005, 2018]. Qian, Kavvos, and Birkedal's work, which extends CP with DiLL's co-exponentials, uses the work on hypersequents presented in Chapter 3.

What role does GV play in the propositions-as-sessions correspondence? CP demonstrates a correspondence between a process calculus and CLL—a correspondence so exact that it requires no proof—but CP is unsatisfactory as a foundation for a programming language for one simple reason: CP does not have functions. In CLL, everything is defined in terms of duality. This matches well with session types and channel-based communication, where two processes act in dual ways, but does not match well with functions. CLL's implication is *classical*, and does not lend itself to an interpretation as a function type. A significant portion of programming language theory is built on the foundations of functions and the λ-calculus. Hence, adopting CP as the foundation for concurrent programming means parting ways with a significant body of work.

In essence, GV extends CP with functions. Extending CP with a function type requires extending CLL with an intuitionistic implication, which is

more complicated than simply adding a connective, as CLL's sequents are fundamentally classical. There are some approaches to mixing classical and intuitionistic connectives in the literature on logic, such as Display Logic [Belnap, 1982]. However, none of these approaches have the widespread acceptance of CLL, nor are they easily amenable to a term calculus. Instead, Wadler [2012] approaches GV from the perspective of concurrent λ-calculus, and adapts Gay and Vasconcelos' LAST [Gay and Vasconcelos, 2010] to fit a correspondence with CP. From a theoretical viewpoint, GV embeds an axiomatisation of CP into the linear λ-calculus using constants. From a practical viewpoint, GV resembles the way in which concurrency is exposed to the user in real-world programming languages, such as the POSIX Threads API for the C programming language. GV offers the user an API—a collection of constant functions which create threads and channels, send messages, etc—but concrete threads, parallel composition, and name restriction are not representable in the static language. The user cannot write a program that *is* the parallel composition of two processes, only a program which creates that configuration. This makes it very easy to implement GV's safe concurrency primitives as a library in existing programming languages, on top of the language's unsafe concurrency primitives, and doubly so if the languages already supports some form of linear types [see, e.g. Lindley and Morris, 2016a, Kokke, 2019, Kokke and Dardha, 2021b].

To put it plainly, CP is a theoretical tool for studying foundational well-behaved concurrent systems, but GV is what you actually implement.

What role does the correspondence between CP and GV play? Wadler [2014, p. 385] writes that the correspondence formalises, for the first time, "a tight connection between a standard presentation of session types and linear logic", which formally confirms the previously assumed connection. However, this leaves us with a question. Now that the connection is formally confirmed, what is the purpose of continuing to maintain the correspondence as we make changes to CP and GV? I am motivated to do so for two reasons.

Firstly, an exact correspondence, especially one that preserves parallel composition, reassures us that GV is correct, in that the communication primitives of GV capture the same communication structures as CP. Unfortunately, constructing an exact correspondence requires some amount of tedium, as one must show that CP's communication channels can correctly emulate GV's functions and data structures. This is well-established and an area of GV significantly less likely to contain mistakes—or, at least, mistakes that are easily caught by a translation into process calculus. Consequently, any part of GV that does not interact with concurrency should be omitted from the correspondence.[3] To further

[3]Wadler's GV only has the concurrency primitives, product types—which are used to type the receive primitive—and the unit type—which is used to type the primitive that

simplify the correspondence, the translation from GV to CP is commonly factored into two translations: a translation from GV into fine-grain GV, which removes higher-order control flow; and a translation from fine-grain GV into CP.[4]

Secondly, the translations between CP and GV are helpful in guiding you to the correct typing and implementation of new concurrency primitives. For instance, Wadler [2014, p. 409] considers several alternative designs, but remarks that "these designs are difficult to translate into CP, which suggests they may suffer from deadlock." Indeed, the suggested alternatives *do* suffer from deadlock. Hence, by following the translation, Wadler correctly chose to abandon alternative concurrency primitives, even though they appear more principled.

1.1 Contributions

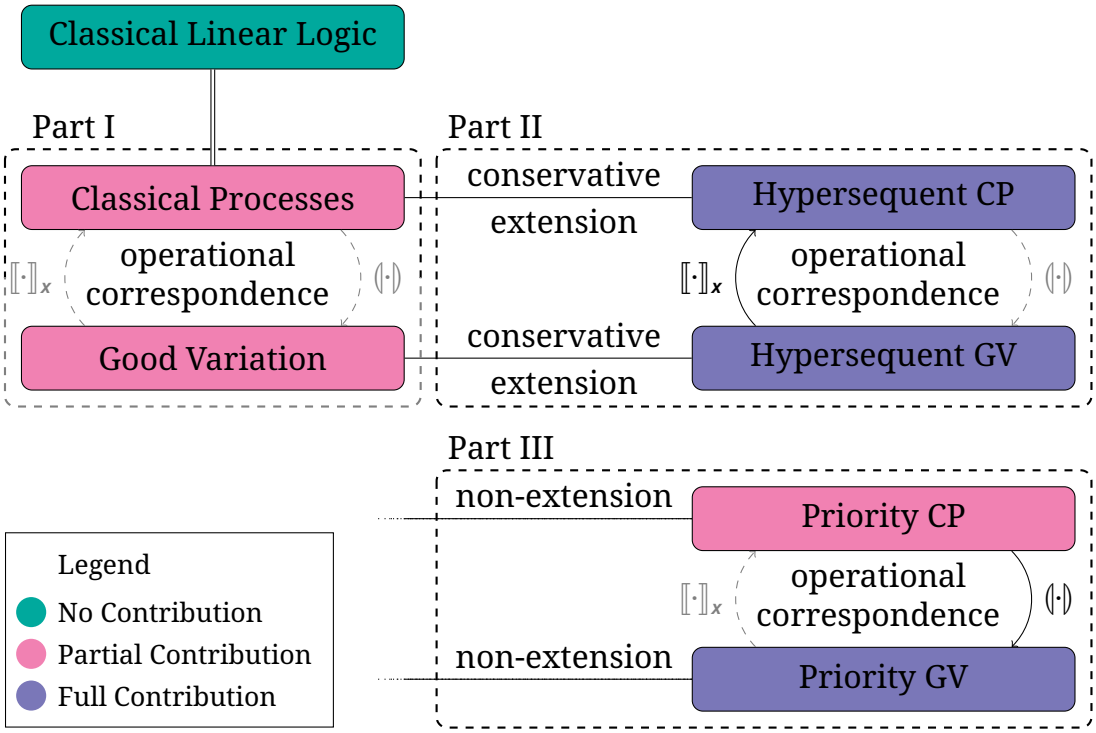
The contributions of this thesis are centred around Wadler's propositions-as-sessions correspondence, and are divided into four parts. Part I is concerned with Wadler's propositions-as-sessions correspondence. It discusses CP and its metatheory. Part II is concerned with the hypersequential variants. It discusses Hypersequent CP (HCP), Hypersequent GV (HGV), their metatheory and operational correspondence, as well as their relation to CP and GV. Part III is concerned with the priority-based variants. It discusses Priority CP (PCP), Priority GV (PGV), their metatheory and operational correspondence, as well as their relation to CP and GV. Part IV is concerned with implementations of Good Variation and Priority GV in Linear Haskell.

The following visualisation gives an overview of the landscape of the formal systems discussed in this thesis, and the level of my contributions.

---

closes a channel. Lindley and Morris [2014] add polymorphism and replicated sessions to GV, which correspond to CP's polymorphism and exponentials. Lindley and Morris [2015] add sum types to GV, which are used to replace the concurrency primitives for choice by simply sending and receiving a value of a sum type.

[4]The clearest presentation of this decomposed translation can be found in Fowler et al. [2023], which will be presented in Chapter 4, but its origins can be traced back all the way to Lindley and Morris [2014], where fine-grain GV is referred to as HGVπ. (Note that the "H" in HGVπ stands for "Harmonious", not "Hypersequent" as in Chapter 4.).



The sole *green* box signifies that I do not claim to have made any serious contribution to Classical Linear Logic. The *pink* boxes signify that, while Classical Processes, Good Variation, and Priority CP were not originally developed by me, I have since contributed to these theories in one way or another. The *periwinkle* boxes signify that Hypersequent CP, Hypersequent GV, and Priority GV were developed by me and colleagues.

The arrows represent the translations and operational correspondences between the various systems. The arrows are labelled by the conventional names of these translations. By convention, any translation from a variant of GV to a variant of CP is denoted by double square brackets with an endpoint name as a subscript, i.e. $[\![\cdot]\!]_x$, and any translation from a variant of CP to a variant of GV is denoted by double parentheses, i.e. $(\!|\cdot|\!)$. The solid arrows—the translation from Hypersequent GV to Hypersequent CP and the translation from Priority CP to Priority GV—and were developed by me and colleagues. The translations corresponding to the dashed and greyed-out arrows are not discussed in this thesis.

The solid lines represent the extension and non-extension results for the various systems with respect to CP and GV, and were developed by me and colleagues.

The dashed boxes—labelled Part I, Part II, and Part III—roughly group the systems by the part of this thesis in which they are discussed.

In the remainder of this section, I give a detailed account of the contributions made in this thesis.

Part I: Classical Processes Revisited

Part I consists of Chapter 2, which revisits Classical Processes. The principal focus is to present, in detail, a variant of CP without commuting conversions, which break confluence in Wadler's CP, and show that this variant has adequate canonical forms.

Chapter 2 contains the following contributions by me:

- I drop the commuting conversions from the reduction semantics, which cause Wadler's CP to be non-confluent, and prove that *progress* (Proposition 2.32) continues to hold, albeit with a different canonical form (Definition 2.30).

  The reduction semantics without commuting conversions first appeared in Kokke [2017].

- I formalise the relation between CP with and without the commuting conversions. I prove that (1) any process in canonical form can reduce to a process in Wadler's canonical form using only commuting conversions (Proposition 2.57); and (2) any reduction with commuting conversions is equivalent to a reduction without commuting conversions followed by a reduction using only commuting conversions (Proposition 2.58).

  To the best of my knowledge, this is the first publication in which these properties are explicitly stated and proven for CP, though they are hinted at by Lindley and Morris [2016b].

- I formalise the notion of *dependency graph* (Definition 2.35) and *deadlock* (Definition 2.38) for CP, prove that CP processes are deadlock-free (Proposition 2.39), and prove that my canonical forms are adequate, by showing that processes in canonical form are blocked on free endpoints (Corollary 2.47).

  To the best of my knowledge, this is the first publication which explicitly defines dependency and deadlock for CP, though it is related to Lindley and Morris' definitions of dependency and deadlock for GV, and to PCP's priority constraints [Dardha and Gay, 2018]. The proof of deadlock freedom is adapted from a similar proof of deadlock freedom for GV by Lindley and Morris [2015]. My characterisation of "blocked on free endpoints" is stronger than Lindley and Morris' characterisation, which corresponds to my Proposition 2.34.

- I formalise the notion of *connection graph* for CP (Definition 2.49), prove that CP's connection graphs are trees (Proposition 2.50), prove that every CP process can be rewritten into *right-branching form* (Proposition 2.51), and prove that connection graphs are canonical representatives of processes up to a restricted structural congruence (Proposition 2.53).

I have informally referred to the connection graph as the *process structure* or *communication structure* [Kokke and Dardha, 2021a,b]. The definition and corresponding theorems are adapted from work by Simon Fowler for Hypersequent GV [Fowler et al., 2023, Definition 3.9], who formulate the *abstract process structure* of the free names in a process in terms of its typing environment and a set of co-names.

Good Variation is introduced in Chapter 4, together with the proof that HGV is a conservative extension of GV, and the variant of GV with cuts is described in § 4.3.1 and § 4.3.2.

I believe that my changes to CP (in Chapter 2), are more important than my changes to GV (in § 4.3.1). Unlike GV, CP has an exact correspondence with CLL. Furthermore, my changes affect its reduction semantics in a fundamental way, and these changes are used in HCP and PCP. My changes to GV, on the other hand, are immediately superseded by the introduction of hypersequents in HGV in Chapter 4.

If you require a detailed discussion of GV without hypersequents, I recommend Fowler's Ph.D. thesis, *Typed Concurrent Functional Programming with Channels, Actors, and Sessions* [Fowler, 2019, Chapter 3].

Part II: Taking Linear Logic Apart

Part II consists of Chapter 3 and Chapter 4. Chapter 3 introduces Hypersequent CP, which is a session-typed process calculus based on CP with a tighter correspondence to the π-calculus. Chapter 4 introduces Hypersequent GV, which is the corresponding session-typed concurrent λ-calculus.

Chapter 3 contains the following contributions by me:

- I introduce Hypersequent CP with its typing rules and reduction semantics, and prove *preservation* (Proposition 3.30) and *progress* (Proposition 3.35).

  The main innovation—HCP's type system—was developed independently by myself and by Fabrizio Montesi & Marco Peressotti[5]. The reduction semantics were primarily developed by me, and the label-transition semantics were primarily developed by Fabrizio Montesi & Marco Peressotti.

[5]Our initial developments had minor errors. My initial development permitted hyper-environments to occur in all logical rules, which breaks progress. Progress can be repaired for the multiplicative rules, by using delayed actions [as in Kokke et al., 2019a], but not for the additive rules. These errors made it into the first publication [Kokke et al., 2019b]. Fabrizio Montesi & Marco Peressotti's initial development [Montesi and Peressotti, 2018] typed the terminated process using the empty environment, as opposed to the empty hyper-environment, which admits MIX_0.

The similarities between my early work on Hypersequent CP and Hypersequent Calculus [Avron, 1991] were noted by Simon Castellan at an ABCD meeting.[6]

- I adapt the notions of *dependency graph* (Definition 3.36) and *deadlock* (Definition 3.39) for HCP, prove that HCP processes are deadlock-free (Corollary 3.42), and prove that my canonical forms are adequate (Corollary 3.50).
- I adapt the notion of *connection graph* for HCP (Definition 3.51), and prove that HCP's connection graphs are forests (Proposition 3.52), and that every HCP process can be rewritten into *right-branching forest form* (Proposition 3.57).
- I formalise disentanglement from Hypersequent CP processes to multisets of CP processes, and prove that disentanglement preserves typing, structural congruence, and reduction.

  Disentanglement was first described by Kokke et al. [2019b] as a rewrite relation that preserves provability, but the presentation in this thesis is novel. To the best of my knowledge, this is the first publication which shows that disentanglement preserves structural congruence and reduction.
- I introduce a label-transition semantics for HCP and prove *harmony* between the reduction and label-transition semantics (Proposition 3.93).

  The label-transition semantics is a minor variation of the label-transition semantics by Montesi and Peressotti [2018]. To the best of my knowledge, this is the first publication that proves harmony for a reduction and label-transition semantics for HCP.
- I introduce a variant of HCP which uses an alternative *exceptional* semantics for the additive units, with both reduction and label-transition semantics, and extend the proofs of preservation, progress, and harmony.
- I introduce a variant of HCP which uses an alternative *synchronous* semantics for link, based on identity expansion, which, I claim, continues to work in the presence of polymorphism, and discuss the consequent changes in canonical forms, progress, and adequacy.
- I introduce a variant of HCP which uses a combination of variant types and focusing to further tighten the correspondence to the π-calculus by permitting prefixing and (guarded) summation as

[6] My talk at the ABCD meeting on Monday, December 18, 2017, was titled *Taking Apart Classical Processes*. Oddly, it appears to refer to the calculus as Hypersequent CP, which implies that I used the word "hypersequent" *before* I was aware of Avron's work on Hypersequent Calculus.

syntactic constructs, extend the proof of preservation, and sketch a proof of operational correspondence between HCP and the variant.

Chapter 4 consists primarily of the paper *Separating Sessions Smoothly* by Fowler et al. [2023], written in collaboration with Simon Fowler, Ornela Dardha, Sam Lindley, and J. Garrett Morris.

- We introduce Hypersequent GV with its typing rules and reduction semantics, and prove *preservation* (Theorem I.3.3), the *tree-structure* of connections in configurations (Theorem I.3.14), *global progress* (Theorem I.3.20), the *diamond property* (Theorem I.3.21), and *termination* (Theorem I.3.22).
- We define a translation from GV to HGV (Theorem I.4.3)
- We define a translation from HGV to GV (Corollary I.4.7).
- We prove an operational correspondence between HGV and HCP. We define fine-grain call-by-value HGV (HGV$*$), define translations from HGV to HGV$*$ and from HGV$*$ to HCP, and prove that the latter translation preserves types (Lemma I.5.9) and is a sound and complete operational correspondence (Theorem I.5.11).

  To the best of my knowledge, this is the first publication of a sound and complete correspondence between any variant of CP and GV.
- We define a translation from HCP to HGV.

I co-developed Hypersequent GV and was primarily responsible for the translations and operational correspondence between HGV and HCP.

The remainder of the chapter contains the following contributions by me:

- I introduce a variant of GV that uses cuts rather than lock typing. Consequently, its structural congruence preserves types, which significantly simplifies its metatheory, and I argue that under lock types, the parallel composition construct is equivalent to a cut.

Part III: Prioritise the Best Variation

Part III consists of Chapter 5, which revisits Priority CP, a variant of CP with priorities whose typing rules permit benign cyclic process configurations, and introduces Priority GV, which is the corresponding session-typed concurrent λ-calculus.

Chapter 5 consists primarily of the paper *Prioritise The Best Variation* by Kokke and Dardha [2021a], written in collaboration with Ornela Dardha.

- We drop the commuting conversions from the reduction semantics of Priority CP, which cause Dardha and Gay's CP to be non-confluent, and prove that *progress* (Theorem II.4.4) continues to hold, albeit with a different canonical form.

These changes are similar to the changes made to CP in Part I.

- We introduce Priority GV with its typing rules and reduction semantics, and prove *preservation* (referred to as *subject reduction*, Theorem II.3.5) and *global progress* (Theorem II.3.14).
- We define a translation from PCP to PGV, and prove that the translation preserves types (Theorem II.4.6) and is a sound and complete operational correspondence (Theorems II.4.7 and II.4.10).

I co-developed Priority GV and was primarily responsible for the initial draft of its theory and metatheory. I adapted the changes to the reduction semantics of Priority CP from my previous work on CP.

The remainder of the chapter contains the following contributions by me:

- I prove that PCP is not an extension of CP (Counterexample 5.1).
- I define priority inference for PCP, and prove that priority inference is sound and complete with respect to typing for PCP (Proposition 5.5 and Proposition 5.7).

Part IV: Deadlock-Free Session Types in Linear Haskell

Part IV consists of Chapter 6, which describes a library in Linear Haskell which implements session-typed channels based on GV and Priority GV.

Chapter 6 consists primarily of the paper *Deadlock-Free Session Types in Linear Haskell* by Kokke and Dardha [2021b], written in collaboration with Ornela Dardha.

- We implement session-typed channels in Linear Haskell and argue that a restricted interface to those channels corresponds to GV's channels and therefore enjoys GV's safety guarantees.
- We define a refined variant of Priority GV's type system, which gives exact rather than approximate priority bounds.
- We define the monadic reflection of Priority GV's priority-based type system into a graded linear monad, where the grading is pairs of lower and upper priority bounds.
- We implement priority-based session-typed channels in Linear Haskell and argue that those channels correspond to PGV's channels and therefore enjoys PGV's safety guarantees.
- We compare our Haskell library to existing Haskell libraries for session-typed channels, extending the comparison by Orchard and Yoshida [2017].

I implemented the Haskell library, developed the refined version of Priority GV's type system and its monadic reflection, wrote the initial

draft of the paper, and co-authored the comparison with existing Haskell libraries for session-typed channels.

1.2 Conventions

In this section, I introduce several conventions that I intend use throughout this thesis.

Barendregt's Convention

My proofs use Barendregt's Convention [1985]. If a term occurs in a proof, then all bound variables in that term are chosen to be different from each other and all the free variables. This applies equally to variables in λ-calculus terms and endpoint names in process calculus processes. In each case, the proofs can be made more formal by permitting and applying α-conversion where necessary, or by using a nameless term syntax such as deBruijn indices.

Pattern Matching

I use the phrase "is of the form" to imply pattern matching, where any unbound meta variable on the right hand side of that phrase is implicitly existentially quantified, and the entire phrase is converted into an equality, e.g. the phrase "P is of the form Q ∥ R" should be interpreted as "there exist some Q and R such that P = Q ∥ R".

Syntax Highlighting

Except where noted, each chapter is focused primarily on one system.

The terms of that system are printed in red, its types are printed in blue, and its annotation—where applicable—are printed in yellow, and all three are rendered in a sans-serif font.

To save on accessible colour combinations, the terms, types, and annotations of *any other system* are printed in *pink*, *green*, and *periwinkle*, respectively, and all three are rendered in an italicised font with serif. The relations of other system, such as typing and reduction, are marked by a subscript.

Let us discuss CP's syntax highlighting as an example. In Chapter 2, which focuses on CP, its processes and types of are printed in red and blue, respectively, and are rendered in a sans-serif font. However, in Chapter 3, which focuses on HCP, CP's processes and types are rendered in *pink* and *green*, respectively, and both are rendered in an italicised font with serif. Furthermore, its relations, such as its typing relation, are marked by a subscript C, even when it was unmarked in Chapter 2.

As a matter of personal preference, the metavariables for those syntactic categories that are collections of terms, types, and annotations, such as sets of names or typing environments, are typeset in the same color as the terms, types, and annotations of that system, respectively. However, the constructors of these collections are not typeset in that color, e.g. $\Gamma, x : A \vdash P$ and not $\Gamma, x : A \vdash P$. (The comma and colon are printed in blue in the second statement.) Likewise, functions that act on syntactic categories are not typeset in the corresponding colours, e.g. $\overline{A}$ and not $\overline{A}$.

Omitted Proof Sections

Some chapters end with a section titled “Omitted Proofs”. These sections contain proofs that were omitted from the main text of the chapter. Any proposition whose proof is omitted should provide a summary of the proof that references the relevant omitted proofs section. For lemmas, the proof is simply omitted, and only appears in the relevant omitted proofs section.

Embedded Papers and Pink Pages

This thesis contains three embedded papers. The text in these papers was not composed exclusively by myself, and does not always respect the conventions set out in this section. In an effort to signpost the beginning and end of these papers, they are preceded and followed by one pink page, in the following colour:

The pink page preceding each paper summarises its publication history and my contributions, and the pink page following each paper is empty.

To ensure an easy transition into the text of these papers, the section prior to each paper provide a legend—which summarises the differences in its conventions, notations, and definitions—and an erratum—which clarifies any errors in the paper.

1.3 Thesis Structure

This thesis proceeds as follows.

- In Part I, I present a variant of Classical Processes without commuting conversions, which break confluence in Wadler’s CP, and show that this variant is free from deadlock and has adequate canonical forms.

- In Part II, I present Hypersequent CP, which is a session-typed process calculus based on CP with a tighter correspondence to the π-calculus, and Hypersequent GV, which is the corresponding session-typed concurrent λ-calculus.
- In Part III, I present a variant of Priority CP without commuting conversions and introduce Priority GV which is the corresponding session-typed concurrent λ-calculus.
- In Part IV, I describe an implementation of session-typed channels based on GV and Priority GV in Linear Haskell.

Part I

Classical Processes Revisited

Chapter 2

Classical Processes Revisited

This chapter presents Classical Processes (CP), a session-typed process calculus introduced by Wadler [2012] based on Classical Linear Logic [Girard, 1987, CLL].

Wadler went to great lengths to ensure an *exact* correspondence between CP and CLL:

1. If you erase all that's written in red from the typing rules of CP, you get *exactly* the inference rules of CLL, as given by Girard [1987].
2. The operational semantics of CP resembles that of a process calculus (§ 2.1), but if you examine the proof of progress [Wadler, 2012, Proposition 2], which implements the concrete reduction strategy for CP, the manner in which the reduction rules are used resembles a proof of cut elimination for CLL in the style of Gentzen, as described for classical logic by Girard et al. [1989, § 13.2].[1]

Wadler chose to make serious concessions to CP as a process calculus in order to achieve its exact correspondence with CLL:

1. He combines name restriction and parallel composition into a single term constructor, corresponding to the logical *Cut* rule.

   As a consequence, the structural congruence for CP looks quite unlike that of any process calculus, and many of the concepts used in concurrency theory, such as labelled transition systems, bisimilarity, and observational equivalents, cannot easily be applied to CP. For example, Atkey [2017] defines an observational equivalence for CP but, in order to do so, must introduce a second,

[1] It is easy to mistake Wadler's claim for a correspondence between CP's reduction and the proof of cut elimination for the sequent calculus for CLL, as given by Girard [1987]. The citations "Girard [1987]" and "Girard et al. [1989]" are similar, and, given the context, it is reasonable to assume that Wadler is referencing a proof by Girard [1987]—except that Girard [1987] gives no such proof! He proves cut elimination for the proof nets of CLL, proves that the sequent calculus is sound and complete with respect to the proof nets, and (implicitly) obtains cut elimination for the sequent calculus as a corollary.

separate term language that once again separates name restriction and parallel composition.

2. He introduces additional reduction rules, named *commuting conversions*, into CP's operational semantics.

   The commuting conversions are not canonical for process calculi, and they are difficult to justify if we want CP to model parallel or distributed computation, since they require that if some process is blocked, waiting for input on some channel, then any other process can also become blocked, waiting for input on that same channel, whether they have access to that channel or not, and *without any communication*. (I discuss the commuting conversions in detail in § 2.3.)

The version of CP presented in this section retains the first part of the correspondence, which requires that we keep name restriction and parallel composition combined in a single term construct. There are several approaches to separating the two, which I present in Chapter 3, Chapter 4, and Chapter 5, but each weakens the strict correspondence between the typing rules and the inference rules of CLL. Therefore, I believe that adopting any of these for CP would compromise its status as a canonical reference point.

The version of CP presented in this section weakens the second part of the correspondence by dropping the commuting conversions from the operational semantics. This results in a reduction strategy much closer to that of a process calculus and repairs several defects in Wadler's CP. I believe that adopting this change does not compromise the status of CP as canonical reference point, which I discuss in detail in § 2.3 (see Proposition 2.57 and Proposition 2.58). The reduction strategy presented in this section still resembles a proof of cut elimination, just not in Gentzen's style.

In this chapter, CP's processes are printed in red, its types are printed in blue, and both are rendered in a sans-serif font. To save on accessible colour combinations, the terms and types of *any other system* are printed in *pink* and *green*, respectively, both are rendered in an italicised font with serif, and any relations, such as typing and reduction, are marked by a subscript.

This chapter proceeds as follows:

- In § 2.1, I introduce CP.
- In § 2.2, I introduce the metatheory for CP.

  Notably, I prove *preservation* (Proposition 2.27) and *progress* (Proposition 2.32), that its processes are deadlock-free (Proposition 2.39), that its canonical forms are adequate (Corollary 2.47), and that its connection graphs are trees (Proposition 2.50).

- In § 2.3, I discuss the relation between CP with and CP without commuting conversions.
- Finally, § 2.5 contains all omitted proofs.

2.1 Classical Processes

In this section, I introduce Classical Processes (CP), a session-typed process calculus that was introduced by Wadler [2012] and based on πDILL by Caires and Pfenning [2010]. CP's process calculus resembles the π-calculus [Milner et al., 1992b], and its type system is Classical Linear Logic [Girard, 1987, CLL].

The fundamental notions of programs and computation in CP—as in most process calculi—are processes and message-passing communication. Processes communicate by passing messages over channels. Unlike the π-calculus, CP's channels are *binary*, which means that communication always takes place between two processes. Each channel has exactly two endpoints. Each endpoint is held by exactly one process. Names refer to *channel endpoints* rather than the channels themselves. CP has no notion of multiparty communication, though there are extensions that address this [e.g. Carbone et al., 2016].

In the π-calculus, channel names fulfil two roles. Channel names are used as communication channels and as labels, compared with the conditional operator. In CP, the two roles are separate. Channel endpoints and labels are in different syntactic sorts, and are communicated by different send and receive operators. To simplify the presentation, CP is commonly restricted to the labels 'left' and 'right', written as inl and inr. This is no less general, as any *finite* set of labels can be encoded as a sequence of binary choices. For any practical implementation, this restriction is easily lifted.

Processes (ranged over by P, Q, R) are defined by the following grammar:

P, Q, R	::=	$x \leftrightarrow y$	link
	|	$(\nu x \bar{x})(P \parallel Q)$	cut
	|	$x[y].(P \parallel Q)$	send
	|	$x(y).P$	receive
	|	$x[].0$	close
	|	$x().P$	wait
	|	$x \triangleleft \mathsf{inl}.P$	select left
	|	$x \triangleleft \mathsf{inr}.P$	select right
	|	$x \triangleright \{\mathsf{inl}: P; \mathsf{inr}: Q\}$	choice
	|	$x \not\downarrow N$	absurd offer

The names x, y, z, and w range over the endpoints of communication channels—'channel endpoints' or 'endpoints' for short. The names $\bar{x}$, $\bar{y}$,

$\bar{z}$, and $\bar{w}$ as well as a, b, and c also range over endpoints, though they are only used in specific circumstances, which I discuss shortly. The names N and M range over sets of endpoints.

An endpoint is *bound* in the following cases:

- In $(\nu x\bar{x})(P \parallel Q)$, x and $\bar{x}$ are bound in P and Q, respectively.
- In $x[y].\,(P \parallel Q)$, y is bound in P, but not in Q.
- In $x(y).\,P$, y is bound in P.

An endpoint is *free* if it is not bound. Notably, for $x \not\frown N$, x and all names in N are free. I write $\mathrm{fn}(P)$ to denote the set of free endpoints in P. By convention, the names a, b, and c are used as a shorthand to imply to the reader that the endpoint is free.

Two endpoints are *dual* if they are bound by the same name restriction, e.g. in $(\nu x\bar{x})(P \parallel Q)$, x is bound in P, $\bar{x}$ is bound in Q, and x and $\bar{x}$ are dual. By convention, the names $\bar{x}$, $\bar{y}$, $\bar{z}$, and $\bar{w}$ are used as a shorthand to imply duality to the reader, e.g. I use x and $\bar{x}$ when they are dual endpoints of the same channel.

In CP, actions are not a well-defined syntactic sort, unlike in the π-calculus. Nonetheless, I informally write "action" in reference to the bit before the dot, e.g. the action for $x[y].\,(P \parallel Q)$ is $x[y]$. For the troublemakers without a dot, $x \triangleright \{\mathrm{inl}: P; \mathrm{inr}: Q\}$ and $x \not\frown N$, I write $x \triangleright \mathrm{inl}$, $x \triangleright \mathrm{inr}$, and $x \not\frown$, respectively. (The syntax $x \triangleright \{\mathrm{inl}: P; \mathrm{inr}: Q\}$ is the primary obstacle to separating actions out into their own syntactic sort. I discuss a variant that decomposes the offer construct in this manner in § 3.3.3.)

Types (ranged over by A, B) are the formula of CLL, as defined by the following grammar:

$$\begin{aligned} A, B \;::=\; & A \otimes B \mid \mathbf{1} \mid A \oplus B \mid \mathbf{0} \\ & \mid A \parr B \mid \bot \mid A \mathbin{\&} B \mid \top \end{aligned}$$

Duality plays an important role in CP as in CLL. Viewed from the perspective of a logic, it corresponds to negation. Viewed from the perspective of a process calculus, it guarantees that processes act on dual endpoints of the same channel in dual ways, e.g. one process sends when the other receives. As in CLL, duality is not defined as a type constructor, but as a function on types:

$$\begin{aligned} \overline{A \otimes B} &\triangleq \overline{A} \parr \overline{B} & \overline{\mathbf{1}} &\triangleq \bot \\ \overline{A \parr B} &\triangleq \overline{A} \otimes \overline{B} & \overline{\bot} &\triangleq \mathbf{1} \\ \overline{A \oplus B} &\triangleq \overline{A} \mathbin{\&} \overline{B} & \overline{\mathbf{0}} &\triangleq \top \\ \overline{A \mathbin{\&} B} &\triangleq \overline{A} \oplus \overline{B} & \overline{\top} &\triangleq \mathbf{0} \end{aligned}$$

As we will see, dual endpoints have dual types. The notation for duality (A, $\overline{A}$) and the naming convention for dual endpoints (x, $\bar{x}$) were chosen to emphasize this. Duality is involutive.

$$\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}}\ \textsc{T-Link} \qquad \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, \tilde{x} : \overline{A}}{(\nu x\tilde{x})(P \parallel Q) \vdash \Gamma, \Delta}\ \textsc{T-Cut}$$

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].\,(P \parallel Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \textsc{T-Send} \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).\,P \vdash \Gamma, x : A \parr B}\ \textsc{T-Recv}$$

$$\frac{}{x[].\,0 \vdash x : \mathbf{1}}\ \textsc{T-Close} \qquad \frac{P \vdash \Gamma}{x().\,P \vdash \Gamma, x : \bot}\ \textsc{T-Wait}$$

$$\frac{P \vdash \Gamma, x : A}{x \triangleleft \mathrm{inl}.\,P \vdash \Gamma, x : A \oplus B}\ \textsc{T-Select}_1 \qquad \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathrm{inl}.\,P \vdash \Gamma, x : A \oplus B}\ \textsc{T-Select}_2$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathrm{inl}\colon P; \mathrm{inr}\colon Q\} \vdash \Gamma, x : A \,\&\, B}\ \textsc{T-Offer} \qquad \frac{N = \mathrm{fn}(\Gamma)}{x \lightning N \vdash \Gamma, x : \top}\ \textsc{T-Absurd}$$

Figure 2.1: Typing Rules for Classical Processes

Lemma 2.1. $\overline{\overline{A}} = A$

Typing environments (ranged over by Γ, Δ) are sets of type assignments, as defined by the following grammar:

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

The set of free endpoint names in a typing environment, written $\mathrm{fn}(\Gamma)$, is defined by recursion on the typing environment, i.e. $\mathrm{fn}(\cdot) \triangleq \varnothing$ and $\mathrm{fn}(\Gamma, x : A) \triangleq \mathrm{fn}(\Gamma) \cup \{x\}$. The extension $\Gamma, x : A$ is only defined when x is not free in Γ, i.e. $x \notin \mathrm{fn}(\Gamma)$.

I write Γ, Δ for the concatenation of typing environments Γ and Δ. The concatenation Γ, Δ is only defined when the names in Γ and Δ are unique, i.e. $\mathrm{fn}(\Gamma) \cap \mathrm{fn}(\Delta) = \varnothing$.

The typing judgment $P \vdash \Gamma$ means that P is well-typed if, for each type assignment $x : A$ in Γ, exactly one process in P uses the endpoint x according to the session type A.

Definition 2.2 (Typing)**.** *A process P is well-typed under typing environment Γ if there exists a derivation with conclusion $P \vdash \Gamma$ that uses the typing rules in Figure 2.1.*

The names $\mathscr{E}$ and $\mathscr{F}$ range over *evaluation contexts*. Evaluation contexts are one-hole process contexts that consist only of cuts.

Definition 2.3 (Evaluation Context)**.** Evaluation contexts *are one-hole process contexts, as defined by the following grammar:*

$$\mathscr{E}, \mathscr{F} ::= \Box \mid (\nu x\tilde{x})(\mathscr{E} \parallel Q) \mid (\nu x\tilde{x})(Q \parallel \mathscr{E})$$

$$
\begin{array}{lll}
x{\leftrightarrow}y & \equiv y{\leftrightarrow}x & \textsc{SC-LinkComm} \\
(\nu x\bar{x})(P \parallel Q) & \equiv (\nu \bar{x}x)(Q \parallel P) & \textsc{SC-CutComm} \\
(\nu x\bar{x})((\nu y\bar{y})(P \parallel Q) \parallel R) & \equiv (\nu y\bar{y})((\nu x\bar{x})(P \parallel R) \parallel Q) & \textsc{SC-CutAssoc} \\
\multicolumn{3}{c}{\text{where } x \notin Q \text{ and } y \notin R}
\end{array}
$$

Figure 2.2: Structural Congruence for Classical Processes

Plugging is defined by replacing the one hole with a process:

$$
\begin{array}{ll}
\Box & [P] \triangleq P \\
(\nu x\bar{x})(\mathscr{E} \parallel Q)[P] & \triangleq (\nu x\bar{x})(\mathscr{E}[P] \parallel Q) \\
(\nu x\bar{x})(Q \parallel \mathscr{E})[P] & \triangleq (\nu x\bar{x})(Q \parallel \mathscr{E}[P])
\end{array}
$$

I write $\mathrm{fn}(\mathscr{E})$ *for the free endpoints in* $\mathscr{E}$.

I write $\mathrm{bn}(\mathscr{E})$ *for the endpoints bound by* $\mathscr{E}$.

I write $\mathscr{E} \vdash \Gamma' \to \Gamma$ *to mean that the evaluation context* $\mathscr{E}$ *is well-typed under input typing context* Γ' *and output typing context* Γ.

$$
\frac{}{\Box \vdash \Gamma \to \Gamma} \qquad \frac{\mathscr{E} \vdash \Gamma' \to \Gamma, x : A \quad Q \vdash \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})(\mathscr{E} \parallel Q) \vdash \Gamma' \to \Gamma, \Delta} \qquad \frac{Q \vdash \Gamma, x : A \quad \mathscr{E} \vdash \Delta' \to \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})(Q \parallel \mathscr{E}) \vdash \Delta' \to \Gamma, \Delta}
$$

Processes are considered equivalent up to structural congruence.

Definition 2.4 (Structural Congruence)**.** Structural congruence, *written* $P \equiv Q$, *is the congruence closure over processes which satisfies the rules in Figure 2.2.*

The semantics of CP processes is given by *reduction*. Reduction is closed over evaluation contexts, and structural congruence is embedded in reduction by allowing pre- and post-composition using E-Cong, written as $\equiv\!\!\longrightarrow$, $\longrightarrow\!\!\equiv$, or $\equiv\!\!\longrightarrow\!\!\equiv$.

Definition 2.5 (Reduction)**.** Reduction, *written* $P \longrightarrow Q$, *is the smallest relation on processes defined by the rules in Figure 2.3.*

In the remainder of the section, I discuss each process construct together with its typing rule and operational semantics, either by itself—e.g. *link*—or together with its dual—e.g. *send* and *receive*.

2.1.1 Process Structure

The process $(\nu x\bar{x})(P \parallel Q)$ denotes a *cut*, which creates a communication channel with two endpoints, x and $\bar{x}$. For communication safety, the two endpoints must have dual types. Hence, $x : A$ and $\bar{x} : \overline{A}$. For deadlock freedom, the two endpoints must be used in different processes, and those processes cannot share any other channels. Hence, x is bound in

$$
\begin{array}{lll}
(\nu x\bar{x})(x{\leftrightarrow}w \parallel P) & \longrightarrow P\{w/\bar{x}\} & \text{E-Link} \\
(\nu x\bar{x})(x[y].\,(P \parallel Q) \parallel \bar{x}(\bar{y}).\,R) & \longrightarrow (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R)) & \text{E-Send} \\
(\nu x\bar{x})(x[].\,0 \parallel \bar{x}().\,Q) & \longrightarrow Q & \text{E-Close} \\
(\nu x\bar{x})(x \triangleleft \mathrm{inl}.\,P \parallel \bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\}) & \longrightarrow (\nu x\bar{x})(P \parallel Q) & \text{E-Select}_1 \\
(\nu x\bar{x})(x \triangleleft \mathrm{inr}.\,P \parallel \bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\}) & \longrightarrow (\nu x\bar{x})(P \parallel R) & \text{E-Select}_2
\end{array}
$$

$$
\text{E-Equiv}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}
\qquad\qquad
\text{E-Cong}\ \frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']}
$$

Figure 2.3: Reduction for Classical Processes

P and $\bar{x}$ is bound in Q, and the typing environment is split between P and Q.

$$
\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})(P \parallel Q) \vdash \Gamma, \Delta}\ \text{T-Cut}
$$

The cut combines two process constructs from the π-calculus: name restriction $(\nu x\bar{x})$ and parallel composition $P \parallel Q$. The two do not detach. In CP, $(\nu x\bar{x})P$ and $P \parallel Q$ are not syntactically well-formed processes.

Cuts are *commutative* and *quasi-associative*:

$$
\begin{array}{lcll}
(\nu x\bar{x})(P \parallel Q) & \equiv & (\nu \bar{x}x)(Q \parallel P) & \text{SC-CutComm} \\
(\nu x\bar{x})((\nu y\bar{y})(P \parallel Q) \parallel R) & \equiv & (\nu y\bar{y})((\nu x\bar{x})(P \parallel R) \parallel Q) & \text{SC-CutAssoc} \\
 & & \text{where } x \notin Q \text{ and } y \notin R &
\end{array}
$$

The SC-CutAssoc rule is not quite an associativity rule. Hence, *quasi-associative*. Wadler [2012] refers to the rule as "(Assoc)" in reference to the change of bracketing. In fact, the rule combines the associativity rule of parallel composition with scope extrusion, which commutes parallel composition and name restriction, as we will see in § 3.2.7 (see Proposition 3.76).

There is no specific reduction rule for cut. Rather, all other reduction rules resolve dual communication actions *under a cut*, i.e. over an existing channel. E-Cong allows reduction under cuts.

$$
\text{E-Cong}\ \frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']}
$$

2.1.2 Link

The process $x{\leftrightarrow}y$ denotes a *link*. It forwards any messages received on x to y, and vice versa. For communication safety, the two endpoints must have dual types. Hence, $x : A$ and $y : \overline{A}$.

$$
\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}}\ \text{T-Link}
$$

Links are *commutative*. If two channels are connected by a link, the order in which they are connected is irrelevant. This property is captured by the following equivalence:

$$x{\leftrightarrow}y \equiv y{\leftrightarrow}x \quad \text{SC-LinkComm}$$

CP's semantics for link does not explicitly forward messages, but treats links as suspended α-renaming. When a link $x{\leftrightarrow}y$ reduces, it renames all occurrences of the dual of x to y, or all occurrences of the dual of y to x. In essence, this updates all the processes connected to the one side of the link to point directly at the other side, circumventing the link.

$$(\nu x\bar{x})(x{\leftrightarrow}y \parallel P) \longrightarrow P\{y/\bar{x}\} \quad \text{E-Link}$$

The renaming targets a bound name. Hence, there cannot be any other occurrences of that name, and the link can be removed. The rule E-Link gives link an asynchronous semantics, as P is not required to be ready on $\bar{x}$, whereas all other actions are synchronous. (I discuss synchronous semantics for link in § 3.3.2.)

2.1.3 Send and Receive

The send and receive actions are dual:

- The process $x[y].\,(P \parallel Q)$ denotes a *send* action.
  It creates a fresh channel, names one endpoint of that channel y, sends the other endpoint over x, then continues as P and Q in parallel, where y is bound in P and x is bound in Q.
- The process $x(y).\,P$ denotes a *receive* action.
  It receives an endpoint over x, names it y, then continues as P.

The typing rules for send and receive are as follows:

$$\frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].\,(P \parallel Q) \vdash \Gamma, \Delta, x : A \otimes B}\ \text{T-Send} \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).\,P \vdash \Gamma, x : A \parr B}\ \text{T-Recv}$$

The behaviour of send and receive is given by the following rule:

$$(\nu x\bar{x})(x[y].\,(P \parallel Q) \parallel \bar{x}(\bar{y}).\,R) \longrightarrow (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R)) \quad \text{E-Send}$$

After a send action, the endpoints y and x are passed to different parallel processes, which ensures that they are handled independently. After a receive action, the two endpoints are kept by the same process. This may seem restrictive—*and it is*—but it is paramount to type preservation and deadlock freedom.

The send action is a *bound* send. It cannot send just any old endpoint that the process happens to have laying around. It only sends *fresh* endpoints. As part of the reduction, the send action *creates* a fresh channel with two

fresh endpoints. It sends one of those endpoints, and keeps the other. You can see this in E-SEND. The left-hand side has one name restriction, $(\nu x\bar{x})$, but the right-hand side has two, creating a fresh channel with $(\nu y\bar{y})$.

You can think of the bound send action as a suspended cut. For deadlock freedom, the two processes connected by a cut cannot share any other channel. (The relation between $\otimes$ and cut is widely known in logic, e.g. Girard [1987] treats cut as a special case of $\otimes$.)

I could extend CP with an *unbound* send action. Lindley and Morris [2015] discuss such an extension [section 3.1, under "A Simpler Send"]. They propose the following syntax, typing, and reduction rules:

$$P, Q, R ::= \cdots \mid x\langle y\rangle.\, P$$

$$\text{T-USEND}\quad \frac{P \vdash \Gamma, y : A, x : B}{x\langle y\rangle.\, P \vdash \Gamma, x : A \otimes B, y : \overline{A}} \qquad \text{E-USEND}\quad (\nu x\bar{x})(x\langle y\rangle.\, P \parallel \bar{x}(z).\, Q) \longrightarrow (\nu x\bar{x})(P \parallel Q\{y/z\})$$

The unbound send $x\langle y\rangle.\, P$ sends the *free* endpoint y over x, then continues as P. In the π-calculus, unbound send is strictly more expressive than bound send [see Boreale, 1996, Sangiorgi, 1996]. In CP, unbound send can be defined in terms of bound send, but the converse does not hold. Unbound send can be defined as follows:

$$x\langle y\rangle.\, P \triangleq x[z].\, (z{\leftrightarrow}y \parallel P)$$

The definition has the same reduction behavior as E-USEND, albeit with an extra step for the link reduction.

$$(\nu x\bar{x})(x\langle y\rangle.\, P \parallel \bar{x}(z).\, Q) \longrightarrow (\nu x\bar{x})(P \parallel Q\{y/z\})$$

$$\begin{aligned}(\nu x\bar{x})(x\langle y\rangle.\, P \parallel \bar{x}(z).\, Q) &\triangleq (\nu x\bar{x})(x[w].\, (w{\leftrightarrow}y \parallel P) \parallel \bar{x}(z).\, Q)\\ &\longrightarrow (\nu x\bar{x})(P \parallel (\nu w\bar{w})(w{\leftrightarrow}y \parallel Q\{\bar{w}/z\}))\\ &\longrightarrow (\nu x\bar{x})(P \parallel Q\{y/z\})\end{aligned}$$

The converse does not hold. One might try and define bound send in terms of unbound send as follows:

$$x[y].\, (P \parallel Q) \triangleq (\nu y\bar{y})(P \parallel x\langle\bar{y}\rangle.\, Q)$$

However, the two do not behave quite the same. In the former, reductions in both P and Q are blocked on the send action. In the latter, only reductions in Q are blocked on the send action, while reductions in P can happen before the send action.

In Wadler's CP, the adoption of unbound send breaks the proof of progress, as it introduces cuts that cannot easily be eliminated by means of a commuting conversion. The cut in $(\nu x\bar{x})(a\langle x\rangle.\, P \parallel x().\, Q)$ cannot be eliminated by a commuting conversion, as the unbound send action $a\langle x\rangle$

cannot commute past it, which breaks the simple left-to-right evaluation strategy proposed by Wadler [2012]. In CP, as presented in this chapter, the adoption of unbound send would not pose any problem, as my canonical form accounts for and justifies ineliminable top-level cuts.

By its very nature, bound send can only send endpoints, which means it does not easily generalise to sending other data. Hence, unbound send is important in the context of GV, where programs can send arbitrary data.

In the interest of an exact correspondence with CLL, and a close correspondence to Wadler's CP, the version of CP presented in this chapter uses bound send.

2.1.4 Close and Wait

The close and wait actions are dual:

- The process $x[].\,0$ denotes a *close* action.
  It sends a ping over x, then terminates.
- The process $x().\,P$ denotes a *wait* action.
  It receives a ping over x, then continues as P.

(I say 'ping' to imply the interaction between close and wait is merely a synchronisation and does not transmit any information.)

The typing rules for close and wait are as follows:

$$\frac{}{x[].\,0 \vdash x : \mathbf{1}}\ \text{T-Close} \qquad \frac{P \vdash \Gamma}{x().\,P \vdash \Gamma, x : \bot}\ \text{T-Wait}$$

The behaviour of these two actions is given by the following rule:

$$(\nu x \bar{x})(x[].\,0 \parallel \bar{x}().\,Q) \longrightarrow Q \quad \text{E-Close}$$

After a close action, the process must terminate, but after a wait action, the process continues. As with send and receive, this seems restrictive—*because it is restrictive*—but is paramount to type preservation.

The close and wait actions close the channel, which removes the cut. Any two processes connected by a cut cannot share any other channel. If both processes were allowed to continue, that would leave them unconnected. This would break an important property of CP—that all processes are connected—and relaxing it would be logically equivalent to admitting the Mix rule (see § 3.3.7) and would break the exact correspondence with CLL. (Such a variant of CP is explored by Atkey et al. [2016].)

2.1.5 Select and Offer

The select and offer actions are dual:

- The process $x \triangleleft \mathsf{inl}.\, P$ denotes a *left selection* action.
  It sends the label inl over x, then continues as P.
- The process $x \triangleleft \mathsf{inr}.\, P$ denotes a *right selection* action.
  It sends the label inr over x, then continues as P.
- The process $x \triangleright \{\mathsf{inl}: P; \mathsf{inr}: Q\}$ denotes a *choice* action.
  It receives a label over x, and then continues as either P or Q, depending on which label was received.

The typing rules for select and offer are as follows:

$$\frac{P \vdash \Gamma, x : A}{x \triangleleft \mathsf{inl}.\, P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_1 \qquad \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathsf{inl}.\, P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_2$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl}: P; \mathsf{inr}: Q\} \vdash \Gamma, x : A \,\&\, B}\ \text{T-Offer}$$

The behaviour of these actions is given by the following rules:

$$(\nu x \bar{x})(x \triangleleft \mathsf{inl}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\}) \longrightarrow (\nu x \bar{x})(P \parallel Q)\quad \text{E-Select}_1$$
$$(\nu x \bar{x})(x \triangleleft \mathsf{inr}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\}) \longrightarrow (\nu x \bar{x})(P \parallel R)\quad \text{E-Select}_2$$

The syntax for the select and offer actions was adapted from Dardha and Gay [2018], rather than from Wadler [2012]. It is easily generalized from binary choice to variant types by removing the restriction that labels must always be drawn from $\{\mathsf{inl}, \mathsf{inr}\}$. (I discuss variant types in § 3.3.3.)

2.1.6 The Absurd Offer

The process $x \not\triangleright N$ denotes the *absurd offer*. It waits to receive a choice between *zero* alternatives. Such a choice cannot be made, which means that there is no corresponding select action, and no corresponding reduction rule. In essence, an absurd offer is inert. The absurd offer is the sole process that is allowed to leave endpoints unused, and the set of those unused endpoints is denoted by N.

$$\frac{N = \mathrm{fn}(\Gamma)}{x \not\triangleright N \vdash \Gamma, x : \top}\ \text{T-Absurd}$$

The 'inert semantics' for absurd is unsatisfying—something has clearly gone wrong, and instead of cleaning up, we leave the garbage strewn around. However, it is not *incorrect*. By accident, it satisfies all the expected properties of normalisation for CP. (The accident is the fact that CP processes are fully connected. Hence, even if $x \triangleright \{\}$ did not have immediate license to kill some process P, it would have obtained it eventually.) In variants where processes are not always fully connected, the inert semantics break progress (see, e.g. Chapter 3). (I present an *exceptional* semantics for the absurd offer in § 3.3.1[2].)

[2] I use 'exceptional' to mean 'reminiscent of exceptions', but I also happen to believe they are very good semantics.

The syntax for the absurd offer deviates from the naming scheme for the binary select and offer actions. If I were to follow the naming scheme, I would write '$x \triangleright \{\}$'. However, it is important for the absurd offer to record which endpoints it discards. For one, recording those names allows us to define linearity by counting names (as in § 2.2.1). More importantly, to give an exceptional semantics to the absurd offer, those names are needed to be able to erase types. For an intuition, let us look at the absurd offer in Wadler's CP. It does not record the discarded endpoints and it has the following reduction rule:

$$\frac{\dfrac{}{a \triangleright \{\} \vdash \Gamma, a : \top, x : A} \qquad P \vdash \Delta, \bar{x} : \overline{A}}{(\nu x \bar{x})(a \triangleright \{\} \parallel P) \vdash \Gamma, \Delta, a : \top} \longrightarrow \frac{}{a \triangleright \{\} \vdash \Gamma, \Delta, a : \top}$$

If the absurd offer discarded an endpoint x, it may kill the process P that owns the dual endpoint $\bar{x}$. The reduction rule appears innocuous, but a problem surfaces when we erase types and consider the equivalent reduction rule on *processes*:

$$(\nu x \bar{x})(a \triangleright \{\} \parallel P) \longrightarrow a \triangleright \{\}$$

It is no longer possible to see whether P owns any of the endpoints dual to those discarded, which means that $a \triangleright \{\}$ has license to kill *any process*[3]. Surprisingly, by itself this does not pose a *huge* problem for CP. Ultimately, all processes in CP are connected, so even if $a \triangleright \{\}$ were to kill a process it was not immediately connected to, it would have gotten there eventually. However, if the language were extended with parallel but unconnected processes—as in, e.g. HCP—this leads to immediate problems, where some $a \triangleright \{\}$ kills a process it was in no way connected to. It also leads to issues with scope extrusion in SC-CutAssoc:

$$(\nu x \bar{x})((\nu y \bar{y})(P \parallel a \triangleright \{\}) \parallel b \triangleright \{\}) \equiv (\nu y \bar{y})((\nu x \bar{x})(P \parallel b \triangleright \{\}) \parallel a \triangleright \{\})$$

If $a \triangleright \{\}$ is responsible for killing x on the left-hand side of the structural congruence, then on the right-hand side it moves out of the scope of $(\nu x \bar{x})$.

My syntax, $a \lightning N$, is an effort to avoid clutter: I omit the absurd continuation and add a lightning bolt, which, to me, implies that something has gone wrong[4].

2.2 Metatheory

In this section, I introduce the metatheory for Classical Processes. This section proceeds as follows:

[3] Wadler was unaware of the issue with the absurd offer and type erasure.

[4] The lightning bolt is an homage to Simon Fowler's Exceptional GV, where it marks "zapper threads" [Fowler, 2019, Figure 9.4], and to a poster titled "*zap* is why we can't have nice things" that I presented at my CDT's 2016 Industrial Engagement Event.

- In § 2.2.1, I give several preliminary definitions that are used throughout the discussion of the metatheory.
- In § 2.2.2, I prove preservation (Proposition 2.27). In essence, the proof is a reproduction of Wadler's proof of preservation, but without the commuting conversions and with small changes to account for the my changes to CP's syntax.
- In § 2.2.3, I define canonical form (Definition 2.30) and prove progress (Proposition 2.32). The proof cannot be reproduced from Wadler [2012], as the reduction rules I give in § 2.1 are a proper subset of those given by Wadler [2012]. I could adapt the proof by Lindley and Morris [2015], but it relies on the property that CP processes can be rewritten to right-branching form (see Proposition 2.51), which does not hold for the variants of CP that I discuss in later chapters. The proof for progress I give in this section generalises Lindley and Morris' appeal to right-branching form using process contexts, which allows me to reuse the same proof structure in later chapters. This proof of progress first appeared in my M.Sc. thesis [Kokke, 2017].
- In § 2.2.4, I define dependency graphs for CP processes (Definition 2.35). I prove that CP is deadlock-free, as its dependency graphs are always acyclic (Proposition 2.39), and I prove that my definition of canonical form is adequate (Corollary 2.47) and stronger than Wadler's definition.
- In § 2.2.5, I define connection graphs for CP processes (Definition 2.49) and prove that CP's connection graphs are always trees (Proposition 2.50). The validity of right-branching form for CP follows as a corollary.
- In § 2.2.6, I sketch the basis of a behavioural theory for CP.

2.2.1 Preliminaries

Configuration Contexts

Processes in CP can be rewritten to right-branching form:

$$(\nu x_1 \bar{x}_1)(P_1 \parallel \cdots (\nu x_n \bar{x}_n)(P_n \parallel P_{n+1}) \cdots)$$

Right-branching form is a convenient form to work with. It neatly captures the prefix of top-level cuts and the connected processes, and it relates the endpoints and processes in a predictable manner—each x_i is free in P_i, and each $\bar{x}_i$ is free in some P_j where $j > i$.

Right-branching form has its downsides. Proving that processes can be rewritten to right-branching form is far from trivial (Proposition 2.51). More importantly, right-branching form is a quirk of CP's rigid process

structure. In variants of CP with more expressive process structure, processes cannot be rewritten to right-branching form.

I generalise right-branching form using *configuration contexts*. Configuration contexts are multi-hole process contexts that consist only of cuts. For instance, for the process above, an example configuration context is

$$(\nu x_1 \bar{x}_1)(\Box_1 \parallel \cdots (\nu x_n \bar{x}_n)(\Box_n \parallel \Box_{n+1}) \cdots)$$

The expressions $\Box_1$, ..., $\Box_{n+1}$ are the $n+1$ holes, numbered from left to right for convenience. The name $\mathscr{C}^k$ ranges over configuration contexts, where k denotes the number of holes. Plugging, written $\mathscr{C}^k[P_1, ..., P_k]$, replaces the holes, from left to right, with the processes P_1, ..., P_k, such that each $\Box_i$ is replaced with the process P_i (for $1 < i \leq k$). If $\mathscr{C}^{n+1}$ refers to the example configuration context shown above and P_1, ..., P_{n+1} to the processes of the example process shown above, then $\mathscr{C}^{n+1}[P_1, ..., P_{n+1}]$ is exactly equal to the example process. The use of right-branching form requires that the prefix of cuts is a right-branching tree, i.e. a list. Configuration contexts permit arbitrary trees.

Definition 2.6 (Configuration Context)**.** Configuration contexts *are n-hole process contexts, as defined by the following grammar:*

$$\mathscr{C}, \mathscr{D} ::= \Box \mid (\nu x \bar{x})(\mathscr{C} \parallel \mathscr{D})$$

If there is risk of ambiguity, we explicitly write the number of holes in a configuration context with a superscript, e.g. as $\mathscr{C}^n$*.*

Plugging is defined by replacing the n holes with n processes, left to right:

$$\begin{aligned} \Box \quad & [P \quad] \triangleq P \\ (\nu x \bar{x})(\mathscr{C}^n \parallel \mathscr{D}^k)\ & [P_1, ..., P_n, P_{n+1}, ..., P_{n+k}] \triangleq (\nu x \bar{x})(\mathscr{C}^n[P_1, ..., P_n] \parallel \mathscr{D}^k[P_{n+1}, ..., P_{n+k}]) \end{aligned}$$

I write $\mathscr{C}[P_1,...,\Box_i,...,P_n]$ *for the evaluation context focused on the i'th hole in* $\mathscr{C}$ *such that* $\mathscr{C}[P_1,...,\Box_i,...,P_n][P_i] = \mathscr{C}[P_1,...,P_i,...,P_n]$*.*

I write $\mathrm{dn}(\mathscr{C})$ *for the unordered pairs of dual endpoints bound by* $\mathscr{C}$*.*

$$\begin{aligned} \mathrm{dn}(\Box) &\triangleq \varnothing \\ \mathrm{dn}((\nu x \bar{x})(\mathscr{C} \parallel \mathscr{D})) &\triangleq \{\{x, \bar{x}\}\} \cup \mathrm{dn}(\mathscr{C}) \cup \mathrm{dn}(\mathscr{D}) \end{aligned}$$

I write $\mathrm{bn}(\mathscr{C})$ *for the endpoints bound by* $\mathscr{C}$*, i.e.* $\mathrm{bn}(\mathscr{C}) \triangleq \bigcup \mathrm{dn}(\mathscr{C})$*.*

I write $\mathscr{C}^n \vdash \Gamma_1 \mid \cdots \mid \Gamma_n \to \Gamma$ *to mean that the configuration context* $\mathscr{C}^n$ *is well-typed under input typing contexts* $\Gamma_1, \ldots, \Gamma_n$ *and output typing context* Γ*.*

$$\frac{}{\Box \vdash \Gamma \to \Gamma} \qquad \frac{\mathscr{C}^n \vdash \Gamma_1 \mid \cdots \mid \Gamma_n \to \Gamma, x : A \quad \mathscr{D}^k \vdash \Delta_1 \mid \cdots \mid \Delta_k \to \Delta, \bar{x} : \overline{A}}{(\nu x \bar{x})(\mathscr{C}^n \parallel \mathscr{D}^k) \vdash \Gamma_1 \mid \cdots \mid \Gamma_n \mid \Delta_1 \mid \cdots \mid \Delta_k \to \Gamma, \Delta}$$

Shallow Structural Congruence

Reduction is closed under evaluation contexts, not under arbitrary process contexts, and only acts on the topmost actions. As such, reduction only needs a structural congruence that is similarly closed under evaluation contexts—a *shallow* structural congruence. Therefore, it is useful to distinguish different kinds of structural congruence, based on which portions of the process they are permitted to rewrite.

Definition 2.7 (Shallow Structural Congruence)**.** Shallow structural congruence*, written* $\overset{s}{\equiv}$*, is the smallest symmetric relation over processes that satisfies the rules in Figure 2.2 (p. 34) and is closed under evaluation contexts, as per the following rule:*

$$\textit{SC-Cong}\ \frac{P \overset{s}{\equiv} P'}{\mathscr{E}[P] \overset{s}{\equiv} \mathscr{E}[P']}$$

Most rules of the structural congruence target name restriction and parallel composition. The odd one out is SC-LinkComm, which rewrites a link action. It will be useful to single out the portions of a structural congruence that rewrite links.

Definition 2.8 (Link-Preserving Structural Congruence)**.** Link-preserving structural congruence*, written* $\overset{l}{\equiv}$*, is the congruence closure over processes that satisfies the rules in Figure 2.2 except for SC-LinkComm.*

Finally, it will be useful to have variants which combine these restrictions. In practice, I only need link-preserving shallow structural congruence and deep structural congruence.

Definition 2.9 (Link-Preserving Shallow Structural Congruence)**.** Link-preserving shallow structural congruence*, written* $\overset{ls}{\equiv}$*, is the intersection of link-preserving and shallow structural congruence.*

Definition 2.10 (Deep Structural Congruence)**.** Deep structural congruence*, written* $\overset{d}{\equiv}$*, is the equivalence closure of the complement of* $\overset{ls}{\equiv}$ *with respect to* $\equiv$*.*

Any structural congruence can be decomposed into its link-preserving shallow and deep structural components.

Lemma 2.11. *If* $P \equiv Q$*, then there exists some* R *such that* $P \overset{ls}{\equiv} R$ *and* $R \overset{d}{\equiv} Q$*.*

Ready Processes and Threads

A process is ready if it is ready to perform some communication action, i.e. if it is a link or it is prefixed by an action. The formal definition of *ready* processes is a bit verbose, since actions are not a syntactic sort in

CP, so "prefixed by an action" is not well-defined. To work around this, I enumerate the process constructors that contain an action.

Definition 2.12 (Ready)**.** *A process* P *is* ready to act on x, *written* ready(P, x)*, if it is of one of the forms:*

$$\begin{array}{llllll} x{\leftrightarrow}y & x[y].\,(P \parallel Q) & x[].\,0 & x \triangleleft \mathit{inl}.\,P & x \triangleright \{\mathit{inl}\colon P;\, \mathit{inr}\colon Q\} \\ y{\leftrightarrow}x & x(y).\,P & x().\,P & x \triangleleft \mathit{inr}.\,P & x \lightning N \end{array}$$

A process is ready *if it is ready to act on some endpoint.*

In particular, links $x{\leftrightarrow}y$ are considered ready to act on both x and y, and absurd $x \lightning N$ is *not* considered ready to act on the channels $y \in N$.

A process can be decomposed into a prefix of its cuts, and a series of threads connected by those cuts. Such a prefix is the *maximum* configuration context, in the sense that no further cuts can be added.

Definition 2.13 (Maximum Configuration Context)**.** *The* maximum configuration context $\mathscr{C}^n$ *of a process* P *is the configuration context such that* $P = \mathscr{C}^n[P_1, \ldots, P_n]$ *(for some* $n \geq 1$*) and (for* $1 \leq i \leq n$*) each* P_i *is ready. The processes* P_i *are the* threads *of* P*. Every process has a unique maximum configuration context.*

Likewise, evaluation contexts are maximal if no further cuts can be added. Informally, maximal evaluation contexts are paths to the threads contained within some process, so each maximum configuration context $\mathscr{C}^n$ gives us n distinct maximal evaluation contexts.

Definition 2.14 (Maximal Evaluation Context)**.** *A* maximal evaluation context $\mathscr{E}$ *of a process* P *is an evaluation context such that* $P = \mathscr{E}[Q]$ *and* Q *is ready. If* $\mathscr{C}$ *is the maximum configuration context of* P*, then* $\mathscr{C}[P_1, \ldots, \square_i, \ldots, P_n]$ *is a maximal evaluation context of* P*.*

Finally, I refer to top-level ready processes as *threads*. A significant portion of CP's metatheory deals with threads. Let the metavariable T range over threads.

Definition 2.15 (Thread)**.** *A process* P *is a* thread of Q *if there exists some evaluation context* $\mathscr{E}$ *of* Q *such that* $Q = \mathscr{E}[P]$ *and* P *is ready. I say that* Q contains the thread P *to mean that* P *is a thread of* Q*. I say* P *is a* thread *when the process* Q *of which* P *is a thread can be inferred from context. Let* T *range over threads.*

The use of the thread metavariable allows us to succinctly decompose any process P into its maximum configuration context and its threads, by stating "P is of the form $\mathscr{C}[T_1, \ldots, T_n]$", as the notation implies that each thread T_i is a ready process and therefore that $\mathscr{C}$ is the maximum configuration context.

In CP, every thread is ready. Therefore, it is tempting to do away with the separate definition of threads and let T range over ready processes.

However, the two definitions diverge for GV, since GV's threads have internal reduction behaviour and therefore not all threads are ready. To keep my terminology consistent, I keep the definitions of ready processes and threads separate.

Process Contexts

For the occasional convenience, I define full *n*-hole process contexts, which are arbitrary processes with any number of holes. Process contexts may contain any process construct, and may contain holes in any position, including nested under actions. As such, process contexts generalise evaluation and configuration contexts.

Definition 2.16 (Process Context)**.** Process contexts *are defined by the grammar for processes, extended with the hole constructor, written* $\square$*. A process context may have any number of holes.*

The names $P[\cdot]$, $Q[\cdot]$, *and* $R[\cdot]$ *range over process contexts, where the trailing* $[\cdot]$ *is intended to help distinguish between processes and process contexts, and denotes the position of the arguments for plugging. I write* $P^n[\cdot]$ *to denote that the process context* $P[\cdot]$ *has n holes.*

Plugging, written $P[P_1, \ldots, P_n]$, *is defined by replacing the n holes in the process context* $P[\cdot]$ *with the processes* $P_1, \ldots, P_n$ *in order from left to right.*

Linearity

CP has a *linear* type system. It ensures that resources are always used exactly once, and never copied or dropped. Given the correspondence between CP and linear logic, that seems almost painfully obvious. However, due to CP's reuse of endpoint names, it may appear that resources are used multiple times. (For instance, the process $x().\, x[].\, 0$ appears to use the endpoint x twice.)

The true measure of a linear calculus lies in its execution. Any execution should use all linear resources exactly once. Unfortunately, such a formulation of linearity is a bit tedious.

For my purposes, it suffices to define linearity by counting the uses of free endpoint names—taking the sum across any parallel composition, and the union across an offer. This formulation reveals the implicit rebinding of names in CP's actions. For instance, the action $x(y).\, P$ uses the name x, but then binds a *fresh* x for the remainder of the session.

Definition 2.17 (Free Name Count)**.** *The multiset of free endpoints in* P, *written* $\underline{\text{fn}}(P)$, *is a multiset (see Definition A.3) with support set* $\text{fn}(P)$ *and*

multiplicity function $\mu_{\underline{\mathrm{fn}}(P)}$.

$$
\begin{array}{lcl}
\underline{\mathrm{fn}}(x{\leftrightarrow}y) & \triangleq & \lbag x, y \rbag \\
\underline{\mathrm{fn}}((\nu x\bar{x})(P \parallel Q)) & \triangleq & (\underline{\mathrm{fn}}(P) \setminus \{x\}) + (\underline{\mathrm{fn}}(Q) \setminus \{\bar{x}\}) \\
\underline{\mathrm{fn}}(x[y].\,(P \parallel Q)) & \triangleq & \lbag x \rbag + (\underline{\mathrm{fn}}(P) \setminus \{y\}) + (\underline{\mathrm{fn}}(Q) \setminus \{x\}) \\
\underline{\mathrm{fn}}(x[].\,0) & \triangleq & \lbag x \rbag \\
\underline{\mathrm{fn}}(x(y).\,P) & \triangleq & \lbag x \rbag + (\underline{\mathrm{fn}}(P) \setminus \{x, y\}) \\
\underline{\mathrm{fn}}(x().\,P) & \triangleq & \lbag x \rbag + \underline{\mathrm{fn}}(P) \\
\underline{\mathrm{fn}}(x \lhd \mathit{inl}.\,P) & \triangleq & \lbag x \rbag + (\underline{\mathrm{fn}}(P) \setminus \{x\}) \\
\underline{\mathrm{fn}}(x \lhd \mathit{inr}.\,P) & \triangleq & \lbag x \rbag + (\underline{\mathrm{fn}}(P) \setminus \{x\}) \\
\underline{\mathrm{fn}}(x \rhd \{\mathit{inl}\colon P; \mathit{inr}\colon Q\}) & \triangleq & \lbag x \rbag + ((\underline{\mathrm{fn}}(P) \setminus \{x\}) \cup (\underline{\mathrm{fn}}(Q) \setminus \{x\})) \\
\underline{\mathrm{fn}}(x \not\downarrow N) & \triangleq & \lbag x \rbag + \lbag w \mid w \in N \rbag
\end{array}
$$

Note that the operation $\mathscr{X} \setminus X$ removes all occurrences of the elements in the set X from the multiset $\mathscr{X}$ (see Definition A.3).

The only unusual case is the case for the absurd offer $x \not\downarrow N$, which counts all names in N as used. This seems wrong, since the absurd offer never uses the names in N. However, it is correct in the tedious sense. All executions of $x \not\downarrow N$ use all the names in N, trivially, since $x \not\downarrow N$ is never executed.

For actions, $\underline{\mathrm{fn}}$ removes all previous occurrences of the relevant name, which hides any potential non-linear usage of that name, e.g. $\underline{\mathrm{fn}}(x().\,P) \triangleq \lbag x \rbag + (\underline{\mathrm{fn}}(P) \setminus \{x\})$ hides the count of x in P. Hence, $\underline{\mathrm{fn}}$ is *shallow*, in the sense that it only accurately counts the top-most parallel usages of the name. This suffices for my purposes.

Linearity states that, for well-typed processes, each endpoint in the typing environment is used exactly once in the process, and vice versa.

Proposition 2.18 (Linearity)**.** *If $P \vdash \Gamma$, then:*

- $x \in^k \underline{\mathrm{fn}}(P) \implies k = 1$
- $x \in^1 \underline{\mathrm{fn}}(P) \iff x \in \mathrm{fn}(\Gamma)$

Proof. By induction on the derivation of $P \vdash \Gamma$.

□

For any well-typed process $\mathscr{E}[P]$, each endpoint bound by $\mathscr{E}$ is used exactly once in the process P, and vice versa.

Corollary 2.19. *If $\mathscr{E}[P] \vdash \Gamma$, then:*

- $x \in^k \underline{\mathrm{fn}}(P) \implies k = 1$
- $x \in^1 \underline{\mathrm{fn}}(P) \iff x \in \mathrm{bn}(\mathscr{E})$
- $\mathrm{bn}(\mathscr{E}) \subseteq \mathrm{fn}(P)$

For any well-typed configuration $\mathscr{C}^n[P_1,\ldots,P_n]$, the processes P_1, ..., P_n must collectively use all the endpoints bound by $\mathscr{C}^n$ exactly once.

Corollary 2.20. *If* $\mathscr{C}^n[P_1,\ldots,P_n] \vdash \Gamma$, *then:*

- $x \in^k \bigcup_{1\le i\le n} \underline{\mathrm{fn}}(P_i) \implies k = 1$
- $x \in^1 \bigcup_{1\le i\le n} \underline{\mathrm{fn}}(P_i) \iff x \in \mathrm{bn}(\mathscr{C})$
- $\mathrm{bn}(\mathscr{C}) \subseteq \bigcup_{1\le i\le n} \mathrm{fn}(P_i)$

Separation

Separation relates configuration contexts and evaluation contexts—it 'zooms in', from viewing a process as a series of connected processes, to viewing two specific processes and the cut connecting them. Separation also captures an essential property of CP's type system: dual endpoints must be in distinct processes, separated by a cut.

Lemma 2.21 (Separation)**.** *If* $P \vdash \Gamma$, $\mathscr{C}^n$ *is a configuration context such that* $P = \mathscr{C}^n[P_1, \ldots, P_n]$ *(for some* $n \ge 2$*), and there exists some* $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$ *such that* $x \in \mathrm{fn}(P_i)$ *and* $\bar{x} \in \mathrm{fn}(P_j)$ *(for some* $1 \le i, j \le n$*), there exist* $\mathscr{E}$, $\mathscr{F}_i$, *and* $\mathscr{F}_j$ *such that either*

1. $P = \mathscr{E}[(\nu x\bar{x})(\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j])]$, *or*
2. $P = \mathscr{E}[(\nu \bar{x}x)(\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i])]$.

Proof. By induction on the structure of the configuration context $\mathscr{C}$.

There are two cases:

- P_i and P_j are on different sides of the cut.

  There are two cases, corresponding to the two equations:

  - P_i is to the left and P_j is to the right.

    $P = (\nu x\bar{x})(\mathscr{C}_i[P_1,\ldots,P_i,\ldots,P_k] \parallel \mathscr{C}_j[P_{k+1},\ldots,P_j,\ldots,P_n])$.

    By $P \vdash \Gamma$ and T-Cut, the bound endpoints must be $\{x, \bar{x}\}$.

    $$\text{Let } \begin{array}{rcl} \mathscr{E} & = & \Box \\ \mathscr{F}_i & = & \mathscr{C}_i[P_1,\ldots,\Box_i,\ldots,P_k] \\ \mathscr{F}_j & = & \mathscr{C}_j[P_{k+1},\ldots,\Box_j,\ldots,P_n] \end{array}$$

    The result follows.

  - P_i is to the right and P_j is to the left.

    As above.

- P_i and P_j are on the same side of the cut.

  There are two cases:

  - P_i and P_j are both to the left.

    $P = (\nu z\bar{z})(\mathscr{D}[P_1, \ldots, P_k] \parallel R)$ and $1 \le i, j \le k$.

    By $P \vdash \Gamma$ and T-Cut, the bound endpoints $\{z, \bar{z}\}$ cannot be $\{x, \bar{x}\}$.

By induction, one of the following holds:

* $\mathscr{D}[P_1, ..., P_k] = \mathscr{E}'[(\nu x\bar{x})(\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j])]$; or
* $\mathscr{D}[P_1, ..., P_k] = \mathscr{E}'[(\nu \bar{x}x)(\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i])]$.

Let $\mathscr{E} = (\nu z\bar{z})(\mathscr{E}' \parallel R)$.

The result follows.

– P_i and P_j are both to the right.

As above.

□

The separation lemma is rather precise, and gives us one of two equalities. However, both cases are equivalent up to structural congruence.

Corollary 2.22 (Separation)**.** *If* $P \vdash \Gamma$*,* $\mathscr{C}^n$ *is a configuration context such that* $P = \mathscr{C}^n[P_1, ..., P_n]$ *(for some* $n \geq 2$*), and there exists some* $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$ *such that* $x \in \mathrm{fn}(P_i)$ *and* $\bar{x} \in \mathrm{fn}(P_j)$ *(for some* $1 \leq i, j \leq n$*), there exist* $\mathscr{E}$*,* $\mathscr{F}_i$*, and* $\mathscr{F}_j$ *such that* $P \overset{\text{LS}}{\equiv} \mathscr{E}[(\nu x\bar{x})(\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j])]$*.*

Proof. By Lemma 2.21 and SC-CutComm.

□

Evaluation contexts commute with cuts.

Lemma 2.23. *If* $\mathrm{fn}(P) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$*, then* $\mathscr{E}[(\nu x\bar{x})(P \parallel Q)] \overset{\text{LS}}{\equiv} (\nu x\bar{x})(P \parallel \mathscr{E}[Q])$*.*

Proof. By induction on the structure of the evaluation context $\mathscr{E}$.

□

2.2.2 Preservation

Structural congruences preserve typing. If some process P is well-typed and is equivalent to some process Q under structural congruence, then Q is well-typed under the same typing environment.

Lemma 2.24. *If* $P \equiv Q$*, then* $P \vdash \Gamma$ *if and only if* $Q \vdash \Gamma$*.*

Proof. By induction on the derivation of the equivalence $P \equiv Q$.

The case for reflexivity follows immediately. The cases for symmetry and transitivity follow immediately by induction. The case for congruence closure follows by induction and the injectivity of the type derivation rules. The cases for applications of SC-LinkComm, SC-CutComm, and SC-CutAssoc are as follows, presented as equivalences on type derivations:

- Case SC-LinkComm:

$$\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}} \equiv \frac{\dfrac{}{y{\leftrightarrow}x \vdash y : \overline{A}, x : \overline{\overline{A}}}}{y{\leftrightarrow}x \vdash y : \overline{A}, x : A} \text{ Lemma 2.1}$$

- Case SC-CutComm:

$$\frac{P \vdash \Gamma_1, x : A \qquad Q \vdash \Gamma_2, \bar{x} : \overline{A}}{(\nu x\bar{x})(P \parallel Q) \vdash \Gamma_1, \Gamma_2}$$

$$\equiv$$

$$\frac{Q \vdash \Gamma_2, \bar{x} : \overline{A} \qquad \dfrac{P \vdash \Gamma_1, x : \overline{\overline{A}}}{P \vdash \Gamma_1, x : A}}{(\nu \bar{x}x)(Q \parallel P) \vdash \Gamma_1, \Gamma_2} \text{ Lemma 2.1}$$

- Case SC-CutAssoc:

$$\frac{\dfrac{P \vdash \Gamma_1, x : A, y : B \qquad Q \vdash \Gamma_2, \bar{y} : \overline{B}}{(\nu y\bar{y})(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : A} \qquad R \vdash \Gamma_3, \bar{x} : \overline{A}}{(\nu x\bar{x})((\nu y\bar{y})(P \parallel Q) \parallel R) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

$$\equiv$$

$$\frac{\dfrac{P \vdash \Gamma_1, x : A, y : B \qquad R \vdash \Gamma_3, \bar{x} : \overline{A}}{(\nu x\bar{x})(P \parallel R) \vdash \Gamma_1, \Gamma_3, y : B} \qquad Q \vdash \Gamma_2, \bar{y} : \overline{B}}{(\nu y\bar{y})((\nu x\bar{x})(P \parallel R) \parallel Q) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

The above derivation for the left-hand side (symmetrically, right-hand side) is the only derivation, as $x \notin Q$ (symmetrically, $y \notin R$).

□

Renaming preserves typing. If a process is well-typed, then renaming any free endpoint does not affect its typing.

Lemma 2.25. *If* $P \vdash \Gamma, x : A$, *then* $P\{w/x\} \vdash \Gamma, w : A$.

Proof. The result follows by induction.

□

Plugging with any form of process context preserves typing.

Lemma 2.26. *If* $P^n[\cdot] \vdash \Gamma_1 \mid \cdots \mid \Gamma_n \to \Gamma$ *and* $Q_i \vdash \Gamma_i$ *(for* $1 \le i \le n$*), then* $P^n[Q_1, \ldots, Q_n] \vdash \Gamma$.

Proof. By induction on the derivation of $P^n[\cdot] \vdash \Gamma_1 \mid \cdots \mid \Gamma_n \to \Gamma$.

□

Reduction preserves typing. If a process P is well-typed and reduces to some other process Q, then Q is well-typed under the same typing environment.

Proposition 2.27 (Preservation)**.** *If* $P \vdash \Gamma$ *and* $P \longrightarrow Q$*, then* $Q \vdash \Gamma$.

Proof. By induction on the derivation of the reduction $P \longrightarrow Q$.

The cases for E-Cong follow by induction and Lemma 2.26. The case for E-Equiv follows by induction and Lemma 2.24. The cases for the rules E-Link, E-Send, E-Close, E-Select$_1$, and E-Select$_2$ are as follows, presented as reductions on type derivations:

- Case E-Link:

$$\dfrac{x{\leftrightarrow}w \vdash x : A, w : \overline{A} \qquad P \vdash \Gamma, \bar{x} : \overline{A}}{(\nu x\bar{x})(x{\leftrightarrow}w \parallel P) \vdash \Gamma, w : \overline{A}}$$

$$\downarrow$$

$$\dfrac{P \vdash \Gamma, \bar{x} : \overline{A}}{P\{w/\bar{x}\} \vdash \Gamma, w : \overline{A}}\ \text{Lemma 2.25}$$

- Case E-Send:

$$\dfrac{\dfrac{P \vdash \Gamma_1, y : A \qquad Q \vdash \Gamma_2, x : B}{x[y].\,(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : A \otimes B} \qquad \dfrac{R \vdash \Gamma_3, \bar{y} : \overline{A}, \bar{x} : \overline{B}}{\bar{x}(\bar{y}).\,R \vdash \Gamma_3, \bar{x} : \overline{A} \parr \overline{B}}}{(\nu x\bar{x})(x[y].\,(P \parallel Q) \parallel \bar{x}(\bar{y}).\,R) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

$$\downarrow$$

$$\dfrac{P \vdash \Gamma_1, y : A \qquad \dfrac{Q \vdash \Gamma_2, x : B \qquad R \vdash \Gamma_3, \bar{y} : \overline{A}, \bar{x} : \overline{B}}{(\nu x\bar{x})(Q \parallel R) \vdash \Gamma_2, \Gamma_3, \bar{y} : \overline{A}}}{(\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R)) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

- Case E-Close:

$$\dfrac{\dfrac{}{x[].\,0 \vdash x : \mathbf{1}} \qquad \dfrac{Q \vdash \Gamma_2}{\bar{x}().\,Q \vdash \Gamma_2, \bar{x} : \bot}}{(\nu x\bar{x})(x[].\,0 \parallel \bar{x}().\,Q) \vdash \Gamma_2} \longrightarrow Q \vdash \Gamma_2$$

- Case E-Select$_1$:

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : A}{x \triangleleft \mathrm{inl}.\,P \vdash \Gamma_1, x : A \oplus B} \qquad \dfrac{Q \vdash \Gamma_2, \bar{x} : \overline{A} \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{\bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\} \vdash \Gamma_2, \bar{x} : \overline{A} \mathbin{\&} \overline{B}}}{(\nu x\bar{x})(x \triangleleft \mathrm{inl}.\,P \parallel \bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\}) \vdash \Gamma_1, \Gamma_2}$$

$$\downarrow$$

$$\dfrac{P \vdash \Gamma_1, x : A \qquad Q \vdash \Gamma_2, \bar{x} : \overline{A}}{(\nu x\bar{x})(P \parallel Q) \vdash \Gamma_1, \Gamma_2}$$

- Case E-SELECT$_2$:

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : B}{x \triangleleft \mathrm{inr}.\, P \vdash \Gamma_1, x : A \oplus B} \quad \dfrac{Q \vdash \Gamma_2, \bar{x} : \overline{A} \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{\bar{x} \triangleright \{\mathrm{inl}: Q; \mathrm{inr}: R\} \vdash \Gamma_2, \bar{x} : \overline{A} \,\&\, \overline{B}}}{(\nu x \bar{x})(x \triangleleft \mathrm{inr}.\, P \parallel \bar{x} \triangleright \{\mathrm{inl}: Q; \mathrm{inr}: R\}) \vdash \Gamma_1, \Gamma_2}$$

$$\Downarrow$$

$$\dfrac{P \vdash \Gamma_1, x : B \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{(\nu x \bar{x})(P \parallel R) \vdash \Gamma_1, \Gamma_2}$$

□

2.2.3 Progress

What should the canonical forms be for processes in CP? The usual starting point would be to ask what the canonical forms are for *closed* processes. For instance, in the λ-calculus, a closed process of the function type must be a lambda, and in the π-calculus, a closed process must be equivalent to the terminated process. Alas, that question is meaningless for CP, which *has no closed processes*. Any closed process would correspond to a proof of the empty sequent, which is not provable in CLL. A CP process is never *done*. There is no terminated process. There are only processes that are temporarily stuck, blocked on a free endpoint—one endpoint of a channel whose other endpoint has not yet been connected. Unfortunately, it is not sufficient to simply require that the process contains an action on a free endpoint. For instance, the process

$$(\nu x \bar{x})(a().\, P \parallel (\nu y \bar{y})(y[].\, 0 \parallel \bar{y}().\, Q))$$

has an action that is blocked on an external channel, but it can still reduce. Hence, the definition of canonical form should also capture the fact that no further reduction rules can be applied. Unsurprisingly, that condition is in and of itself sufficient, so let us formalise it as a starting point.

The simplest and most useless definition would be to say that P is in canonical form if and only if $P \not\longrightarrow$. However, it would be more satisfying if we could characterise the conditions under which no reduction rules apply.

First, note that E-LINK acts differently from most reduction rules. Most reduction rules, i.e. E-SEND, E-CLOSE, E-SELECT$_1$, and E-SELECT$_2$, are synchronous interactions between two processes and are blocked unless both processes are ready to act on dual endpoints. However, link is *asynchronous*. Any ready link can evaluate to a renaming in the process it communicates with, regardless of whether that process is ready to act on the relevant channel.

To capture this difference, I divide reduction into α-reduction, which captures link evaluating as α-renaming, and β-reduction, which captures all other reduction. In essence, α-reduction captures asynchronous reduction, and β-reduction captures synchronous reduction.

Definition 2.28 (α-Reduction)**.** *A process* P *α-reduces to* Q*, written* $P \longrightarrow_\alpha Q$*, if there exists a reduction* $P \longrightarrow Q$ *which only uses the rules E-Link, E-Cong, and E-Equiv.*

Definition 2.29 (β-Reduction)**.** *A process* P *β-reduces to* Q*, written* $P \longrightarrow_\beta Q$*, if there exists a reduction* $P \longrightarrow Q$ *that does not use the rule E-Link.*

Any reduction is either an α-reduction or a β-reduction.

I could say that P is in canonical form if and only if $P \not\longrightarrow_\alpha$ and $P \not\longrightarrow_\beta$. However, I have already given a tighter characterisation above. A process P is in canonical form if and only if

1. P contains no link thread ready to act on a bound endpoint; and
2. P contains no two threads ready to act on dual endpoints.

If (1) does not hold, E-Link applies, and if (2) does not hold, at least one of the other reduction rules applies… *and that is progress in a nutshell!*

Definition 2.30 (Canonical Form)**.** *A process* P *is in* canonical form*, written* canonical(P)*, if* P *is of the form* $\mathscr{C}^n[T_1,\ldots,T_n]$ *(for some* $n \geq 1$*) and (for* $1 \leq i, j \leq n$*)*

1. *no* T_i *is a link thread ready to act on an endpoint* $x \in \mathrm{bn}(\mathscr{C})$*; and*
2. *no* T_i *and* T_j *are ready to act on dual endpoints* $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$*.*

If condition (1) holds, $P \not\longrightarrow_\alpha$*. If condition (2) holds,* $P \not\longrightarrow_\beta$*.*

If a process contains two threads ready to act on dual endpoints, then it can reduce. The following lemma abstracts over the reduction rules for CP's many dual actions.

Lemma 2.31 (Reduction)**.** *If* $(\nu x\bar{x})(P \parallel Q) \vdash \Gamma$*, and* P *and* Q *are ready to act on* x *and* $\bar{x}$*, respectively, there exists some* R *such that* $(\nu x\bar{x})(P \parallel Q) \longrightarrow R$*.*

Proof. By inversion on the derivation of $(\nu x\bar{x})(P \parallel Q) \vdash \Gamma$. There are eight cases, which correspond exactly to E-Send, E-Close, E-Select$_1$, E-Select$_2$, and the variants under E-Equiv with SC-CutComm.

□

Progress states that any process is either in canonical form or can reduce. In essence, the proof shows that conditions (1) and (2) of the definition of canonical form correspond to the absence of α- and β-reduction.

Proposition 2.32 (Progress)**.** *If* $P \vdash \Gamma$*, then either* P *is in canonical form, or there exists some* Q *such that* $P \longrightarrow Q$*.*

Proof. P is of the form $\mathscr{C}^n[T_1,\ldots,T_n]$ (for some $n \geq 1$).

If P is in canonical form, the result follows.

Otherwise, $n \geq 2$, and there are two cases:

Condition (1) does not hold. Some T_i (for $1 \leq i \leq n$) is a link thread ready to act on an endpoint $x \in \mathrm{bn}(\mathscr{C})$. By definition, there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$. By Corollary 2.22 and Proposition 2.18, there exists some P_j (for $1 \leq j \leq n$ and $i \neq j$) such that $\bar{x} \in \mathrm{fn}(P_j)$.

$$
\begin{array}{lll}
P & \overset{s}{\equiv} \mathscr{E}[(\nu x\bar{x})(\mathscr{F}_1[x{\leftrightarrow}y] \parallel \mathscr{F}_2[P_j])] & \langle\text{by Corollary 2.22 and SC-LinkComm}\rangle \\
 & \overset{s}{\equiv} \mathscr{E}[\mathscr{F}_1[(\nu x\bar{x})(x{\leftrightarrow}y \parallel \mathscr{F}_2[P_j])]] & \langle\text{by Lemma 2.23}\rangle \\
 & \overset{s}{\equiv} \mathscr{E}[\mathscr{F}_1[\mathscr{F}_2[(\nu x\bar{x})(x{\leftrightarrow}y \parallel P_j)]]] & \langle\text{by Lemma 2.23}\rangle \\
 & \longrightarrow \mathscr{E}[\mathscr{F}_1[\mathscr{F}_2[P_j\{y/\bar{x}\}]]] & \langle\text{by E-Link and E-Cong}\rangle
\end{array}
$$

Condition (2) does not hold. Some threads T_i and T_j (for $1 \leq i, j \leq n$) are ready to act on dual endpoints $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$.

$$
\begin{array}{lll}
P & \overset{s}{\equiv} \mathscr{E}[(\nu x\bar{x})(\mathscr{F}_i[T_i] \parallel \mathscr{F}_j[T_j])] & \langle\text{by Corollary 2.22}\rangle \\
 & \overset{s}{\equiv} \mathscr{E}[\mathscr{F}_i[(\nu x\bar{x})(T_i \parallel \mathscr{F}_j[T_j])]] & \langle\text{by Lemma 2.23}\rangle \\
 & \overset{s}{\equiv} \mathscr{E}[\mathscr{F}_i[\mathscr{F}_j[(\nu x\bar{x})(T_i \parallel T_j)]]] & \langle\text{by Lemma 2.23}\rangle \\
 & \longrightarrow \mathscr{E}[\mathscr{F}_i[\mathscr{F}_j[R]]] & \langle\text{by Lemma 2.31 and E-Cong}\rangle
\end{array}
$$

□

2.2.4 Duality, Dependency, and Deadlock

The definition of canonical form (Definition 2.30) requires justification: it defines "canonical forms" as "processes that do not reduce", which is an easy way to get in trouble by admitting processes as "canonical" when they are, in fact, stuck for undesirable reasons such as deadlock.

It is straightforward to argue that processes in canonical form *must* contain a process ready to act on a free endpoint, which puts Definition 2.30 on par with Wadler's definition of canonical form (see § 2.3) and Lindley and Morris' definition of blocking. The argument relies on the simple combinatorics of configuration contexts. Any configuration context with n holes must contain exactly $n - 1$ cuts, and therefore must create exactly $n - 1$ channels.

Lemma 2.33. *For any* $\mathscr{C}^n$, $|\mathrm{dn}(\mathscr{C}^n)| = n - 1$ *and* $|\mathrm{bn}(\mathscr{C}^n)| = 2(n - 1)$.

Proof. By induction on the structure of the configuration context $\mathscr{C}$.

□

The fact that *some* process must be ready to act on a free endpoint follows, since you cannot have n processes ready to act on $n - 1$ channels without having at least two of them ready to act on the same channel.

Proposition 2.34 (Canonical Form)**.** *Any well-typed process* P *in canonical form contains a thread that is ready to act on a free endpoint.*

Proof. By contradiction. We have P is of the form $\mathscr{C}^n[T_1,\ldots,T_n]$ (for some $n \geq 1$). Assume no thread T_i is ready to act on a free endpoint. Then all n threads T_1, ..., T_n must be ready to act on bound endpoints. By linearity, all n threads T_1, ..., T_n must act on distinct bound endpoints. By Lemma 2.33, $\mathscr{C}^n$ binds $2(n-1)$ endpoints in $n-1$ dual pairs, so any subset of n bound endpoints must contain at least one dual pair. Therefore, at least two threads must be ready to act on dual endpoints, which contradicts the premise that P is in canonical form.

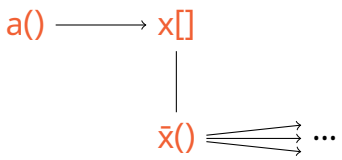
□

Unfortunately, that characterisation is *inadequate*, since it does not guarantee that the process cannot reduce. An adequate definition should to match the intuition that *all communication* in processes that cannot reduce is blocked on free endpoints. To formalise this notion, we must ensure that every ready action is blocked on an action on (1) a free endpoint or (2) a bound endpoint whose dual depends on some ready action that is blocked. For instance, in the process

$$(\nu x\bar{x})(a().\, x[].\, 0 \parallel \bar{x}().\, P)$$

the action a() is blocked on the free endpoint a, but the action $\bar{x}()$ is blocked because its dual action, x[], depends on the action a(), which is blocked on the free endpoint a.

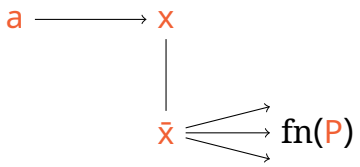
I formalise the notion of one action *depending* on another as the *shallow dependency graph* of a process. The dependency graph is a mixed graph, where undirected edges represent connected endpoints, either by link or by cut, and directed edges—or arcs—represent sequential dependencies. For instance, the shallow dependency graph for the above process is



where the arrows out of $\bar{x}()$ connect to the actions in P. The graph is *shallow* because it only tracks dependencies up to the *first action*, e.g. any dependencies within P are not tracked.

To define the dependency graph, we need some way to uniquely refer to actions. Unfortunately, the actions themselves are not unique—consider, e.g. $x \triangleleft \mathsf{inl}.\, x \triangleleft \mathsf{inl}.\, x[].\, 0$. For shallow dependencies, it suffices to use endpoint names to refer to the first action on that endpoint. The

dependency graph of the above process then becomes



For the *deep* dependency graph, we could augment names with indices tracking the usage, e.g. letting $(x \triangleleft \mathrm{inl}, 1)$ and $(x \triangleleft \mathrm{inl}, 0)$ refer to the first and second occurrence of $x \triangleleft \mathrm{inl}$ in $x \triangleleft \mathrm{inl}.\, x \triangleleft \mathrm{inl}.\, x[].\, 0$. Alternatively, we could assign fresh vertices to each action (as is done in Priority CP, see Chapter 5). Fortunately, the shallow variants suffice for my purposes in this chapter.

The dependency graph is a mixed graph. I informally revisit the relevant definitions. For a detailed discussion, see § A.1.

- A *mixed graph* G has a set of vertices (denoted V_G, ranged over by u, v), a set of edges (denoted E_G), a set of arcs (denoted A_G). Edges are unordered pairs denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$. The set of edges may not contain loops uu. Arcs are ordered pairs denoted by juxtaposition overset with an arrow to indicate the direction, i.e. $\overrightarrow{uv} \triangleq (u, v)$. The set of arcs may not contain loops $\overrightarrow{uu}$.
- For any graph G with vertices $u, v \in V_G$, u is *adjacent* to v when there exists some edge $uv \in E_G$ or some arc $\overrightarrow{uv} \in A_G$.
- A *walk* w is a sequence of pairwise adjacent vertices.
- A *path* p is a walk with no repeated vertices, except possibly the first and last.
- A *cycle* c is a path that begins and ends at the same vertex.
- A walk is *essentially directed* when it contains at least one arc.
- A graph is *essentially acyclic* if and only if it contains no essentially directed cycles.
- The *undirected reachability* relation (denoted by $\sim_G$) is the equivalence closure over E_G.
- The *essentially directed reachability* relation (denoted by $<_G$) is the transitive closure over A_G quotiented by $\sim_G$.

The vertices of the dependency graph are endpoint names, which are a proxy for the first action on that endpoint. The edges represent channels, created by either links or cuts. The arcs represent dependencies, created by prefixing. For instance, in $a().\, b[].\, 0$, the process $b[].\, 0$ is prefixed with the action $a()$. Hence, the action on b depends on the action on a.

Definition 2.35 (Dependency graph)**.** *The* shallow dependency graph *of a process* P*, written* $\mathrm{Dep}(P)$*, is a mixed graph (see § A.1) defined by the structure of the process* P*. The process* P *is of the form* $\mathscr{C}^n[T_1, \ldots, T_n]$ *(for*

some $n \geq 1$). The shallow dependency graph $\mathrm{Dep}(P)$ *is defined as:*

$$V_{\mathrm{Dep}(P)} \triangleq \bigcup_{1 \leq i \leq n} \mathrm{fn}(T_i)$$
$$E_{\mathrm{Dep}(P)} \triangleq \bigcup_{1 \leq i \leq n} \{xy \mid T_i = x \leftrightarrow y\} \cup \{x\bar{x} \mid \{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})\}$$
$$A_{\mathrm{Dep}(P)} \triangleq \bigcup_{1 \leq i \leq n} \{\overrightarrow{xy} \mid x, y \in \mathrm{fn}(T_i), \mathrm{ready}(T_i, x) \wedge \neg\, \mathrm{ready}(T_i, y)\}$$

By Lemma 2.21, $E_{\mathrm{Dep}(P)}$ and $A_{\mathrm{Dep}(P)}$ contain no loops. If G is (the subgraph of) some dependency graph, I write $\mathrm{fn}(G)$ *for its vertices, i.e.* $\mathrm{fn}(G) = V_G$.

The dependency graph gives us *duality* on actions, which is undirected reachability in the dependency graph.

Definition 2.36 (Duality)**.** *An endpoint x is* dual *to some endpoint y in P, written $x \sim_P y$, if and only if there exists an undirected path from x to y in* $\mathrm{Dep}(P)$.

If $x \sim_P y$, the corresponding path in $\mathrm{Dep}(P)$ may be arbitrarily long, as undirected edges arise from both cuts and links. Consider the process

$$(\nu x\bar{x})(a \leftrightarrow x \parallel \bar{x} \leftrightarrow b)$$

The duality $a \sim b$ is witnessed by the path $(ax, x\bar{x}, \bar{x}b)$. The structure generated by cuts and links does not fork, i.e. each component of the undirected subgraph of the dependency graph is a path.

The dependency graph also gives us *dependency* on actions, which is the converse of essentially directed reachability in the dependency graph.

Definition 2.37 (Dependency)**.** *An endpoint x* depends *on some endpoint y, written $x >_P y$, if and only if there exists an essentially directed path from y to x in* $\mathrm{Dep}(P)$.

One quirk of using endpoints as a proxy for actions is that the duality and dependency appear to "leak" restricted names, i.e. $\sim_P$ and $>_P$ are not relations over $\mathrm{fn}(P)$, but relations over $\mathrm{fn}(P) \cup \mathrm{bn}(\mathscr{C})$, where $\mathscr{C}$ is the maximum configuration context of P. However, as discussed, these relations should be viewed as relations on the first actions on those endpoints, not the endpoints themselves.

A process is in deadlock if the dependency relation is not antisymmetric, or, equivalently, if there is an essentially directed cycle in the dependency graph that contains at least one arc.

Definition 2.38 (Deadlock)**.** *A process P is in deadlock, written* $\mathrm{deadlock}(P)$*, if* $\mathrm{Dep}(P)$ *contains an essential cycle.*

Every well-typed CP process is deadlock-free.

Proposition 2.39. *If $P \vdash \Gamma$, then* $\neg\, \mathrm{deadlock}(P)$.

Proof. By induction on the derivation of $P \vdash \Gamma$.

- Case T-Link.

  The process P is of the form $x \leftrightarrow y$.

  $\mathrm{Dep}(x \leftrightarrow y)$ is essentially acyclic as it has no arcs.

- Case T-Cut.

  The process P is of the form $(\nu x\bar{x})(P_1 \parallel P_2)$. By induction $\mathrm{Dep}(P_1)$ and $\mathrm{Dep}(P_2)$ are essentially acyclic.

  By T-Cut, $\mathrm{fn}(P_1) \cap \mathrm{fn}(P_2) = \varnothing$. Under the Barendregt convention, all bound names in P_1 and P_2 are distinct. Hence, $\mathrm{Dep}(P_1)$ and $\mathrm{Dep}(P_2)$ are disjoint.

  $\mathrm{Dep}(P)$ is essentially acyclic by Lemma A.2.

- Case T-Send, T-Recv, T-Close, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, or T-Absurd.

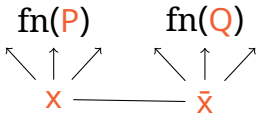
  The process P is ready, i.e. $\mathrm{ready}(P, x)$ for some endpoint x.

  $\mathrm{Dep}(P)$ is essentially acyclic as it has no edges and only arcs out of x.

□

The definition of deadlock is *shallow*. If a process is "free from deadlock", that means the process is not in *immediate deadlock*. However, it does not mean that the process can never become deadlocked. Fortunately, the latter follows by type preservation. Since well-typed processes are free from immediate deadlock, and reduction preserves types, no well-typed process can ever reduce to a deadlocked process.

Care should be taken to only use Definition 2.38 for well-typed processes, as it does not imply session fidelity, the property that dual endpoints are used in dual ways. Hence, there are ill-typed processes that are morally in deadlock, but whose dependency graphs are essentially acyclic. For instance, the ill-typed process $(\nu x\bar{x})(x().\, P \parallel \bar{x}().\, Q)$ is *morally* in deadlock, but its dependency graph is essentially acyclic:



A *blocking action* is an action that blocks a process from making progress. For instance, in the process

$$(\nu x\bar{x})(a().\, x[].\, 0 \parallel \bar{x}().\, P)$$

the action $a()$ is *blocking*. However, not every ready action is *blocking*. The action $\bar{x}()$ is ready, but not blocking. Rather, it is *blocked*: its dual $x[]$ depends on $a()$, so it cannot reduce until $a()$ does.

As with dependency, I approximate blocking actions by their endpoints. *Blocking endpoint* are the maxima of the dependency relation, or, equivalently, the sources of the dependency graph. The blocking set of a process is the set of all sources of its dependency graph. Every ready action in a process is blocked on one of the endpoints in the blocking set.

Definition 2.40 (Blocking Set)**.** *The* blocking set *of endpoints of a process* P*, written* blocking(P)*, is the set of sources of* Dep(P)*, i.e.* $\{x \in V_{\mathrm{Dep}(P)} \mid \nexists y. x >_P y\}$.

The blocking set is closed under duality.

Lemma 2.41. *If* $P \vdash \Gamma$ *and* $x \sim_P y$*, then* $x \in \mathrm{blocking}(P) \implies y \in \mathrm{blocking}(P)$.

Each endpoint in the blocking set corresponds to a ready action.

Lemma 2.42. *If* $P \vdash \Gamma$ *and* $x \in \mathrm{blocking}(P)$*, then* P *is of the form* $\mathscr{E}[T]$ *such that* ready(T, x).

Due to the dualities generated by links, the blocking set may contain more endpoints than necessary. For instance, the blocking set of the process

$$(\nu x\bar{x})(x{\leftrightarrow}a \parallel \bar{x}().\, P)$$

is $\{x, \bar{x}, a\}$. An action in P is blocked on all of these endpoints. However, I want to be able to say that every action is blocked on a free name, and, in this case, the set $\{a\}$ suffices. A process is blocked on a set of endpoints if any action is blocked on at least one endpoint in that set.

Definition 2.43 (Blocked)**.** *A process* P *is* blocked *on a set of endpoints* X*, written* blocked(P, X)*, if closing* X *under duality yields the blocking set* blocking(P)*, i.e. if* $x \in X$ *and* $x \sim_P y$*, then* $y \in \mathrm{blocked}(P)$.

Any process is blocked on its blocking set.

Lemma 2.44. *If* $P \vdash \Gamma$*, then* blocked(P, blocking(P)).

If a process is blocked on some set of endpoints, it is blocked on the set formed by replacing any endpoint in that set with its dual.

Lemma 2.45. *If* $P \vdash \Gamma$ *and* $x \sim_P y$*, then* $\mathrm{blocked}(P, X) \implies \mathrm{blocked}(P, X\{y/x\})$.

If a process cannot β-reduce, then it is blocked on some set of free names.

Proposition 2.46. *If* $P \vdash \Gamma$*, then* $P \not\rightarrow_\beta \iff \exists A \subseteq \mathrm{fn}(P).\, \mathrm{blocked}(P, A)$.

Proof. Let P be of the form $\mathscr{C}^n[T_1,\ldots,T_n]$ (for some $n \geq 1$).

There are two cases:

- Case ($\Rightarrow$).

  By contradiction. Assume $x \in \mathrm{blocking}(P)$ and $\nexists a \in \mathrm{fn}(P). x \sim_P a$.

  There are two cases:

- If $x \in \mathrm{fn}(P)$, then $x \sim_P x$.
- If $x \in \mathrm{bn}(\mathscr{C})$, then there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$.

  There are two cases:

  * If $\bar{x} \in \mathrm{blocking}(P)$, then there exist threads T_i and T_j that are ready to act on dual endpoints $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C})$. By Lemma 2.31, P is not β-free.
  * If $\bar{x} \notin \mathrm{blocking}(P)$, then there exists some y such that $\bar{x} >_P y$. By definition, $x\bar{x} \in E_{\mathrm{Dep}(P)}$. Therefore, $x >_P y$ and $x \notin \mathrm{blocking}(P)$.

- Case ($\Leftarrow$).

  By contradiction. Assume $P \longrightarrow_{\beta}$.

  By inversion, there exist some T_i and T_j (for $1 \leq i, j \leq n$) that are ready to act on dual endpoints x and $\bar{x}$.

  By definition, x and $\bar{x}$ only have outgoing arcs in $\mathrm{Dep}(P)$, and the only edge connected to either is $x\bar{x}$. Therefore, $\{x, \bar{x}\} \subseteq \mathrm{blocking}(P)$ and $\nexists a \in \mathrm{fn}(P).x \sim_P a \vee \bar{x} \sim_P a$.

□

Corollary 2.47. *If* $P \vdash \Gamma$, *then* $\mathrm{canonical}(P) \iff \mathrm{blocking}(P) \subseteq \mathrm{fn}(P)$.

Proof. By Proposition 2.46, there exists a set $A \subseteq \mathrm{fn}(P)$ such that $\mathrm{blocked}(P, A)$. By definition, $A \subseteq \mathrm{blocking}(P)$. It remains to show that $\mathrm{blocking}(P) \subseteq A$.

By contradiction. Assume $x \in \mathrm{blocking}(P)$ and $x \notin A$.

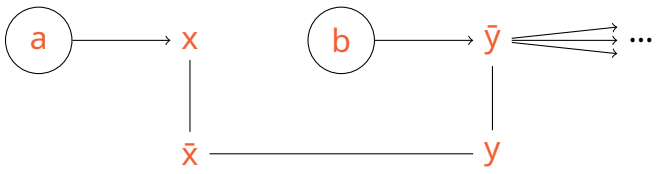
By definition, there exists some $a \in A$ such that $x \sim_P a$. The duality $x \sim_P a$ corresponds to some undirected path $p_{xa} = (x, \ldots, a)$ in $\mathrm{Dep}(P)$, which must contain at least one edge that connects some *bound* name y to some *free* name b, say, yb. By definition, any edge in $\mathrm{Dep}(P)$ generated by a cut connects two *bound* names. Therefore, yb must be generated by a link. By Lemma 2.42, $P = \mathscr{E}[y \leftrightarrow b]$. Hence, $P \longrightarrow_{\alpha}$.

□

Unfortunately, "blocked on free endpoints" does not characterise canonical forms, as $\mathrm{blocking}(P) \subseteq \mathrm{fn}(P) \not\Rightarrow P \not\longrightarrow_{\alpha}$. For instance, the process

$$(\nu x\bar{x})(a().\, x[].\, 0 \parallel (\nu y\bar{y})(\bar{x} \leftrightarrow y \parallel b().\, \bar{y}().\, P))$$

can α-reduce. Its dependency graph, with blocking endpoints circled, is



In conclusion, my definition of canonical form (Definition 2.30) is *adequate*: any process in canonical form is blocked on some set of free endpoints. Due to the behaviour of links, "blocked on free endpoints" is insufficient to characterise canonical forms. As this would be a desirable property to have, I consider an alternative semantics for the link construct when discussing HCP in § 3.3. I have not adopted any of these alternatives as standard to maintain backwards compatibility with the work based on Wadler's CP.

2.2.5 Connection and Right-Branching Form

In this section, I formalise the notion of the *connection graph* of a process, and show that any CP process can be rewritten into right-branching form.

Definition 2.48 (Right-branching Form)**.** *A process* P *is in* right-branching form *if* P *is ready or* P *is of the form* $(\nu x_1\bar{x}_1)(T_1 \parallel \cdots(\nu x_n\bar{x}_n)(T_n \parallel T_{n+1})\cdots)$ *for some* $n \geq 1$.

As discussed earlier, right-branching form is often used in the literature, but my metatheory avoids it, since it does not generalise to variants of CP that relax its rigid connection structure. So why am I talking about it? Of course, it is nice to justify a piece of reasoning that is frequently used. However, as it turns out, the theory I develop for converting processes to right-branching form will be important in the discussion of HCP.

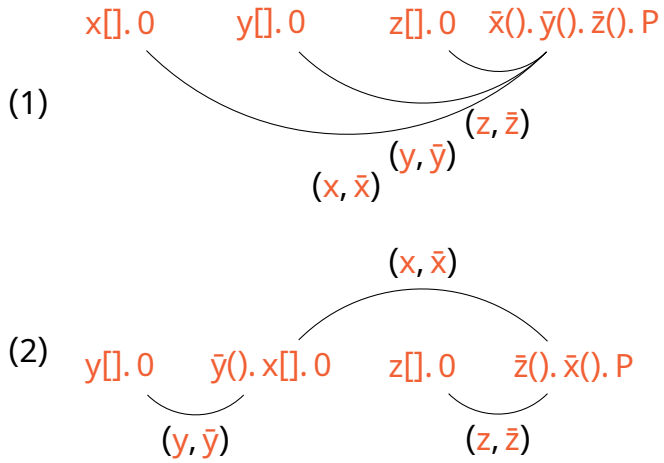
Let us start by examining the connection graphs and right-branching forms of two simple example processes:

(1) $(\nu x\bar{x})(\, x[].\, 0 \parallel (\nu y\bar{y})(\, y[].\, 0 \parallel (\nu z\bar{z})(\, z[].\, 0 \parallel \bar{x}().\, \bar{y}().\, \bar{z}().\, P\,)\,)\,)$

(2) $(\nu x\bar{x})(\, (\nu y\bar{y})(\, y[].\, 0 \parallel \bar{y}().\, x[].\, 0\,) \parallel (\nu z\bar{z})(\, z[].\, 0 \parallel \bar{z}().\, \bar{x}().\, P\,)\,)$

The connection graph of a process is the graph formed of all ready sub-processes and the channels that connect them. In the case of the example processes, each one consists of four ready sub-processes which

are connected by three channels:



Process (1) is already in right-branching form, though we could reorder the first three processes. Process (2) is not yet in right-branching form, but we can use the connection graph to convert it to right-branching form. The procedure picks a leaf from the connection graph, moves the corresponding cut and thread to the topmost, leftmost position, removes the leaf, and continues until all of the graph is empty. The process below is one of numerous different right-branching forms:

$$(\nu y\bar{y})(\ y[].\,0 \ \parallel\ (\nu x\bar{x})(\ \bar{y}().\,x[].\,0 \ \parallel\ (\nu z\bar{z})(\ z[].\,0 \ \parallel\ \bar{z}().\,\bar{x}().\,P\ )\ )\ )$$

Our notion of connection graph is *shallow*, as opposed to deep, as we only account for the connections up to the maximum configuration context. However, the shallow definition suffices for our purposes and, as we shall see in Chapter 3, can be used to reason about the deep connection graph of a process by reasoning about the collection of shallow connection graphs of all sub-processes.

The connection graph is a undirected edge-labelled graph. I informally revisit the relevant definitions. For a detailed discussion, see § A.1.

- A *undirected edge-labelled graph* G has a set of vertices (denoted V_G, ranged over by u, v), a set of edges (denoted E_G), a set of edge labels (denoted $\mathscr{L}_G$), and an edge-labeling function (denoted ℓ_G) that assigns labels to edges. Edges are unordered pairs denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$. The set of edges may not contain loops uu.
  (It suffices to define V_G and ℓ_G, since $E_G \triangleq \mathrm{dom}(\ell_G)$ and $\mathscr{L}_G \triangleq \mathrm{cod}(\ell_G)$.)
- Two vertices $u, v \in V_G$ are *adjacent* when there exists an edge $uv \in E_G$.
- A *walk* w is a sequence of pairwise adjacent vertices.
- A *path* p is a walk with no repeated vertices, except possibly the first and last.
- A *cycle* c is a path that begins and ends at the same vertex.
- A graph is *acyclic* when it does not contain a cycle.

- A graph is *connected* when there is a path between any two vertices.
- A *tree* T is a graph that is connected and acyclic.

Definition 2.49 (Connection graph)**.** *The* shallow connection graph *of a well-typed process* P*, written* $\mathrm{Con}(P)$*, is an undirected edge-labelled graph (see § A.1) where the vertices are threads, the edges are the channels that connect those threads, and the edges are labelled by unordered pairs of their endpoints. The process* P *is of the form* $\mathscr{C}^n[T_1,\ldots,T_n]$ *(for some* $n \geq 1$*). The shallow connection graph* $\mathrm{Con}(P)$ *is defined as:*

$$V_{\mathrm{Con}(P)} \triangleq \{T_1, \ldots, T_n\}$$
$$\ell_{\mathrm{Con}(P)} \triangleq \{T_i T_j \mapsto (x, \bar{x}) \mid T_i, T_j \in V_{\mathrm{Con}(P)} \wedge \{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}) \wedge x \in T_i \wedge \bar{x} \in T_j\}$$

By Lemma 2.21, $E_{\mathrm{Con}(P)}$ *contains no loops.*

By Proposition 2.18, $\ell_{\mathrm{Con}(P)}$ *is a function.*

If G *is a subgraph of some connection graph, I write* $\mathrm{fn}(G)$ *for the free names in the vertices of* G*, i.e.* $\mathrm{fn}(G) \triangleq \bigcup_{P \in V_G} \mathrm{fn}(P)$*.*

In CP, the connection graph of a process is always a tree.

Proposition 2.50 (Connection Tree)**.** *If* P *is well-typed,* $\mathrm{Con}(P)$ *is a tree.*

Proof. By induction on the structure of $\mathscr{C}$ and inversion on the structure of P and the derivation of $P \vdash \Gamma$.

There are two cases:

- Case $\mathscr{C}$ is of the form $(\nu x \bar{x})(\mathscr{C}_1 \parallel \mathscr{C}_2)$.

  By inversion, the typing derivation is of the form

  $$\frac{Q \vdash \Gamma, x : A \qquad R \vdash \Delta, \bar{x} : \overline{A}}{(\nu x \bar{x})(Q \parallel R) \vdash \Gamma, \Delta}\ \text{T-Cut}$$

  The process P is of the form $(\nu x \bar{x})(Q \parallel R)$.

  By T-Cut, $\mathrm{fn}(Q)$ and $\mathrm{fn}(R)$ as well as $\mathrm{Con}(Q)$ and $\mathrm{Con}(R)$ are disjoint.

  By induction, $\mathrm{Con}(Q)$ and $\mathrm{Con}(R)$ are trees.

  The vertices of $\mathrm{Con}(P)$ are exactly the union of those of $\mathrm{Con}(Q)$ and $\mathrm{Con}(R)$. Furthermore, the unordered pair $\{x, \bar{x}\}$ is the only element of $\mathrm{dn}(\mathscr{C})$ that is not present in $\mathrm{dn}(\mathscr{C}_1)$ or $\mathrm{dn}(\mathscr{C}_1)$. By Proposition 2.18, there is exactly one $T_i \in V_{\mathrm{Con}(Q)}$ and one $T_j \in V_{\mathrm{Con}(R)}$ such that $x \in \mathrm{fn}(T_i)$ and $\bar{x} \in \mathrm{fn}(T_j)$. Therefore,

  $$V_{\mathrm{Con}(P)} = V_{\mathrm{Con}(Q)} \cup V_{\mathrm{Con}(R)}$$
  $$\ell_{\mathrm{Con}(P)} = \{T_i T_j \mapsto (x, \bar{x})\} \cup \ell_{\mathrm{Con}(Q)} \cup \ell_{\mathrm{Con}(R)}$$

  The result follows, as $\mathrm{Con}(P)$ is formed by connecting trees $\mathrm{Con}(Q)$ and $\mathrm{Con}(R)$ with the single edge $T_i T_j$, and connecting two trees with a single edge always yields another tree.

- Case $\mathscr{C}$ is of the form $\Box$.

  The result follows, as Con(P) is the singleton graph, which is a tree.

□

The tree structure of the connection graph can be used to rewrite any process to right branching form.

Proposition 2.51 (Right-branching Form)**.** *For any well-typed process* P*, there exists a process* Q *such that* $P \overset{\text{LS}}{\equiv} Q$ *and* Q *is in right-branching form.*

Proof. By Proposition 2.50, Con(P) is a tree. Let T_i be any leaf of Con(P) (for $1 \le i \le |V_{\text{Con(P)}}|$). As T_i is a leaf, exactly one free endpoint in T_i is bound in P. Let us name that endpoint x. There exist some $\mathscr{E}$, $\mathscr{F}$, and P′ such that:

$$\begin{aligned} P &\overset{\text{LS}}{\equiv} \mathscr{E}[(\nu x\bar{x})(\mathscr{F}[T_i] \parallel P')] && \langle\text{by Corollary 2.22}\rangle \\ &\overset{\text{LS}}{\equiv} \mathscr{E}[\mathscr{F}[(\nu x\bar{x})(T_i \parallel P')]] && \langle\text{by Lemma 2.23 and SC-CutComm}\rangle \\ &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(T_i \parallel \mathscr{E}[\mathscr{F}[P']]) && \langle\text{by Lemma 2.23}\rangle \end{aligned}$$

By Lemma 2.24, $\mathscr{E}[\mathscr{F}[P']]$ is well-typed. By induction on the process $\mathscr{E}[\mathscr{F}[P']]$, there exists some Q′ such that $\mathscr{E}[\mathscr{F}[P']] \overset{\text{LS}}{\equiv} Q'$ and Q′ is in right-branching form. Let Q be $(\nu x\bar{x})(T_i \parallel Q')$. The result follows, as $P \overset{\text{LS}}{\equiv} Q$ and Q is in right-branching form.

□

The tree structure of the connection graph implies deadlock freedom.

Proposition 2.52. *If* Con(P) *is a tree, then* ¬ deadlock(P)*.*

Sketch. Let us give the following definitions:

- P is of the form $\mathscr{C}^n[T_1, \ldots, T_n]$ (for some $n \ge 1$).
- T is the connection graph Con(P).
- G is the dependency graph Dep(P).
- E^C_G are the cut-edges in G, i.e. $\{x\bar{x} \mid \{x, \bar{x}\} \in \text{dn}(\mathscr{C})\}$.
- E^L_G are the link-edges in G, i.e. $\bigcup_{1 \le i \le n} \{xy \mid T_i = x \leftrightarrow y\}$.
- $G/_{A_G}$ is the quotient of G by its arcs A_G.

The result follows from the following facts:

1. Any essentially directed path p in G must alternate cut-edges with arcs or link-edges. (Cut-edges connect endpoints in different threads. Link-edges and arcs connect endpoints in the same thread. By definition, each ready link only generates a single link-edge, and every other thread only generates arcs out of the same vertex.)
2. There exists a surjective graph homomorphism from $G/_{A_G}$ to T, which preserves cut-edges and contracts the vertices connected by each link-edge, defined as:

$$f \triangleq \{X \mapsto T_i \mid 1 \le i \le n \wedge X \subseteq \text{fn}(T_i)\}$$

Assume P is in deadlock, i.e. there exists an essential cycle c in G.

There are three cases:

- If c contains no cut-edges, then, by fact (1), c must be a loop.
  By definition, G contains no loops.
- If c contains one cut-edge, then, by fact (1), T must contain a loop.
  By definition, T contains no loops.
- Otherwise, by fact (2), T contains a cycle.
  This contradicts our premise.

□

Connection graphs are not merely a tool for converting processes to right-branching form and proving deadlock freedom. They are a full-fledged alternative representation for processes, with the interesting property that they represent the maximum configuration context of a process without any spurious ambiguity. Processes with equivalent maximum configuration contexts have *equal* connection graphs. Moreover, the correspondence works both ways! Processes with equal connection graphs have equivalent maximum configuration contexts.

Proposition 2.53. *If* P *is well-typed, then* $\mathrm{P} \overset{\text{LS}}{\equiv} \mathrm{Q} \iff \mathrm{Con}(\mathrm{P}) = \mathrm{Con}(\mathrm{Q})$.

Proof. There are two cases:

- Case ($\Rightarrow$).

  By induction on the proof of the structural congruence $\mathrm{P} \overset{\text{LS}}{\equiv} \mathrm{Q}$. The cases for reflexivity, transitivity, symmetry, and SC-Cong follow by induction and those same properties of equality. The cases for SC-CutComm and SC-CutAssoc follow immediately.

- Case ($\Leftarrow$).

  Let Proc(T) be the set of processes in right-branching form obtained from the connection tree T of a well-typed process by Proposition 2.51. (This is a set because Proposition 2.51 defines a non-deterministic procedure.)

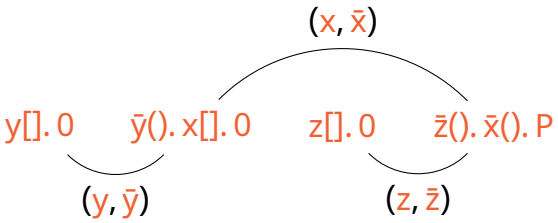
  Pick any $\mathrm{R} \in \mathrm{Proc}(\mathrm{Con}(\mathrm{P}))$. As $\mathrm{Con}(\mathrm{P}) = \mathrm{Con}(\mathrm{Q})$, $\mathrm{Proc}(\mathrm{Con}(\mathrm{P})) = \mathrm{Proc}(\mathrm{Con}(\mathrm{Q}))$. Hence, $\mathrm{R} \in \mathrm{Proc}(\mathrm{Con}(\mathrm{Q}))$. By definition, $\mathrm{P} \overset{\text{LS}}{\equiv} \mathrm{R}$ and $\mathrm{Q} \overset{\text{LS}}{\equiv} \mathrm{R}$. Hence, $\mathrm{P} \overset{\text{LS}}{\equiv} \mathrm{Q}$.

□

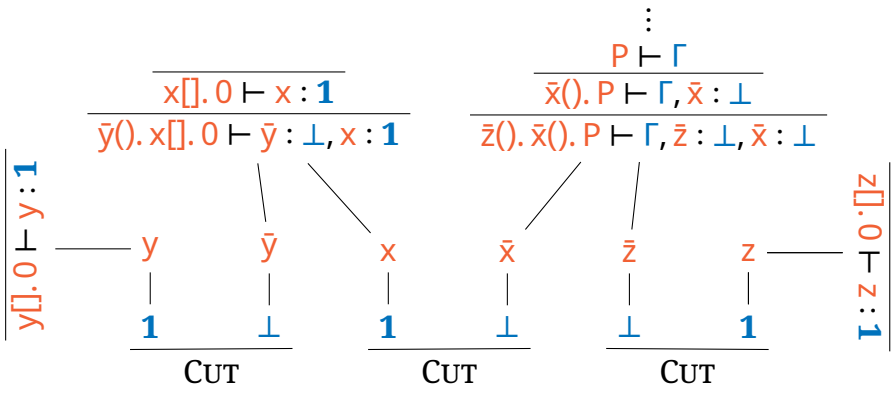
In conclusion, connection graphs are a unique representation for maximum configuration contexts that exactly capture link-preserving shallow structural congruence. That sounds awfully familiar. The canonical representation for Classical Linear Logic is *proof nets* [see Girard, 1987, p. 28]. Proof nets are a graphical representation of proofs

described as "classical natural deduction" which equate proofs up to various commutations. Connection graphs are *exactly* shallow proof nets that equate proofs up to the commutations SC-CutComm and SC-CutAssoc!

The translation from connection graphs to shallow proof nets is simple. Consider the connection graph for example (2) as presented above:



To create the corresponding shallow proof net, convert each process to its corresponding sequent calculus proof, convert each *edge* of the connection graph to a cut node, and connect each port of the cut node to the corresponding proposition in the sequent calculus proofs. (The vertical dots are the sequent calculus proof corresponding to the process P.)



The correspondence between connection graphs is well-known, e.g. Honda and Laurent [2010] discuss this relation between structural congruence and proof nets in the context of polarised linear logic. For CP, this correspondence opens up several interesting avenues for research. For instance, we could leverage the correspondence to construct a type-checking algorithm for CP's connection graphs, rather than its process terms, based on the various correctness criteria for proof nets, such as Girard's long trip criterion [Girard, 1987, p. 30], the Danos-Regnier criterion [Danos and Regnier, 1989], or Melliès' ribbon criterion [Melliès, 2004]. Moreover, it would be interesting to examine the proof structures that correspond to stronger versions of structural congruence, such as the shallow or deep congruences defined in this chapter, or strong bisimulation with delayed actions, as defined, e.g. by Kokke et al. [2019a, DHCP], which I suspect has a much tighter correspondence to proof nets.

2.2.6 Observational Equivalence

In this section, I provide the basis for a behavioural theory for CP, namely, definitions for an observability predicate and barbed congruence. The purpose of this section is merely to show that these definitions can be given without the additional machinery introduced by Atkey [2017]. Therefore, I will not delve too deeply into the details. Access to an observability predicate also somewhat eases the discussion of commuting conversions in § 2.3.

The definition of observability predicates is non-standard, as it is not defined in terms of a label-transition system, but it is well-known, and follows Sangiorgi and Walker [2003, p. 56, Exercise 2.1.3]. The definition of the barbed congruence is standard, and follows Kokke et al. [2019a].

Definition 2.54 (Observability predicates)**.** *The* observability predicate $\downarrow_x P$ *holds if and only if* P *is of the form* $\mathscr{E}[Q]$ *such that* $\mathrm{ready}(Q, x)$ *and* $\{y \in \mathrm{fn}(Q) \mid \mathrm{ready}(Q, y)\} \cap \mathrm{bn}(\mathscr{E}) = \varnothing$*.*

Definition 2.55 (Barbed congruence)**.** Barbed congruence*, written* $\cong$*, is the largest symmetric relation on well-typed processes that is:*

1. *type preserving (i.e. if* $P \cong Q$ *and* $P \vdash \Gamma$ *then* $Q \vdash \Gamma$*)*
2. *barb preserving (i.e. if* $P \cong Q$ *and* $\downarrow_x P$ *then* $\downarrow_x Q$*)*
3. *context-closed (i.e. if* $P \cong Q$ *then* $R[P] \cong R[Q]$ *for any well-typed* $R[\cdot]$*)*

For a coarser barbed congruence, which distinguishes, e.g. $x \triangleleft \mathrm{inl}.\, P$ and $x \triangleleft \mathrm{inr}.\, P$, index the observability predicates with coarser labels [Yoshida et al., 2002, Atkey, 2017, Kokke et al., 2019a]. However, such considerations are outside of the scope of this section.

2.3 Commuting Conversions Considered Bad

Wadler's CP has additional reduction rules, known as *commuting conversions* [Wadler, 2014, § 3.6]. The reason for adding these rules is that they allow the reduction strategy for Wadler's CP to correspond step-by-step to a standard proof of cut elimination in Gentzen's style [see Girard et al., 1989, § 13.2].

The commuting conversions are summarised in Figure 2.4. For clarity, I separate the rules into two different reduction relations [following Lindley and Morris, 2015]. I write $\longrightarrow$ for cut reductions and $\longrightarrow_{cc}$ for commuting conversions.[5] Wadler's reduction relation is defined as as

[5] My names for the reduction relations differ from those of Lindley and Morris [2015]. I use the right arrow ($\longrightarrow$) for CP's the reduction relation without commuting conversions, which I consider the canonical reduction relation and is consistent with later chapters, and the right arrow labelled with a 'W' for *Wadler* ($\longrightarrow_W$) for Wadler's reduction. Lindley and Morris [2015] use the right arrow ($\longrightarrow$) for Wadler's reduction, which they consider the canonical reduction relation, and use the right arrow

$$
\begin{array}{lll}
(\nu z\bar{z})(x[y].\,(P \parallel Q) \parallel R) \longrightarrow_{cc} x[y].\,(P \parallel (\nu z\bar{z})(Q \parallel R)) & \text{if } z \notin P & \text{CC-Send}_1 \\
(\nu z\bar{z})(x[y].\,(P \parallel Q) \parallel R) \longrightarrow_{cc} x[y].\,((\nu z\bar{z})(P \parallel R) \parallel Q) & \text{if } z \notin Q & \text{CC-Send}_2 \\
(\nu z\bar{z})(x(y).\,P \parallel Q) \longrightarrow_{cc} x(y).\,(\nu z\bar{z})(P \parallel Q) & & \text{CC-Recv} \\
(\nu z\bar{z})(x().\,P \parallel Q) \longrightarrow_{cc} x().\,(\nu z\bar{z})(P \parallel Q) & & \text{CC-Wait} \\
(\nu z\bar{z})(x \lhd \mathrm{inl}.\,P \parallel Q) \longrightarrow_{cc} x \lhd \mathrm{inl}.\,(\nu z\bar{z})(P \parallel Q) & & \text{CC-Select}_1 \\
(\nu z\bar{z})(x \lhd \mathrm{inr}.\,P \parallel Q) \longrightarrow_{cc} x \lhd \mathrm{inr}.\,(\nu z\bar{z})(P \parallel Q) & & \text{CC-Select}_2 \\
(\nu z\bar{z})(x \rhd \{\mathrm{inl}\colon P; \mathrm{inr}\colon Q\} \parallel R) & & \\
\qquad \longrightarrow_{cc} x \rhd \{\mathrm{inl}\colon (\nu z\bar{z})(P \parallel R); \mathrm{inr}\colon (\nu z\bar{z})(Q \parallel R)\} & & \text{CC-Offer}
\end{array}
$$

$$
\text{CC-Absurd}\quad \frac{z \in N,\ \bar{z} \in \mathrm{fn}(P) \qquad N' = (N \setminus \{z\}) \cup (\mathrm{fn}(P) \setminus \{\bar{z}\})}{(\nu z\bar{z})(x \not\downarrow N \parallel P) \longrightarrow_{cc} x \not\downarrow N'}
$$

$$
\text{CC-SC}\quad \frac{P \equiv P' \qquad P' \longrightarrow_{cc} Q' \qquad Q' \equiv Q}{P \longrightarrow_{cc} Q}
\qquad\qquad
\text{CC-Cong}\quad \frac{P \longrightarrow_{cc} P'}{\mathscr{E}[P] \longrightarrow_{cc} \mathscr{E}[P']}
$$

Figure 2.4: Commuting Conversions for Classical Processes

their union, written $\longrightarrow_w$:

$$\longrightarrow_w \triangleq \longrightarrow \cup \longrightarrow_{cc}$$

A consequence of the commuting conversions is that Wadler's CP has an appealingly simple canonical form: 'canonical form' simply means 'not a cut'. I call this *weak cut-free form*, by analogy to weak-head normal form.

Definition 2.56 (Weak Cut-Free Form)**.** *A process is in* weak cut-free form *if is not a cut, i.e. it is not of the form* $(\nu x\bar{x})(P \parallel Q)$ *for any* x, $\bar{x}$, P, *and* Q.

Equivalently, a process is in weak cut-free form *if it is ready.*

The commuting conversions are required to reduce processes to weak cut-free form. For instance, the process

$$(\nu x\bar{x})(a().\,\bar{x}[].\,0 \parallel x().\,P)$$

is not in weak cut-free form, but cannot reduce with any cut reduction.

Lindley and Morris [2015, Page 574] claim that it suffices to reduce using only cut reductions, and use commuting conversions only to reduce from canonical form to weak cut-free form. I formalize and prove their claim as a pair of theorems.

---

labelled with a 'C' for *principal cut* ($\longrightarrow_c$) for the reduction relation without commuting conversions.

First, I relate Lindley and Morris' claim to my definition of canonical form by showing that any process in canonical form reduces to a process in weak cut-free form using only commuting conversions.

Proposition 2.57. *For any process* P *in canonical form, there is a reduction using only commuting conversions* $P \longrightarrow^{\star}_{cc} P'$ *such that* P' *is in weak cut-free form.*

Proof. There are two cases:

1. If P is ready, it is in weak cut-free form.
2. Otherwise, we have $P \equiv (\nu x_1 \bar{x}_1)(P_1 \parallel \cdots (\nu x_n \bar{x}_n)(P_n \parallel P_{n+1}) \cdots)$ such that (1) each P_i is ready; (2) no P_i is a link; and (3) no P_i and P_j are ready to act on dual endpoints of the same channel.

   At least one process P_i is ready to act on a free endpoint: There are n pairs of dual endpoints (x_1, $\bar{x}_1$, ..., x_n, $\bar{x}_n$) and $n+1$ processes (P_1, ..., P_{n+1}). By (1), each process P_i is ready. If all $n+1$ processes P_1, ..., P_{n+1} are ready to act on endpoints from the bound channels x_1, $\bar{x}_1$, ..., x_n, $\bar{x}_n$, at least two processes must be ready to act on dual endpoints, which contradicts (3).

   By (2), the process ready to act on a free endpoint cannot be a link. Therefore, we can use the commuting conversion to propagate that action to the top, and get the reduction $P \longrightarrow^{\star}_{cc} P'$ such that P' is in weak cut-free form.

□

Secondly, I show that it is *sufficient* to reduce using only cut reductions followed by only commuting conversions, by showing that any reduction in Wadler's CP can be rewritten to this form.

Proposition 2.58. *For any reduction* $P \longrightarrow^{\star}_{w} R$*, there is a reduction* $P \longrightarrow^{\star} \longrightarrow^{\star}_{cc} R$.

Proof. By iteratively commuting all commuting conversion past each reduction using Lemma 2.59. (For the full proof, see § 2.5.)

□

Lemma 2.59. *If* $P \longrightarrow^{\star}_{cc} \longrightarrow \longrightarrow^{\star}_{cc} R$*, then* $P \longrightarrow \longrightarrow^{\star}_{cc} R$.

Proof. By iteratively permuting each commuting conversion past the reduction using Lemma 2.60. (For the full proof, see § 2.5.)

□

Lemma 2.60. *If* $P \longrightarrow_{cc} \longrightarrow R$*, then either* $P \longrightarrow \longrightarrow_{cc} R$ *or* $P \longrightarrow R$.

Proof. I will give the high-level intuition behind the proof, and give the formal proof for several interesting cases in § 2.5.

Recall that commuting conversions commute an action past a cut, e.g. to

$$(\nu z\bar{z})(\underline{x(y).}P \parallel Q) \longrightarrow_{cc} \underline{x(y).}(\nu z\bar{z})(P \parallel Q)$$

Recall that cut reductions eliminate actions, e.g.

$$(\nu x\bar{x})(\underline{x[y].}(P \parallel Q) \parallel \underline{\bar{x}(\bar{y}).}R) \longrightarrow (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R))$$

It follows that for any reduction $P \longrightarrow_{cc} \longrightarrow R$ there are two options. The commuting conversion and cut reduction either act on *the same action* or on *different actions*:

- If they act on *the same action*, the commuting conversion is being used to move the action into place for the cut reduction. We can do this using the structural congruence instead.
- Otherwise, the two reduction steps act on *different actions* in different subprocesses. We can postpone the commuting conversion.

Unfortunately, we cannot easily formalize this high-level intuition, as actions are not a standalone syntactic sort in CP. Therefore, the formal proof proceeds by induction on the cut reduction $\longrightarrow R$ followed by inversion on the commuting conversion $P \longrightarrow_{cc}$.

(For the full proof, see § 2.5.)

□

What is the relation between weak cut-free form and canonical form? Can we split some reduction $P \longrightarrow_{w}^{\star} R$ into $P \longrightarrow^{\star} Q$ and $Q \longrightarrow_{cc}^{\star} R$ using Proposition 2.58, then forget about the commuting conversions, and expect the process Q to be in canonical form? The short answer is "no".

Any process in weak cut-free form must be an action on a free endpoint, since there can be no bound endpoints at the top-level of a process. The commuting conversions can move an action on a free endpoint past any number of cuts. Therefore, a process can reduce to weak cut-free form using only the commuting conversions if it has any action on a free name under only cuts. As I discussed in § 2.2.4, that is inadequate as a definition for canonical form, e.g. the process

$$(\nu x\bar{x})(a().\, P \parallel (\nu y\bar{y})(y[].\, 0 \parallel \bar{y}().\, Q))$$

has an action on a free endpoint under a single cut but can still reduce.

Lemma 2.60 reveals an issue with commuting conversions: *redundancy*. Pairs of dual actions must be adjacent in the process configuration

before any reduction rule applies. This is a problem of syntax. The structural congruence dictates that process configurations can be permuted. However, in Wadler's CP, there are at least two ways to move dual actions into adjacent positions. We can either use the structural congruence or the commuting conversions. Worse, by combining both, we can pick any number of unrelated processes for the dual actions to commute over.

Example 2.61 (Redundancy)**.** *The following process has two distinct reductions that eliminate the dual actions* $x[]$ *and* $\bar{x}()$*. One uses structural congruence, and the other uses commuting conversions.*

$$
\begin{array}{ccc}
 & (\nu x\bar{x})(x[].\,0 \parallel (\nu y\bar{y})(\bar{x}().\,Q \parallel R)) & \\
 & \equiv \quad \searrow_{cc} & \\
(\nu x\bar{x})(x[].\,0 \parallel \bar{x}().\,(\nu y\bar{y})(Q \parallel R)) & & (\nu y\bar{y})((\nu x\bar{x})(x[].\,0 \parallel \bar{x}().\,Q) \parallel R) \\
 & \searrow \quad \swarrow & \\
 & (\nu y\bar{y})(Q \parallel R) &
\end{array}
$$

The reduction strategy described by Wadler [2014, Proposition 2] does not meaningfully use the structural congruence; its only use is to derive the symmetric versions of the reduction rules, such as, e.g. the symmetric version of E-Link:

$$
\frac{(\nu \bar{x}x)(P \parallel x{\leftrightarrow}w) \equiv (\nu x\bar{x})(x{\leftrightarrow}w \parallel P) \qquad (\nu x\bar{x})(x{\leftrightarrow}w \parallel P) \longrightarrow P\{w/\bar{x}\}}{(\nu \bar{x}x)(P \parallel x{\leftrightarrow}w) \longrightarrow P\{w/\bar{x}\}}
$$

There is another issue with commuting conversions. To illustrate the issue, let us have a look at CC-Wait:

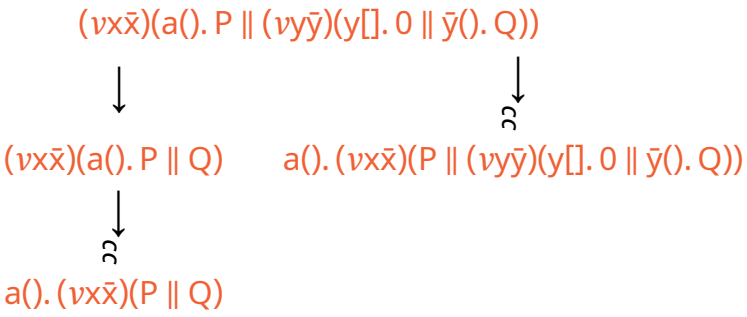
$$
(\nu z\bar{z})(x().\,P \parallel Q) \longrightarrow_{cc} x().\,(\nu z\bar{z})(P \parallel Q)
$$

On the left-hand side of the reduction rule, Q is a top-level process. Therefore, any action on a free endpoint in Q can be observed, and any reduction $Q \longrightarrow_w Q'$ can happen. On the right-hand side of the reduction, both of these things are blocked by the action $x()$.

Example 2.62 (Blocked Observable)**.** *The reduction with CC-Wait on the action* $a()$ *blocks the observation* $\downarrow_b$

$$
\begin{array}{ll}
(\nu x\bar{x})(a().\,P \parallel b().\,Q) & \textit{has observables } \downarrow_a, \downarrow_b \\
\quad \downarrow_{cc} & \\
a().\,(\nu x\bar{x})(P \parallel b().\,Q) & \textit{has observables } \downarrow_a
\end{array}
$$

Example 2.63 (Blocked Reduction)**.** *The reduction with CC-Wait on the action* a() *blocks the reduction with E-Close on the actions* y[] *and* y()



Example 2.63 reveals that Wadler's CP is non-confluent! The processes at the bottom of the tree, a(). (νxx̄)(P ‖ Q) and a(). (νxx̄)(P ‖ (νyȳ)(y[]. 0 ‖ ȳ(). Q)), are distinct weak cut-free forms of (νxx̄)(a(). P ‖ (νyȳ)(y[]. 0 ‖ ȳ(). Q)).

Canonical form is a stronger property than weak cut-free form. For some reductions $P \longrightarrow^{\star}_{w} R$ where R is weak cut-free, when we apply Proposition 2.58 to decompose the reduction into $P \longrightarrow^{\star} Q$ and $Q \longrightarrow^{\star}_{cc} R$, we find the process Q is not in canonical form: further cut reductions were initially possible, but became blocked by the commuting conversions. Formally, weak cut-free form corresponds to a variant of canonical form, called *weak canonical form*, which swaps the universal "all processes are ready" for an existential "some process is ready".

Definition 2.64 (Weak Canonical Form)**.** *A process* P *is in* weak canonical form *if* P *is of the form* $\mathscr{E}[Q]$ *such that* Q *is ready to act on a free endpoint that is not bound by* $\mathscr{E}$*.*

Equivalently, a process P *is in* weak canonical form *if there is a reduction* $P \longrightarrow^{\star}_{cc} P'$ *such that* P′ *is in weak cut-free form.*

As a consequence of the blocking behaviour of commuting conversions, the parallel composition of Wadler's CP is not truly parallel composition. To illustrate this, let us each imagine an implementation of Wadler's CP in our favourite programming language, and let us recall CC-Wait:

$$(\nu z\bar{z})(x().\, P \parallel Q) \longrightarrow_{cc} x().\, (\nu z\bar{z})(P \parallel Q)$$

Imagine that we implement each ready action P_i in a process as some program `ti` running in its own thread. The program `t1` implements x(). P: it is blocked, waiting to receive a ping on the channel `x`. The programs `t2`,...,`tN` implement Q: they are a collection of programs connected by channels, each program running in its own thread. There may be some communication between the programs `t2`,...,`tN`, but due to the typing rule for cut, only one of them is connected to `t1`.

The commuting conversion CC-Wait requires that, because `t1` is blocked on `x`, *all* of the programs `t2`,...,`tN` can become blocked on `x`. An implementation would require *some* communication between `t1` and

`t2`,…,`tN`, but this *cannot be* communication on shared channels, as `t1` only shares a channel with one of `t2`,…,`tN`. Therefore, when we implement the cut from Wadler's CP, we cannot simply use two parallel threads with a shared channel. We *must* add some kind of global scheduler. Consequently, implementations of Wadler's CP *must be* less parallel than we would expect from the syntax.

Formally, simulations of Wadler's CP in the π-calculus *must* decrease the degree of distribution, i.e. it is not possible to translate parallel composition to parallel composition.

Claim 2.65. *There exists no translation from the processes of Wadler's CP to processes of the π-calculus, written* $[\![\cdot]\!]_\pi$*, such that:*

- *The translation preserves the degree of distribution, i.e. is a homomorphism on parallel composition, which, due to CP's glued syntax, requires that the following equalities hold*

$$\begin{aligned} [\![(\nu x\bar{x})(P \parallel Q)]\!]_\pi &= (\nu x)([\![P]\!]_\pi \parallel [\![Q\{x/\bar{x}\}]\!]_\pi) \\ [\![x[y].(P \parallel Q)]\!]_\pi &= x[y].([\![P]\!]_\pi \parallel [\![Q]\!]_\pi) \end{aligned}$$

  (Where the terms on the right-hand side denote π-calculus processes, i.e. (νx) *is channel name restriction and* $x[y]$ *is bound output.)*
- *The translation is a simulation, i.e.* $P \longrightarrow_W Q$ *implies* $[\![P]\!]_\pi \longrightarrow_\pi^\star [\![Q]\!]_\pi$*. (Where* $\longrightarrow_\pi^\star$ *denotes the reflexive-transitive closure of the reduction relation for the π-calculus.)*

In summary, the commuting conversions in Wadler's CP are redundant, break confluence, and break the interpretation of parallel composition as parallel composition.

The version of CP presented in this chapter *is* confluent, but this is difficult to prove for the reduction semantics, as it requires showing that the structural congruence preserves the observable actions. Ultimately, a proof of congruence can be constructed by its operational correspondence with HCP (Chapter 3), which is the subject of the next chapter, and the proof of congruence for HCP by Montesi and Peressotti [2021].

2.4 Conclusion

In this chapter, I revisited Classical Processes and its metatheory. I dropped the commuting conversions, which cause Wadler's CP to be non-confluent, from the reduction semantics. I proved *preservation* (Proposition 2.27), and proved that *progress* (Proposition 2.32) continues to hold, albeit with a different canonical form (Definition 2.30). I proved that CP's processes are deadlock-free (Proposition 2.39), and that the new canonical forms are adequate (Corollary 2.47). I proved

that CP's connection graphs are trees (Proposition 2.50), and that any process can be rewritten to right-branching form (Proposition 2.51). Finally, I discussed the relation between CP with and CP without commuting conversions. I proved that any process in canonical form can reduce to a process in Wadler's canonical form using only commuting conversions (Proposition 2.57); and that any reduction that uses commuting conversions is equivalent to a reduction that does not use commuting conversions followed by a reduction using only commuting conversions (Proposition 2.58).

2.5 Omitted Proofs

Proposition 2.58. *For any reduction* $P \longrightarrow_w^\star R$, *there is a reduction* $P \longrightarrow^\star \longrightarrow_{cc}^\star R$.

Proof. By induction on the length of the reduction $P \longrightarrow_w^\star R$.

There are three cases:

1. The length is 0.

   The result follows by reflexivity.

2. The length is $N+1$ and the reduction is of the form $P \longrightarrow_w^\star R' \longrightarrow R$.

   By induction, we have $P \longrightarrow^\star \longrightarrow_{cc}^\star R'$. Hence, $P \longrightarrow^\star \longrightarrow_{cc}^\star R' \longrightarrow R$.

   The result follows by Lemma 2.59.

3. The length is $N+1$ and the reduction is of the form $P \longrightarrow_w^\star R' \longrightarrow_{cc} R$.

   By induction, we have $P \longrightarrow^\star \longrightarrow_{cc}^\star R'$. Hence, $P \longrightarrow^\star \longrightarrow_{cc}^\star R' \longrightarrow_{cc} R$.

   The result follows immediately.

□

Lemma 2.59. *If* $P \longrightarrow_{cc}^\star \longrightarrow \longrightarrow_{cc}^\star R$, *then* $P \longrightarrow \longrightarrow_{cc}^\star R$.

Proof. By induction on the length of the reduction $P \longrightarrow_{cc}^\star$.

There are two cases:

1. The length is 0.

   The reduction is of the form $P \longrightarrow \longrightarrow_{cc}^\star R$.

   The result follows immediately.

2. The length is $N+1$.

   The reduction is of the form $P \longrightarrow_{cc}^\star P' \longrightarrow_{cc} \longrightarrow R' \longrightarrow_{cc}^\star R$.

   By Lemma 2.60, one of two cases must hold:

a. There is a reduction $P' \longrightarrow\!\!\!\!\longrightarrow_{cc} R'$.

   Hence, $P \longrightarrow^{\star}_{cc} P' \longrightarrow\!\!\!\!\longrightarrow_{cc} R' \longrightarrow^{\star}_{cc} R$.

   The result follows from the induction hypothesis.

b. There is a reduction $P' \longrightarrow R'$.

   Hence, $P \longrightarrow^{\star}_{cc} P' \longrightarrow R' \longrightarrow^{\star}_{cc} R$.

   The result follows from the induction hypothesis.

□

Lemma 2.60. *If* $P \longrightarrow_{cc}\!\!\!\!\longrightarrow R$, *then either* $P \longrightarrow\!\!\!\!\longrightarrow_{cc} R$ *or* $P \longrightarrow R$.

Proof. By induction on the cut reduction $\longrightarrow R$ followed by inversion on the commuting conversion $P \longrightarrow_{cc}$. The induction is guarded by the decreasing length of the derivation of the cut reduction. We examine three cases:

1. The cut reduction uses E-Send and is of the form (reusing P and R)

$$(\nu x\bar{x})(x[y].(P \parallel Q) \parallel \bar{x}(\bar{y}).R) \longrightarrow (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R))$$

   By inversion, there are six cases for the commuting conversion. It uses either CC-Send$_1$, CC-Send$_2$, or CC-Recv under CC-Cong, or the symmetric version of any of these. We examine only the case where the reduction uses CC-Recv under CC-Cong and is of the form (reusing P, Q, and R)

$$\begin{array}{ll} & (\nu x\bar{x})(x[y].(P \parallel Q) \parallel (\nu z\bar{z})(\bar{x}(\bar{y}).R \parallel R')) \\ \longrightarrow_{cc} & (\nu x\bar{x})(x[y].(P \parallel Q) \parallel \bar{x}(\bar{y}).(\nu z\bar{z})(R \parallel R')) \\ \longrightarrow & (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel (\nu z\bar{z})(R \parallel R'))) \end{array}$$

   We replace the use of the commuting conversion with two uses of the structural congruence:

$$\begin{array}{ll} & (\nu x\bar{x})(x[y].(P \parallel Q) \parallel (\nu z\bar{z})(\bar{x}(\bar{y}).R \parallel R')) \\ \equiv & (\nu z\bar{z})((\nu x\bar{x})(x[y].(P \parallel Q) \parallel \bar{x}(\bar{y}).R) \parallel R') \\ \longrightarrow & (\nu z\bar{z})((\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R)) \parallel R') \\ \equiv & (\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel (\nu z\bar{z})(R \parallel R'))) \end{array}$$

2. The cut reduction uses E-Cong and is of the form (reusing P)

$$(\nu x\bar{x})(P \parallel Q) \longrightarrow (\nu x\bar{x})(P' \parallel Q)$$

   The premise to E-Cong is of the form $P \longrightarrow P'$. By inversion, there are two cases for the commuting conversion. It uses CC-Cong.

a. The reduction uses CC-Cong and is of the form

$$(\nu x\bar{x})(P'' \parallel Q) \longrightarrow_{cc} (\nu x\bar{x})(P \parallel Q) \longrightarrow (\nu x\bar{x})(P' \parallel Q)$$

The premise to CC-Cong is of the form $P'' \longrightarrow_{cc} P$. The induction hypothesis gives us either $P'' \longrightarrow\longrightarrow_{cc} P'$ or $P'' \longrightarrow P'$. In either case, the result follows immediately.

b. The reduction uses CC-Cong and is of the form

$$(\nu x\bar{x})(P \parallel Q') \longrightarrow_{cc} (\nu x\bar{x})(P \parallel Q) \longrightarrow (\nu x\bar{x})(P' \parallel Q)$$

The premise to CC-Cong is of the form $Q' \longrightarrow_{cc} Q$. The two reduction steps act on parallel processes, and can be reordered:

$$(\nu x\bar{x})(P \parallel Q') \longrightarrow (\nu x\bar{x})(P' \parallel Q') \longrightarrow_{cc} (\nu x\bar{x})(P' \parallel Q)$$

3. The cut reduction uses E-Equiv and is of the form $Q \equiv Q' \longrightarrow R' \equiv R$. The premise to E-Equiv is of the form $Q' \longrightarrow R'$. The induction hypothesis, applied to $P \longrightarrow_{cc} \equiv Q' \longrightarrow R'$, gives us either $P \longrightarrow\longrightarrow_{cc} R'$ or $P \longrightarrow R'$. The result follows as $P \longrightarrow\longrightarrow_{cc} \equiv R$ or $P \longrightarrow \equiv R$, respectively.

□

Part II

Taking Linear Logic Apart

Chapter 3

Hypersequent Classical Processes

This chapter presents Hypersequent Classical Processes (HCP), a session-typed process calculus based on CP as presented in Chapter 2. Nonetheless, the definitions, statements, and proofs in this chapter are self-contained, and do not reference those in Chapter 2, except where the goal is to relate CP and HCP.

HCP was independently developed by myself and by Fabrizio Montesi & Marco Peressotti [see Montesi and Peressotti, 2018]. Upon finding out about this parallel development, we collaborated on HCP, and published two versions:

- *Taking Linear Logic Apart* at Linearity-TLLA'18 [Kokke et al., 2019b]

  The paper presents HCP as presented in this chapter. Unfortunately, the published version contains errors and uses notation and naming conventions that do not match later publications, so I have chosen not to include it in this thesis.[1]

  The paper refers to the calculus as HCP$^-$, rather than HCP, to distinguish it from the version below.

- *Better Late Than Never* at POPL'19 [Kokke et al., 2019a]

  The paper presents Delayed HCP, which extends HCP with non-blocking actions. DHCP is not presented in this thesis.

  The paper refers to the calculus as HCP, rather than DHCP, as we believed *at the time* that DHCP would be the preferred version. The authors have since changed their minds. I use HCP to refer to the

[1] In May of 2019, Marco Peressotti discovered an error in the published version of Kokke et al. [2019b]. We submitted an erratum to EPTCS in July of 2019 which was approved for publication in October of 2021. Unfortunately, at the time of writing, the erratum has not yet been published. The version of the paper on Marco Peressotti's personal website, linked from the bibliography, contains an erratum.

version *without* delayed actions. Montesi and Peressotti [2021] use the name πLL instead of HCP[2].

HCP's main innovation is its type system, which lets us safely separate name restriction and parallel composition. Consequently, HCP more closely resembles the π-calculus and is more amenable to standard behavioural theory. Before we delve into the details, let us examine HCP at a glance, and investigate how it relates to Classical Processes, Classical Linear Logic [CLL, Girard, 1987], and the π-calculus [Milner et al., 1992b].

In this chapter, HCP's processes are printed in red, and its types are printed in blue, and both are rendered in a sans-serif font. To save on accessible colour combinations, the terms and types of *any other system* are printed in *pink* and *green*, respectively, both are rendered in an italicised font with serif, and any relations, such as typing and reduction, are marked by a subscript.

How does HCP relate to CP? The primary difference between CP and HCP is that HCP has a standalone process construct for parallel composition, whereas CP bakes parallel composition into the constructs for name restriction and sending—i.e. $(\nu \mathsf{x}\bar{\mathsf{x}})\mathsf{P}$, $\mathsf{x}[\mathsf{y}].\,\mathsf{P}$, and $\mathsf{P} \parallel \mathsf{Q}$ versus $(\nu x\bar{x})(P \parallel Q)$ and $x[y].\,(P \parallel Q)$. Likewise, HCP has a standalone process construct for the terminated process, which CP bakes into the construct for channel closing—i.e. $\mathsf{x}[].\,\mathsf{P}$ and $\mathsf{0}$ versus $x[].\,0$.

To match the decomposition of processes, HCP decomposes its typing rules, and adds new typing rules for parallel composition and the terminated process:

$$\frac{\mathsf{P} \vdash \mathscr{G} \qquad \mathsf{Q} \vdash \mathscr{H}}{\mathsf{P} \parallel \mathsf{Q} \vdash \mathscr{G} \parallel \mathscr{H}}\ \text{T-Par} \qquad \frac{}{\mathsf{0} \vdash \circ}\ \text{T-Halt}$$

To ensure that the properties of CP are preserved, only channel endpoints held by different processes are safe to connect. To this end, HCP adds the structural connectives "‖" and "∘" which separate channel endpoints into groups corresponding to the processes that hold them. (Note that "∘" is distinct from the empty typing environment "∅".)

Why are parallel composition and the terminated process typed by structural connectives rather than logical connectives? A channel endpoint is typed by a session type, and a process is typed by a typing environment that maps each free endpoint to its session type. An action acts on an endpoint, so the typing rules for actions are naturally the logical rules for some logical connective. However, parallel composition acts on processes and the terminated process simply is a process, so their typing rules must be structural rules for some structural connectives.

[2] Montesi and Peressotti's πLL continues the work on HCP. Of course, πLL is not a mere renaming. For instance, it includes the exponentials and second-order quantifiers.

To match the decomposition of processes, HCP also decompose the rules of its structural congruence, which become the standard rules for the π-calculus—such as associativity and commutativity for parallel composition, the unit laws for the terminated process, and scope extrusion. The reduction rules of HCP are exactly the same as those of CP.

I defer detailed discussion of the relation between HCP and CP to § 3.2. There we will see that HCP is a calculus for analysing collections of CP-like processes, and that HCP connection graphs are forests, rather than trees.

How does HCP relate to Classical Linear Logic? The crucial property of CP is its *exact* correspondence to linear logic. If you erase all that's written in red from the typing rules of CP, you get the inference rules of CLL, as given by Girard [1987]. Certainly, as HCP changes those typing rules, it must weaken that correspondence. I argue that it does the best possible job of preserving the property.

HCP is a conservative extension of CP. The corresponding logic, HCLL, is a conservative extension of CLL. Any CLL proof is an HCLL proof, and HCLL proves no new theorems about existing connectives. Formally,

$$\vdash_c \Gamma \iff \vdash \Gamma$$

where $\vdash_c \Gamma$ and $\vdash \Gamma$ denote sequents in CLL and HCLL, respectively.

HCLL extends CLL with the ability to reason about multiple unrelated proofs using *hypersequents* [Pottinger, 1983, Avron, 1991]. (Hence, Hypersequent CP.) A hypersequent logic has judgements over finite multisets of sequents, rather than single sequents. Avron writes hypersequents as:

$$\Gamma_1 \vdash \Delta_1 \mid \ldots \mid \Gamma_n \vdash \Delta_n$$

As CLL and HCLL use one-sided sequents, the turnstile is somewhat superfluous, merely serving to remind us of the polarity of the typing environment, i.e. that commas are structurally pars rather than tensors. Frankly, the turnstile remains in CP only because *something* must separate the process and its typing environment and the colon is already taken. Hence, we might as well use the familiar turnstile. It looks pleasing and it reminds us of the polarity to boot. If we were to use Avron's notation for hypersequents, we would find ourselves writing the turnstile over and over. Worse, the turnstile would lose its primary function—to stand between a process and its typing environment. Instead, I present HCLL as a logic over finite multisets of *environments*, where environments are multisets of formulas. I refer to multisets of environments as *hyper-environments*. The turnstile keeps its position to the left of the hyper-environment, and continues to serve its primary function of separating term from type, red from blue. (Nonetheless, we continue to refer to HCLL as a hypersequent calculus.)

Hyper-environments, denoted by $\mathscr{G}$ and $\mathscr{H}$, are multisets of typing environments. I write $\circ$ for the empty hyper-environment, and $\|$ for hyper-environment concatenation. The rules for ($\|$) and ($\circ$) are as follows:

$$\frac{\vdash \mathscr{G} \qquad \vdash \mathscr{H}}{\vdash \mathscr{G} \,\|\, \mathscr{H}}\ (\|) \qquad \frac{}{\vdash \circ}\ (\circ)$$

What does a proof of a hyper-environment mean? A proof of a multiset of typing environments should imply a multiset of disjoint proofs, with one separate proof for each typing environment. In the larger proof, each of the separate proofs may be entangled—their inference rules mixed together—but each inference rule should belong to exactly one of the separate proofs, and we should be able to disentangle them into a sequence of CLL proofs:

$$\begin{array}{c} p \\ \vdots \\ \vdash \Gamma_1 \,\|\, \ldots \,\|\, \Gamma_n \end{array} \implies \begin{array}{ccc} p_1 & & p_n \\ \vdots & \ldots & \vdots \\ \vdash \Gamma_1 & & \vdash \Gamma_n \end{array}$$

There is a deep connection between the logical connective "$\otimes$", the structural connective "$\|$", and branching—having multiple premises—in multiplicative inference rules. All three capture some notion of *disjointness*:

- For branching, the disjointness is immediately apparent. Sequent calculus proofs are trees, so the premises are disjoint.
- For the structural connective "$\|$", the disjointness follows from the disentanglement property stated above.
- For the logical connective "$\otimes$", the disjointness follows from the splitting theorem [e.g. Girard, 1987, p. 39, Theorem 2.9.7, for CLL proof nets], which states that a proof of some judgement with subformula $A \otimes B$ can be decomposed into separate proofs of A and its environment, B and its environment, and a common prefix:

$$\begin{array}{c} p \\ \vdots \\ \vdash \Gamma, C[A \otimes B] \end{array} \implies \begin{array}{c} p_1 \qquad p_2 \\ \vdots \qquad \vdots \\ \dfrac{\vdash \Delta_1, A \qquad \vdash \Delta_2, B}{\vdash \Delta_1, \Delta_2, A \otimes B} \\ \vdots \\ f \\ \vdots \\ \vdash \Gamma, C[A \otimes B] \end{array}$$

In summary, the connective "$\|$" is a *structural* "$\otimes$", and the judgement $\vdash \Gamma \| \Delta$ means that the proofs of Γ and Δ are disjoint and can be disentangled into separate proofs. Likewise, the connective "$\circ$" is a *structural* "$\mathbf{1}$", and the judgement "$\vdash \circ$" means that there is nothing to prove.

Since hyper-environments internalise branching for multiplicative rules into the structure of the logic, the multiplicative rules may become unary.

For instance, HCLL's Cut rule has only one premise:

$$\frac{\vdash \mathscr{G} \parallel \Gamma, A \parallel \Delta, \overline{A}}{\vdash \mathscr{G} \parallel \Gamma, \Delta}\ \textsc{Cut}$$

The premise requires that the positive and negative occurrences of the cut formula are separated by a "‖", which ensures that their respective proofs are disjoint. In the conclusion, the environments are merged, since the proofs have become connected. This bookkeeping maintains the invariant that cut is only allowed to connect disjoint proofs, and avoids admitting rules that fundamentally alter the logic, such as MultiCut [see, e.g. Atkey et al., 2016].

Likewise, HCLL's rule for tensor has only one premise, but there are two possible versions of the rule, which differ in whether or not the rule applies in the presence of unrelated proofs.

$$\frac{\vdash \Gamma, A \parallel \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\ (\otimes) \qquad \frac{\vdash \mathscr{G} \parallel \Gamma, A \parallel \Delta, B}{\vdash \mathscr{G} \parallel \Gamma, \Delta, A \otimes B}\ (\otimes_D)$$

The two are logically equivalent: $(\otimes)$ is derivable from $(\otimes_D)$, and $(\otimes_D)$ is admissible using $(\otimes)$ and disentanglement. However, the two give rise to different process semantics:

- HCP, as presented in this chapter and in Kokke et al. [2019b, with errata], uses $(\otimes)$, which gives rise to the standard semantics of the π-calculus and is compatible with CP.
- DHCP, as presented in Kokke et al. [2019a], uses $(\otimes_D)$, which gives rise to delayed action semantics, as disentanglement requires that the action associated with $(\otimes_D)$ commutes with parallel composition.

How does HCP relate to the π-calculus? There is a slight difference in the semantics of communication channels, name restriction, and parallel composition, between HCP and the π-calculus, as presented by, e.g. Milner et al. [1992b] or Sangiorgi and Walker [2003]. Let us investigate this difference by asking a question: What makes a communication channel? There are at least two answers:

1. A communication channel is a name. Two processes can communicate simply by having access to the same name. Name restriction *restricts* the scope of a name, but communication channels exist irrespective of name restriction.
2. A communication channel is explicitly created. Two processes can only communicate if they have access to the same name bound in the scope of the name restriction that creates it. Unbound names are disconnected endpoints, not attached to any channel.

A bit tongue-in-cheek, I will refer to (1) as the *restrictive* view, and to (2) as the *creative* view. The π-calculus [Milner et al., 1992b] takes the

restrictive[3] view, whereas Hypersequent CP takes the creative view. The two can be distinguished by their reduction rules—or, equivalently, their label-transition rules, which we will discuss later. Under the restrictive view, communication happens on free names, e.g. as in rule (a), and can happen on bound names simply by congruence. Under the creative view, communication can only happen on bound names, e.g. as in rule (b).

(a) $a\langle c\rangle.\, P \parallel a(y).\, Q \longrightarrow_\pi P \parallel Q\{c/y\}$

(b) $(\nu x)(x\langle c\rangle.\, P \parallel x(y).\, Q) \longrightarrow_\pi (\nu x)(P \parallel Q\{c/y\})$

The view we take has significant consequences for our type system:

1. Under the restrictive view, connection happens coincidentally. When two processes are composed in parallel, any number of connections can happen simultaneously, by the simple coincidence of names.
2. Under the creative view, connection happens intentionally. When two processes are composed in parallel, they are not connected, and they remain unconnected until they are intentionally connected by a ν-binder, one channel at a time.

The creative view, due to its simplicity, is significantly more amenable to a correspondence with logic. To illustrate this, let us compare the typing rules for parallel composition from linear π-calculus [Lπ, Kobayashi et al., 1996], which takes the restrictive view, to Hypersequent CP, which takes the creative view:

$$\frac{P \vdash_{L\pi} \Gamma \qquad Q \vdash_{L\pi} \Delta}{P \parallel Q \vdash_{L\pi} \Gamma + \Delta}\ \text{T-Par} \qquad \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \parallel Q \vdash \mathcal{G} \parallel \mathcal{H}}\ \text{T-Par}$$

Superficially, the rules are similar, but complexity hides in the details:

1. In Lπ, all session types are annotated with their capabilities—whether the corresponding channel is used to send, receive, neither, or both. The "+" is a partial function in the meta-language, which merges two typing environments by adding together the uses of a channel name in the two environments. In the well-typed cases, it computes the union of the usages, e.g. if x was used to send in Γ and used to receive in Δ, then it is fully used in $\Gamma + \Delta$. However, in the ill-typed cases, it is undefined, e.g. if x is used to send in both Γ and Δ, then $\Gamma + \Delta$ is undefined, and the typing rule for parallel composition does not apply.
2. In HCP, the "$\parallel$" is a structural connective, much like the comma, and simply means "these resources are used in different processes". The requirement for wellformedness is simpler: $\mathcal{G} \parallel \mathcal{H}$ is well-formed when the names in $\mathcal{G}$ and $\mathcal{H}$ are all distinct.

[3] Confusingly, the word "restriction" [as used by Milner et al., 1992a, submitted in 1989] connotes the restrictive view, but the later introduction of the letter "ν" [pronounced as "new", coined by Milner, 1991] connotes the creative view.

By taking the creative view, we get a typing rule for parallel composition which is much more logical, in the technical sense. What do we lose? We lose connection by coincidence, which I believe is not much of a loss, and perhaps even a win. So Hypersequent CP takes the creative view.

This chapter proceeds as follows:

- In § 3.1, I introduce Hypersequent CP.
- In § 3.2, I introduce the metatheory for Hypersequent CP.

  I prove *preservation* (Proposition 3.30) and *progress* (Proposition 3.35), that its processes are deadlock-free (Corollary 3.42), that its canonical forms are adequate (Corollary 3.50), and that its communication graphs are forests (Proposition 3.52).

  Notably, I also define disentanglement (Definition 3.83), which converts Hypersequent CP processes into multisets of CP processes, and prove it preserves typing (Proposition 3.84), structural congruence (Proposition 3.85), and reduction (Proposition 3.87), and define a label-transition system for HCP (Definition 3.88), and prove harmony (Proposition 3.93).
- In § 3.3, I discuss the relation between HCP and other developments in proof theory, and introduce several variants of HCP.

  Notably, I introduce variants that interpret the absurd offer as session cancellation (§ 3.3.1), assign synchronous semantics to link (§ 3.3.2), and separate actions into their own syntactic sort by deconstructing the offer into a guarded summation (§ 3.3.3).
- Finally, § 3.5 contains all omitted proofs.

This chapter shares its structure with Chapter 2, and since the two systems share a fair amount of their structure, an equally fair amount of exposition is repeated, often verbatim, in Sections 3.1.2 to 3.1.6 and Sections 3.2.1 to 3.2.3. If you have read Chapter 2, it is worth reading § 3.1 up to and including § 3.1.1, the definition of configuration contexts in § 3.2.1, and then reading from § 3.2.5 onwards.

3.1 Hypersequent Classical Processes

In this section, I introduce Hypersequent Classical Processes (HCP), a session-typed process calculus based on CP. HCP's process calculus resembles the π-calculus more closely than CP, and its type system corresponds to a logic that is a slight variation of Classical Linear Logic, which I call Hypersequent Classical Linear Logic (HCLL). To the best of my knowledge, HCLL does not exist in the literature, and is only implicitly defined in this thesis by means of squinting and ignoring the

red parts. HCLL is a well-behaved and interesting logic, and, moreover, a conservative extension of CLL.

The fundamental notions of programs and computation in HCP, as in CP, are processes and message-passing communication. Processes communicate by passing messages over channels. Communication channels are *binary*, which means that each channel has exactly two endpoints, and each endpoint is held by exactly one process. Names refer to channel endpoints, rather than channels.

Processes (ranged over by P, Q, R) are defined by the following grammar:

P, Q, R	$::=$	$x \leftrightarrow y$	link
	$\mid$	$(\nu x \bar{x}) P$	new
	$\mid$	$P \parallel Q$	parallel
	$\mid$	0	terminated process
	$\mid$	$x[y].\, P$	send
	$\mid$	$x(y).\, P$	receive
	$\mid$	$x[].\, P$	close
	$\mid$	$x().\, P$	wait
	$\mid$	$x \triangleleft \mathsf{inl}.\, P$	select left
	$\mid$	$x \triangleleft \mathsf{inr}.\, P$	select right
	$\mid$	$x \triangleright \{\mathsf{inl}: P; \mathsf{inr}: Q\}$	choice
	$\mid$	$x \lightning N$	absurd

The names x, y, z, and w range over the endpoints of communication channels—'channel endpoints' or 'endpoints' for short. The names $\bar{x}$, $\bar{y}$, $\bar{z}$, and $\bar{w}$ as well as a, b, and c also range over endpoints, and are used with the same conventions as in Chapter 2, which we revisit shortly. The names N and M range over sets of endpoints.

An endpoint is *bound* in the following cases:

- In $(\nu x \bar{x}) P$, x and $\bar{x}$ are bound in P.
- In $x[y].\, P$, y is bound in P.
- In $x(y).\, P$, y is bound in P.

An endpoint is *free* if it is not bound. Notably, for $x \lightning N$, x and all names in N are free. I write $\mathrm{fn}(P)$ to denote the set of free endpoints in P. By convention, the names a, b, and c are used as a shorthand to imply to the reader that the endpoint is free.

Two endpoints are *dual* if they are bound by the same name restriction, e.g. in $(\nu x \bar{x}) P$, x and $\bar{x}$ are bound in P, and are dual. By convention, the names $\bar{x}$, $\bar{y}$, $\bar{z}$, and $\bar{w}$ are used as a shorthand to imply duality to the reader, e.g. I use x and $\bar{x}$ when they are dual endpoints of the same channel.

In HCP, as in CP, actions are not a well-defined syntactic sort. Nonetheless, I will informally write "action" in reference to the bit before the dot, e.g. the action for $x[y].\, (P \parallel Q)$ is $x[y]$. For the troublemakers without a dot, $x \triangleright \{\mathsf{inl}: P; \mathsf{inr}: Q\}$ and $x \lightning N$, I write $x \triangleright \mathsf{inl}$, $x \triangleright \mathsf{inr}$, and $x \lightning$, respectively.

Types (ranged over by A, B) are the formulas of CLL, as defined by the following grammar:

$$A, B ::= A \otimes B \mid \mathbf{1} \mid A \oplus B \mid \mathbf{0} \mid A \parr B \mid \bot \mid A \mathbin{\&} B \mid \top$$

Duality plays an important role in HCP, as it does in CP and CLL. Viewed from the perspective of a logic, it corresponds to negation. Viewed from the perspective of a process calculus, it guarantees session fidelity, i.e. that processes act on dual endpoints of the same channel in dual ways, e.g. one process sends when the other receives. As in CP and CLL, duality is not defined as a type constructor, but as a function on types:

$$\begin{array}{llll} \overline{A \otimes B} \triangleq \overline{A} \parr \overline{B} & & \overline{\mathbf{1}} \triangleq \bot \\ \overline{A \parr B} \triangleq \overline{A} \otimes \overline{B} & & \overline{\bot} \triangleq \mathbf{1} \\ \overline{A \oplus B} \triangleq \overline{A} \mathbin{\&} \overline{B} & & \overline{\mathbf{0}} \triangleq \top \\ \overline{A \mathbin{\&} B} \triangleq \overline{A} \oplus \overline{B} & & \overline{\top} \triangleq \mathbf{0} \end{array}$$

As we will see, dual endpoints have dual types. The notation for duality (A, $\overline{A}$) and the naming convention for dual endpoints (x, $\bar{x}$) were chosen to emphasize this. Duality is involutive.

Lemma 3.1. $\overline{\overline{A}} = A$

Typing environments (ranged over by Γ, Δ) are sets of type assignments, as defined by the following grammar:

$$\Gamma, \Delta ::= \varnothing \mid \Gamma, x : A$$

The set of free endpoint names in a typing environment, written $\mathrm{fn}(\Gamma)$, is defined by recursion on the environment, i.e. $\mathrm{fn}(\varnothing) \triangleq \varnothing$ and $\mathrm{fn}(\Gamma, x : A) \triangleq \mathrm{fn}(\Gamma) \cup \{x\}$. The extension $\Gamma, x : A$ is only defined when x is not free in Γ, i.e. $x \notin \mathrm{fn}(\Gamma)$.

We write Γ, Δ for the concatenation of typing environments Γ and Δ. The concatenation Γ, Δ is only defined when the names in Γ and Δ are unique, i.e. $\mathrm{fn}(\Gamma) \cap \mathrm{fn}(\Delta) = \varnothing$.

Hyper-environments (ranged over by $\mathscr{G}$, $\mathscr{H}$) are multisets of typing environments, as defined by the following grammar:

$$\mathscr{G}, \mathscr{H} ::= \circ \mid \mathscr{G} \parallel \Gamma$$

The set of free endpoint names in a hyper environment, written $\mathrm{fn}(\mathscr{G})$, is defined by recursion on the hyper environment, i.e. $\mathrm{fn}(\circ) \triangleq \varnothing$ and $\mathrm{fn}(\mathscr{G} \parallel \Gamma) \triangleq \mathrm{fn}(\mathscr{G}) \cup \mathrm{fn}(\Gamma)$. The extension $\mathscr{G} \parallel \Gamma$ is only defined when the names in all typing environments in $\mathscr{G}$ and Γ are unique, i.e. $\mathrm{fn}(\mathscr{G}) \cap \mathrm{fn}(\Gamma) = \varnothing$.

$$\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}}\ \text{T-Link} \qquad \frac{P \vdash \mathscr{G} \parallel \Gamma, x : A \parallel \Delta, \tilde{x} : \overline{A}}{(\nu x\tilde{x})P \vdash \mathscr{G} \parallel \Gamma, \Delta}\ \text{T-New}$$

$$\frac{P \vdash \mathscr{G} \qquad Q \vdash \mathscr{H}}{P \parallel Q \vdash \mathscr{G} \parallel \mathscr{H}}\ \text{T-Par} \qquad \frac{}{0 \vdash \circ}\ \text{T-Halt}$$

$$\frac{P \vdash \Gamma, x : A \parallel \Delta, y : B}{x[y].\, P \vdash \Gamma, \Delta, x : A \otimes B}\ \text{T-Send} \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).\, P \vdash \Gamma, x : A \parr B}\ \text{T-Recv}$$

$$\frac{P \vdash \circ}{x[].\, P \vdash x : \mathbf{1}}\ \text{T-Close} \qquad \frac{P \vdash \Gamma}{x().\, P \vdash \Gamma, x : \bot}\ \text{T-Wait}$$

$$\frac{P \vdash \Gamma, x : A}{x \triangleleft \mathrm{inl}.\, P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_1 \qquad \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathrm{inl}.\, P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_2$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathrm{inl}\colon P; \mathrm{inr}\colon Q\} \vdash \Gamma, x : A \,\&\, B}\ \text{T-Offer} \qquad \frac{N = \mathrm{fn}(\Gamma)}{x \lightning N \vdash \Gamma, x : \top}\ \text{T-Absurd}$$

Figure 3.1: Typing Rules for Hypersequent CP

We write $\mathscr{G} \parallel \mathscr{H}$ for the concatenation of hyper-environments $\mathscr{G}$ and $\mathscr{H}$. The concatenation $\mathscr{G} \parallel \mathscr{H}$ is only defined when the names in all typing environments in $\mathscr{G}$ and $\mathscr{H}$ are unique, i.e. $\mathrm{fn}(\mathscr{G}) \cap \mathrm{fn}(\mathscr{H}) = \varnothing$.

We write $\mathscr{G}^k$ to mean that the hyper-environment $\mathscr{G}$ consists of k typing environments, i.e. $\mathscr{G}^k$ is of the form $\Gamma_1 \parallel \ldots \parallel \Gamma_k$.

The typing judgment $P \vdash \mathscr{G}$ means that P is well-typed if, for each typing environment Γ in $\mathscr{G}$, and for each type assignment $x : A$ in Γ, exactly one process in P uses the endpoint x according to the session type A and that process only uses free endpoints that are in Γ.

Definition 3.2 (Typing)**.** *A process P is well-typed under typing environment $\mathscr{G}$ if there exists a derivation with conclusion $P \vdash \mathscr{G}$ that uses the typing rules in Figure 3.1.*

Processes are considered equivalent up to structural congruence, written $\equiv$, which ensures that, e.g. the direction of a link and the order of a parallel composition are irrelevant.

Definition 3.3 (Structural Congruence)**.** Structural congruence, *written $P \equiv Q$, is the congruence closure over processes which satisfies the rules in Figure 3.2.*

Definition 3.4 (Evaluation Context)**.** Evaluation contexts *are one-hole process contexts, as defined by the following grammar:*

$$\mathscr{E}, \mathscr{F} ::= \Box \mid (\nu x\tilde{x})\mathscr{E} \mid \mathscr{E} \parallel Q \mid Q \parallel \mathscr{E}$$

$$
\begin{array}{lll}
x{\leftrightarrow}y & \equiv y{\leftrightarrow}x & \text{SC-LinkComm} \\
P \parallel 0 & \equiv P & \text{SC-ParNil} \\
P \parallel Q & \equiv Q \parallel P & \text{SC-ParComm} \\
P \parallel (Q \parallel R) & \equiv (P \parallel Q) \parallel R & \text{SC-ParAssoc} \\
(\nu x\bar{x})P & \equiv (\nu \bar{x}x)P & \text{SC-NewComm} \\
(\nu x\bar{x})(\nu y\bar{y})P & \equiv (\nu y\bar{y})(\nu x\bar{x})P & \text{SC-NewAssoc} \\
(\nu x\bar{x})(P \parallel Q) & \equiv P \parallel (\nu x\bar{x})Q \quad \text{if } x, \bar{x} \notin \mathrm{fn}(P) & \text{SC-ScopeExt}
\end{array}
$$

Figure 3.2: Structural Congruence for Hypersequent CP

Plugging is defined by replacing the one hole with a process:

$$
\begin{array}{l}
\Box\ [P] \triangleq P \\
(\nu x\bar{x})\mathscr{E}\ [P] \triangleq (\nu x\bar{x})\mathscr{E}[P] \\
(\mathscr{E} \parallel Q)\ [P] \triangleq \mathscr{E}[P] \parallel Q \\
(Q \parallel \mathscr{E})\ [P] \triangleq Q \parallel \mathscr{E}[P]
\end{array}
$$

We write $\mathrm{fn}(\mathscr{E})$ *for the free endpoints in* $\mathscr{E}$.

We write $\mathrm{bn}(\mathscr{E})$ *for the endpoints bound by* $\mathscr{E}$.

We write $\mathscr{E} \vdash \mathscr{G} \to \mathscr{H}$ *to mean that the evaluation context* $\mathscr{E}$ *is well-typed under input typing context* $\mathscr{G}$ *and output typing context* $\mathscr{H}$.

$$
\frac{}{\Box \vdash \mathscr{G} \to \mathscr{G}} \qquad \frac{\mathscr{E} \vdash \mathscr{G} \to \mathscr{H} \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})\mathscr{E} \vdash \mathscr{G} \to \mathscr{H} \parallel \Gamma, \Delta}
$$

$$
\frac{\mathscr{E} \vdash \mathscr{G} \to \mathscr{H}_1 \qquad Q \vdash \mathscr{H}_2}{\mathscr{E} \parallel Q \vdash \mathscr{G} \to \mathscr{H}_1 \parallel \mathscr{H}_2} \qquad \frac{Q \vdash \mathscr{H}_1 \qquad \mathscr{E} \vdash \mathscr{G} \to \mathscr{H}_2}{Q \parallel \mathscr{E} \vdash \mathscr{G} \to \mathscr{H}_1 \parallel \mathscr{H}_2}
$$

The semantics of HCP processes is given by *reduction*, written $\longrightarrow$. Reduction is closed over evaluation contexts, and structural congruence is embedded in reduction by allowing pre- and post-composition using E-Cong, written as $\equiv\!\!\longrightarrow$, $\longrightarrow\!\!\equiv$, or $\equiv\!\!\longrightarrow\!\!\equiv$.

Definition 3.5 (Reduction)**.** *Reduction is the smallest relation on processes defined by the rules in Figure 3.3.*

In the remainder of the section, I discuss each process construct together with its typing rule and operational semantics, either by itself—e.g. *link*—or together with its dual—e.g. *send* and *receive*.

3.1.1 Process Structure

The process $(\nu x\bar{x})P$ denotes a *name restriction*, which creates a communication channel with two endpoints, x and $\bar{x}$, and the process $P \parallel Q$ denotes a *parallel composition*, which—well—composes two processes in parallel.

$$
\begin{array}{lll}
(\nu x\bar{x})(x{\leftrightarrow}w \parallel P) & \longrightarrow P\{w/\bar{x}\} & \text{E-Link} \\
(\nu x\bar{x})(x[y].\, P \parallel \bar{x}(\bar{y}).\, Q) & \longrightarrow (\nu x\bar{x})(\nu y\bar{y})(P \parallel Q) & \text{E-Send} \\
(\nu x\bar{x})(x[].\, 0 \parallel \bar{x}().\, Q) & \longrightarrow Q & \text{E-Close} \\
(\nu x\bar{x})(x \triangleleft \mathrm{inl}.\, P \parallel \bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\}) & \longrightarrow (\nu x\bar{x})(P \parallel Q) & \text{E-Select}_1 \\
(\nu x\bar{x})(x \triangleleft \mathrm{inr}.\, P \parallel \bar{x} \triangleright \{\mathrm{inl}\colon Q; \mathrm{inr}\colon R\}) & \longrightarrow (\nu x\bar{x})(P \parallel R) & \text{E-Select}_2
\end{array}
$$

$$
\text{E-Equiv}\ \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}
\qquad\qquad
\text{E-Cong}\ \frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']}
$$

Figure 3.3: Reduction for Hypersequent CP

The typing rules for name restriction and parallel composition are as follows:

$$
\frac{P \vdash \mathscr{G} \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})P \vdash \mathscr{G} \parallel \Gamma, \Delta}\ \text{T-New}
\qquad\qquad
\frac{P \vdash \mathscr{G} \qquad Q \vdash \mathscr{H}}{P \parallel Q \vdash \mathscr{G} \parallel \mathscr{H}}\ \text{T-Par}
$$

For communication safety, it is important that the two endpoints x and $\bar{x}$ have dual types, and that every endpoint is used exactly once.

- The T-New rule guarantees the former.
  It requires that $x : A$ and $\bar{x} : \overline{A}$.
- The T-Par rule guarantees the latter.
  It requires that the processes P and Q do not share any free endpoints, as the concatenation $\mathscr{G} \parallel \mathscr{H}$ is only defined when the hyper-environments $\mathscr{G}$ and $\mathscr{H}$ do not share any endpoint names.

For deadlock freedom—as well as a close correspondence to CLL—it is important that the two endpoints of a channel are used in separate processes, and that any two processes share *at most* one channel. Both of these requirements are guaranteed by the T-New rule, but the T-Par rule does the bookkeeping that enables this. Therefore, let us start our discussion with the T-Par rule:

- In a parallel composition $P \parallel Q$, the two processes P and Q are not connected by any channel, as there is no name restriction that takes scope over both. Hence, there cannot be any dependency between any of their endpoints. The two processes are disjoint. The T-Par rule registers this disjointness in the sequent by separating their respective hyper-environments with a "$\parallel$".
- In a name restriction $(\nu x\bar{x})P$, it is important that the two endpoints of a channel are used in separate processes, and that any two processes share *at most* one channel. The T-New rule guarantees both:
  - For the former, it requires that the two endpoints x and $\bar{x}$ are

separated by "‖" in the premise, which means that they will *eventually* be split between two separate processes.
- For the latter, it removes the "‖" in the conclusion, to register the fact that the two processes that use x and $\bar{x}$ are now connected by that channel.

The process 0 denotes the *terminated process*, which does nothing. The typing rule for the terminated process is as follows:

$$\frac{}{0 \vdash \circ}\ \textsc{T-Halt}$$

Hyper-environments are multisets. They are considered equal up to the unit rule for ‖ and ∘, and the commutativity and associativity rules for ‖.

$$\begin{aligned} \mathcal{G} \parallel \circ &= \mathcal{G} \\ \mathcal{G}_1 \parallel \mathcal{G}_2 &= \mathcal{G}_2 \parallel \mathcal{G}_1 \\ \mathcal{G}_1 \parallel (\mathcal{G}_2 \parallel \mathcal{G}_3) &= (\mathcal{G}_1 \parallel \mathcal{G}_2) \parallel \mathcal{G}_3 \end{aligned}$$

These structural rules for hypersequents are used implicitly, e.g. the following typing derivation implicitly uses the right unit rule.

$$\frac{\begin{array}{c}\vdots \\ P \vdash \mathcal{G}\end{array} \qquad \frac{}{0 \vdash \circ}}{P \parallel 0 \vdash \mathcal{G}}$$

Processes are equivalent up to structural congruence, which includes the unit rule for ‖ and 0 and the commutativity and associativity rules for ‖. These explicit structural rules for processes mirror the implicit structural rules for hyper-environments.

$$\begin{array}{lcll} P \parallel 0 & \equiv & P & \textsc{SC-ParNil} \\ P \parallel Q & \equiv & Q \parallel P & \textsc{SC-ParComm} \\ P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R & \textsc{SC-ParAssoc} \end{array}$$

The structural congruence also permits flipping the direction of a channel, permuting two name restrictions, and scope extrusion—which permits a process to move out of and into the scope of a name restriction, as long as it does not use the channel bound by that name restriction.

$$\begin{array}{lcll} (\nu x\bar{x})P & \equiv & (\nu \bar{x}x)P & & \textsc{SC-NewComm} \\ (\nu x\bar{x})(\nu y\bar{y})P & \equiv & (\nu y\bar{y})(\nu x\bar{x})P & & \textsc{SC-NewAssoc} \\ (\nu x\bar{x})(P \parallel Q) & \equiv & P \parallel (\nu x\bar{x})Q & \text{if } x, \bar{x} \notin \mathrm{fn}(P) & \textsc{SC-ScopeExt} \end{array}$$

There are no reduction rules associated with name restriction, parallel composition, or the terminated process. None of these constructs perform any action. Rather, they manage the structure of connections and parallel processes that facilitate message-passing communication.

Name restriction and parallel composition appear in every reduction rule, as communication can only happen over a channel, which requires

the presence of a name restriction, and between parallel processes, which requires the presence of a parallel composition. Furthermore, the E-Cong rule permits communication to occur under an arbitrary evaluation context of name restrictions, parallel compositions, and unrelated processes.

$$\text{E-Cong}\quad \frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']}$$

3.1.2 Link

The process $x{\leftrightarrow}y$ denotes a *link*. It forwards any messages received on x to y, and vice versa. For communication safety, the two endpoints must have dual types. Hence, $x : A$ and $y : \overline{A}$.

$$\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}}\ \text{T-Link}$$

HCP's semantics follows CP's semantics for link, and does not explicitly forward messages, but treats links as suspended α-renaming. When a link $x{\leftrightarrow}y$ reduces, it renames all occurrences of the dual of x to y, or all occurrences of the dual of y to x. In essence, this updates all the processes connected to the one side of the link to point directly at the other side, circumventing the link.

$$(\nu x\bar{x})(x{\leftrightarrow}w \parallel P) \longrightarrow P\{w/\bar{x}\}\quad \text{E-Link}$$

The renaming targets a bound name. Hence, there cannot be any other occurrences of that name, and the link can be removed.

Links are *commutative*. If two channels are connected by a link, the order in which they are connected is irrelevant. This property is captured by the following equivalence:

$$x{\leftrightarrow}y \equiv y{\leftrightarrow}x \qquad \text{SC-LinkComm}$$

3.1.3 Send and Receive

The send and receive actions are dual:

- The process $x[y].\,P$ denotes a *send* action.
  It creates a fresh channel, names one endpoint of that channel y, sends the other endpoint over x, then continues as P.
- The process $x(y).\,P$ denotes a *receive* action.
  It receives an endpoint over x, names it y, then continues as P.

The typing rules for send and receive are as follows:

$$\frac{P \vdash \Gamma, x : A \parallel \Delta, y : B}{x[y].\,P \vdash \Gamma, \Delta, x : A \otimes B}\ \text{T-Send} \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).\,P \vdash \Gamma, x : A \parr B}\ \text{T-Recv}$$

The behaviour of send and receive is given by the following rule:

$$(\nu x\bar{x})(x[y].\, P \parallel \bar{x}(\bar{y}).\, Q) \longrightarrow (\nu x\bar{x})(\nu y\bar{y})(P \parallel Q) \quad \text{E-Send}$$

The continuation of a send action is P, as opposed to CP's send, $x[y].\,(P \parallel Q)$, which requires that the endpoints y and x are immediately split between the processes P and Q. This is misleading. HCP *still* requires that the channels y and x are split between parallel processes. It does not require that the split happens *immediately*, settling for *eventually*. However, HCP's *eventually* is not as lenient as one might assume. As discussed in the introduction to Chapter 3, the T-Send rule does not permit the presence of any unrelated typing environments—i.e. there is no generic $\mathcal{G}$ in the rule, as there is in the T-New rule. The same is true for the typing rules for the other actions. Consequently, no other action may come between the send action x[y] and the parallel composition that splits x and y. The only process constructs that may come between those are unrelated name restrictions and parallel compositions. In effect, the only possible forms for a send action and its continuation are

$$x[y].\,\mathscr{E}[P \;{}_x\|_y\; Q] \quad \text{and} \quad x[y].\,\mathscr{E}[P \;{}_y\|_x\; Q].$$

Where $P \;{}_x\|_y\; Q$ denotes the parallel composition that splits x and y such that $x \in \mathrm{fn}(P)$ and $y \in \mathrm{fn}(Q)$. This may seem restrictive—*and it is*—but the purpose of HCP is to preserve the semantics of CP.

(The alternative—permitting unrelated typing environments—is explored in DHCP [Kokke et al., 2019a]. DHCP preserves the semantics of CLL, but not of CP. It uses *delayed* actions, where the actions in a process may to some extent resolve out of order.)

3.1.4 Close and Wait

The close and wait actions are dual:

- The process x[]. P denotes a *close* action.
  It sends a ping over x, then continues as P.
- The process x(). P denotes a *wait* action.
  It receives a ping over x, then continues as P.

(I say 'ping' to imply the interaction between close and wait is merely a synchronisation, and does not transmit any information.)

The typing rules for close and wait are as follows:

$$\frac{P \vdash \circ}{x[].\, P \vdash x : \mathbf{1}}\ \text{T-Close} \qquad \frac{P \vdash \Gamma}{x().\, P \vdash \Gamma, x : \bot}\ \text{T-Wait}$$

The behaviour of these two actions is given by the following rule:

$$(\nu x\bar{x})(x[].\, P \parallel \bar{x}().\, Q) \longrightarrow Q \quad \text{E-Close}$$

The continuation of the close action is P, as opposed to CP's close, $x[].\,0$, which requires that the process immediately terminates. This is misleading. HCP *still* requires that the process immediately terminates, but as a consequence of its type system rather than its process syntax. The T-Close rule requires that the continuation P is typed under the empty hyper-environment, and—as I will show Lemma 3.55—all processes typed under the empty hyper-environment are equivalent to the terminated process. In effect, the only possible forms for a close action and its continuation are variations of $x[].\,0 \parallel \ldots \parallel 0$.

3.1.5 Select and Offer

The select and offer actions are dual:

- The process $x \triangleleft \mathsf{inl}.\,P$ denotes a *left selection* action.
  It sends the label inl over x, then continues as P.
- The process $x \triangleleft \mathsf{inr}.\,P$ denotes a *right selection* action.
  It sends the label inr over x, then continues as P.
- The process $x \triangleright \{\mathsf{inl}\colon P; \mathsf{inr}\colon Q\}$ denotes a *choice* action.
  It receives a label over x, and then continues as either P or Q, depending on which label was received.

The typing rules for select and offer are as follows:

$$\frac{P \vdash \Gamma, x : A}{x \triangleleft \mathsf{inl}.\,P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_1 \qquad \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathsf{inl}.\,P \vdash \Gamma, x : A \oplus B}\ \text{T-Select}_2$$

$$\frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathsf{inl}\colon P; \mathsf{inr}\colon Q\} \vdash \Gamma, x : A \,\&\, B}\ \text{T-Offer}$$

The behaviour of these actions is given by the following rules:

$$(\nu x\bar{x})(x \triangleleft \mathsf{inl}.\,P \parallel \bar{x} \triangleright \{\mathsf{inl}\colon Q; \mathsf{inr}\colon R\}) \longrightarrow (\nu x\bar{x})(P \parallel Q)\ \text{E-Select}_1$$
$$(\nu x\bar{x})(x \triangleleft \mathsf{inr}.\,P \parallel \bar{x} \triangleright \{\mathsf{inl}\colon Q; \mathsf{inr}\colon R\}) \longrightarrow (\nu x\bar{x})(P \parallel R)\ \text{E-Select}_2$$

As discussed in § 2.1.5, this syntax was adapted from Dardha and Gay [2018], because is more easily generalized to variant types.

3.1.6 The Absurd Offer

The process $x \not\triangleright N$ denotes the *absurd offer*. It waits to receive a choice between *zero* alternatives. Such a choice cannot be made, which means that there is no corresponding select action, and no corresponding reduction rule. In essence, an absurd offer is inert. The absurd offer is the sole process that is allowed to leave endpoints unused, and the set of those unused endpoints is denoted by N.

$$\frac{N = \mathrm{fn}(\Gamma)}{x \not\triangleright N \vdash \Gamma, x : \top}\ \text{T-Absurd}$$

For an in-depth discussion of the absurd offer, its syntax, its inert semantics, and its relation to Wadler's absurd offer, see § 2.1.6. I present an exceptional semantics for the absurd offer in § 3.3.1.

3.2 Metatheory

In this section, I introduce the metatheory for Hypersequent Classical Processes. The principal developments are as follows:

- In § 3.2.1, I give several preliminary definitions that are used throughout the discussion of the metatheory.
- In § 3.2.2, I prove preservation (Proposition 3.30).
- In § 3.2.3, I define canonical form (Definition 3.33) and prove progress (Proposition 3.35). The proof of progress is adapted from the proof for CP (Proposition 2.32) and first appeared in Kokke et al. [2019b].
- In § 3.2.4, I define dependency graphs for HCP processes (Definition 3.36). I prove that HCP is deadlock-free, as its dependency graphs are always acyclic (Corollary 3.42), and I prove that my definition of canonical form is adequate (Corollary 3.50).
- In § 3.2.5, I define connection graphs for HCP processes (Definition 2.49) and prove that HCP's connection graphs are always forests (Proposition 3.52). The validity of right-branching forest form for HCP follows as a corollary.
- In § 3.2.6, I define Multiset CP, a process calculus whose processes are multisets of parallel CP processes. The principal use of Multiset CP is to clarify the correspondence between HCP and CP.
- In § 3.2.7, I define fission, the translation from Multiset CP into HCP, and its inverse, disentanglement-and-fusion, the translation from HCP into Multiset CP, and prove that these translations preserve types, structural congruence, and reduction.

3.2.1 Preliminaries

Configuration Contexts

Configuration contexts are multi-hole process contexts that consist only of name restrictions, parallel compositions, and the terminated process.

The notion of configuration contexts generalises neatly from CP to HCP. We follow the changes made to the process language, and decompose the cut into name restriction and parallel composition, and add a new constructor for the terminated process. The latter leads to an important

distinction between CP and HCP configuration contexts, which is that HCP configuration contexts can have *zero* holes.

Definition 3.6 (Configuration Context)**.** Configuration contexts *are n-hole process contexts, as defined by the following grammar:*

$$\mathscr{C}[\cdot], \mathscr{D}[\cdot] ::= \Box \mid 0 \mid (\nu x\bar{x})\mathscr{C}[\cdot] \mid \mathscr{C}[\cdot] \parallel \mathscr{D}[\cdot]$$

If there is risk of ambiguity, we explicitly write the number of holes in a configuration context with a superscript, e.g. as $\mathscr{C}^n[\cdot]$.

Plugging is defined by replacing the n holes with n processes, left to right:

$$\begin{array}{lll}
\Box & [P &] \triangleq P \\
0 & [&] \triangleq 0 \\
(\nu x\bar{x})\mathscr{C}^n[\cdot] & [P_1, \ldots, P_n &] \triangleq (\nu x\bar{x})(\mathscr{C}^n[P_1, \ldots, P_n]) \\
(\mathscr{C}^n[\cdot] \parallel \mathscr{D}^k[\cdot]) & [P_1, \ldots, P_n, P_{n+1}, \ldots, P_{n+k}] & \triangleq (\mathscr{C}^n[P_1, \ldots, P_n] \parallel \mathscr{D}^k[P_{n+1}, \ldots, P_{n+k}])
\end{array}$$

I write $\mathscr{C}[P_1, \ldots, \Box_i, \ldots, P_n]$ *for the evaluation context focused on the i'th hole in* $\mathscr{C}[\cdot]$ *such that* $\mathscr{C}[P_1, \ldots, \Box_i, \ldots, P_n][P_i] = \mathscr{C}[P_1, \ldots, P_i, \ldots, P_n]$.

I write $\mathrm{dn}(\mathscr{C}[\cdot])$ *for the unordered pairs of dual endpoints bound by* $\mathscr{C}[\cdot]$.

$$\begin{array}{lcl}
\mathrm{dn}(\Box) & \triangleq & \varnothing \\
\mathrm{dn}(0) & \triangleq & \varnothing \\
\mathrm{dn}((\nu x\bar{x})\mathscr{C}[\cdot]) & \triangleq & \{\{x, \bar{x}\}\} \cup \mathrm{dn}(\mathscr{C}[\cdot]) \\
\mathrm{dn}(\mathscr{C}[\cdot] \parallel \mathscr{D}[\cdot]) & \triangleq & \mathrm{dn}(\mathscr{C}[\cdot]) \cup \mathrm{dn}(\mathscr{D}[\cdot])
\end{array}$$

I write $\mathrm{bn}(\mathscr{C}[\cdot])$ *for the endpoints bound by* $\mathscr{C}[\cdot]$*, i.e.* $\mathrm{bn}(\mathscr{C}[\cdot]) \triangleq \bigcup \mathrm{dn}(\mathscr{C}[\cdot])$.

I write $\mathscr{C}^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \to \mathscr{G}$ *to mean that the configuration context* $\mathscr{C}^n[\cdot]$ *is well-typed under input hyper-environments* $\mathscr{G}_1, \ldots, \mathscr{G}_n$ *and output hyper-environment* $\mathscr{G}$*. I use "·" and "*$\mathscr{G}_1 \mid \ldots \mid \mathscr{G}_n$*" to denote sequences of hyper-environments, which correspond, in left-to-right order, to the holes in the configuration context.*

$$\frac{}{\Box \vdash \mathscr{G} \to \mathscr{G}} \qquad \frac{}{0 \vdash \cdot \to \circ}$$

$$\frac{\mathscr{C}^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \to \mathscr{G} \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})\mathscr{C}^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \to \mathscr{G} \parallel \Gamma, \Delta}$$

$$\frac{\mathscr{C}^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \to \mathscr{G} \qquad \mathscr{D}^k[\cdot] \vdash \mathscr{H}_{n+1} \mid \cdots \mid \mathscr{H}_k \to \mathscr{H}}{\mathscr{C}^n[\cdot] \parallel \mathscr{D}^k[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \mid \mathscr{H}_{n+1} \mid \cdots \mid \mathscr{H}_k \to \mathscr{G}, \mathscr{H}}$$

Shallow Structural Congruence

Reduction is closed under evaluation contexts, not under arbitrary process contexts, and only acts on the topmost actions. As such, reduction only needs a structural congruence that is similarly closed under evaluation contexts—a *shallow* structural congruence. Therefore, it is useful to distinguish different kinds of structural congruence, based on which portions of the process they are permitted to rewrite.

Definition 3.7 (Shallow Structural Congruence)**.** Shallow structural congruence*, written* $\overset{s}{\equiv}$*, is the smallest symmetric relation over processes that satisfies the rules in Figure 3.2 and is closed under evaluation contexts, as per the following rule:*

$$\textit{SC-Cong}\ \frac{P \overset{s}{\equiv} P'}{\mathscr{E}[P] \overset{s}{\equiv} \mathscr{E}[P']}$$

Most rules of the structural congruence target name restriction and parallel composition. The odd one out is SC-LinkComm, which rewrites a link action. It will be useful to single out the portions of a structural congruence that rewrite links.

Definition 3.8 (Link-Preserving Structural Congruence)**.** Link-preserving structural congruence*, written* $\overset{l}{\equiv}$*, is the congruence closure over processes that satisfies the rules in Figure 3.2 except for SC-LinkComm.*

Finally, it will be useful to have variants which combine these restrictions. In practice, I only need link-preserving shallow structural congruence and deep structural congruence.

Definition 3.9 (Link-Preserving Shallow Structural Congruence)**.** Link-preserving shallow structural congruence*, written* $\overset{ls}{\equiv}$*, is the intersection of link-preserving and shallow structural congruence.*

Definition 3.10 (Deep Structural Congruence)**.** Deep structural congruence*, written* $\overset{d}{\equiv}$*, is the equivalence closure of the complement of* $\overset{ls}{\equiv}$ *with respect to* $\equiv$*.*

Any structural congruence can be decomposed into its link-preserving shallow and deep structural components.

Lemma 3.11. *If* $P \equiv Q$*, then there exists some* R *such that* $P \overset{ls}{\equiv} R$ *and* $R \overset{d}{\equiv} Q$*.*

Ready Processes and Threads

A process is ready if it is ready to perform some communication action, i.e. if it is a link or it is prefixed by an action. (The definition of ready remains essentially unchanged from CP, though, of course, the forms of send and close actions has changed.)

Definition 3.12 (Ready)**.** *A process* P *is* ready to act on x*, written* ready(P, x)*, if it is of one of the forms:*

$$\begin{array}{llllll} x \leftrightarrow y & x[y].\,P & x[].\,P & x \triangleleft \mathit{inl}.\,P & x \triangleright \{\mathit{inl}\colon P; \mathit{inr}\colon Q\} \\ y \leftrightarrow x & x(y).\,P & x().\,P & x \triangleleft \mathit{inr}.\,P & x \not\leftrightarrow N \end{array}$$

A process is ready *if it ready to act on some endpoint.*

In particular, that links $x \leftrightarrow y$ are considered ready to act on both x and y, and absurd $x \lightning N$ is *not* considered ready to act on the channels $y \in N$.

A process can be decomposed into a prefix of its cuts, and a series of threads connected by those cuts. Such a prefix is the *maximum* configuration context, in the sense that no further cuts can be added. (The definition of maximal configuration contexts is adjusted to permit configuration contexts with zero holes, i.e. the maximum configuration contexts for terminated processes, but otherwise unchanged.)

Definition 3.13 (Maximum Configuration Context)**.** *The* maximum configuration context $\mathscr{C}^n[\cdot]$ *of a process* P *is the configuration context such that* $P = \mathscr{C}^n[P_1, \ldots, P_n]$ *(for some* $n \geq 0$*) and (for* $1 \leq i \leq n$*) each* P_i *is ready. The processes* P_i *are the* threads *of* P*. Every process has a unique maximum configuration context.*

Likewise, evaluation contexts are maximal if no further cuts can be added. Informally, maximal evaluation contexts are paths to the threads contained within some process, so each maximum configuration context $\mathscr{C}^n[\cdot]$ gives us n distinct maximal evaluation contexts. (The definition of maximal evaluation contexts is unchanged from CP.)

Definition 3.14 (Maximal Evaluation Context)**.** *A* maximal evaluation context $\mathscr{E}$ *of a process* P *is an evaluation context such that* $P = \mathscr{E}[Q]$ *and* Q *is ready. If* $\mathscr{C}[\cdot]$ *is the maximum configuration context of* P*, then* $\mathscr{C}[P_1, \ldots, \square_i, \ldots, P_n]$ *is a maximal evaluation context of* P*.*

Finally, we refer to the top-level ready processes as *threads*. A significant portion of HCP's metatheory deals with threads. Let the metavariable T range over threads.

Definition 3.15 (Thread)**.** *A process* P *is a* thread of Q *if there exists some evaluation context* $\mathscr{E}$ *of* Q *such that* $Q = \mathscr{E}[P]$ *and* P *is ready. We say that* Q contains the thread P *to mean that* P *is a thread of* Q*. We say* P *is a* thread *when the process* Q *that* P *is a thread of can be inferred from context. Let* T *range over threads.*

The use of the thread metavariable allows us to succinctly decompose any process P into its maximum configuration context and its threads, by stating "P is of the form $\mathscr{C}[T_1, \ldots, T_n]$", as the notation implies that each thread T_i is a ready process, and therefore that $\mathscr{C}[\cdot]$ is the maximum configuration context.

Process Contexts

For the occasional convenience, I define full n-hole process contexts, which are arbitrary processes with any number of holes. Process contexts may contain any process construct, and may contain holes in any position, including nested under actions. As such, process contexts generalise evaluation and configuration contexts.

Definition 3.16 (Process Context)**.** *Process contexts are defined by the grammar for processes, extended with the hole constructor, written* $\square$*. A process context may have any number of holes.*

The names $P[\cdot]$*,* $Q[\cdot]$*, and* $R[\cdot]$ *range over process contexts, where the trailing* $[\cdot]$ *is intended to help distinguish between process and process context metavariables, and denotes the position of the arguments for plugging. We write* $P^n[\cdot]$ *to denote that the process context* $P[\cdot]$ *has n holes.*

Plugging, written $P[P_1, \ldots, P_n]$*, is defined by replacing the n holes in the process context* $P[\cdot]$ *with the processes* $P_1, \ldots, P_n$ *in order from left to right.*

Linearity

HCP has a *linear* type system. It ensures that resources are always used exactly once, and never copied or dropped. However, due to HCP's reuse of endpoint names, it may appear that resources are used multiple times. (For instance, the process $x().\, x[].\, 0$ appears to use the endpoint x twice.)

I define linearity by means of a free name count—taking the sum across a parallel composition, the union across an offer, and counting the absurd offer as using all available resources. The definition of the free name count is easily adapted to HCP. (For an in-depth discussion, see § 2.2.1.)

Definition 3.17 (Free Name Count)**.** *The multiset of free endpoints in* P*, written* $\underline{\mathrm{fn}}(P)$*, is a multiset (see Definition A.3) with support set* $\mathrm{fn}(P)$ *and multiplicity function* $\mu_{\mathrm{fn}(P)}$*.*

$$
\begin{array}{lcl}
\underline{\mathrm{fn}}(x \leftrightarrow y) & \triangleq & \{\!|x, y|\!\} \\
\underline{\mathrm{fn}}((\nu x \bar{x})P) & \triangleq & \underline{\mathrm{fn}}(P) \setminus \{x, \bar{x}\} \\
\underline{\mathrm{fn}}(P \parallel Q) & \triangleq & \underline{\mathrm{fn}}(P) + \underline{\mathrm{fn}}(Q) \\
\underline{\mathrm{fn}}(0) & \triangleq & \{\!|\,|\!\} \\
\underline{\mathrm{fn}}(x[y].\, P) & \triangleq & \{\!|x|\!\} + (\underline{\mathrm{fn}}(P) \setminus \{x, y\}) \\
\underline{\mathrm{fn}}(x[].\, 0) & \triangleq & \{\!|x|\!\} \\
\underline{\mathrm{fn}}(x(y).\, P) & \triangleq & \{\!|x|\!\} + (\underline{\mathrm{fn}}(P) \setminus \{x, y\}) \\
\underline{\mathrm{fn}}(x().\, P) & \triangleq & \{\!|x|\!\} + \underline{\mathrm{fn}}(P) \\
\underline{\mathrm{fn}}(x \triangleleft \mathit{inl}.\, P) & \triangleq & \{\!|x|\!\} + (\underline{\mathrm{fn}}(P) \setminus \{x\}) \\
\underline{\mathrm{fn}}(x \triangleleft \mathit{inr}.\, P) & \triangleq & \{\!|x|\!\} + (\underline{\mathrm{fn}}(P) \setminus \{x\}) \\
\underline{\mathrm{fn}}(x \triangleright \{\mathit{inl}\colon P; \mathit{inr}\colon Q\}) & \triangleq & \{\!|x|\!\} + ((\underline{\mathrm{fn}}(P) \setminus \{x\}) \cup (\underline{\mathrm{fn}}(Q) \setminus \{x\})) \\
\underline{\mathrm{fn}}(x \not\triangleright N) & \triangleq & \{\!|x|\!\} + \{\!|\, w \mid w \in N |\!\}
\end{array}
$$

Note that the operation $\mathscr{X} \setminus X$ *removes all occurrences of the elements in the set* X *from the multiset* $\mathscr{X}$ *(see Definition A.3).*

Linearity states that, for well-typed processes, each endpoint in the typing environment is used exactly once in the process, and vice versa.

Proposition 3.18 (Linearity)**.** *If* $P \vdash \mathscr{G}$*, then:*

- $x \in^k \underline{\mathrm{fn}}(P) \implies k = 1$

- $x \in^1 \underline{\mathrm{fn}}(P) \iff x \in \mathrm{fn}(\mathscr{G})$

Proof. By induction on the derivation of $P \vdash \mathscr{G}$.

□

For any well-typed process $\mathscr{E}[P]$, each endpoint bound by $\mathscr{E}$ is used exactly once in the process P, and vice versa.

Corollary 3.19. *If $\mathscr{E}[P] \vdash \mathscr{G}$, then:*

- $x \in^k \underline{\mathrm{fn}}(P) \implies k = 1$
- $x \in^1 \underline{\mathrm{fn}}(P) \iff x \in \mathrm{bn}(\mathscr{E})$
- $\mathrm{bn}(\mathscr{E}) \subseteq \mathrm{fn}(P)$

For any well-typed configuration $\mathscr{C}^n[P_1, \ldots, P_n]$, the processes $P_1, \ldots, P_n$ must collectively use all the endpoints bound by $\mathscr{C}^n[\cdot]$ exactly once.

Corollary 3.20. *If $\mathscr{C}^n[P_1, \ldots, P_n] \vdash \mathscr{G}$, then:*

- $x \in^k \bigcup_{1 \le i \le n} \underline{\mathrm{fn}}(P_i) \implies k = 1$
- $x \in^1 \bigcup_{1 \le i \le n} \underline{\mathrm{fn}}(P_i) \iff x \in \mathrm{bn}(\mathscr{C}[\cdot])$
- $\mathrm{bn}(\mathscr{C}[\cdot]) \subseteq \bigcup_{1 \le i \le n} \mathrm{fn}(P_i)$

Separation

Separation relates configuration contexts and evaluation contexts—it 'zooms in', from viewing a process as a series of connected processes, to viewing two specific processes and the cut connecting them. Separation also captures an essential property of HCP's type system: dual endpoints must be in distinct processes, separated by a parallel composition.

Whereas CP has one separation lemma, HCP has two. One for parallel composition, and one for name restriction.

Lemma 3.21. *If $P \vdash \mathscr{G} \| \Gamma, x : A \| \Delta, \bar{x} : \overline{A}$, $\mathscr{C}^n[\cdot]$ is a configuration context such that $P = \mathscr{C}^n[P_1, \ldots, P_n]$ (for some $n \ge 2$), and there exists some $\{x, \bar{x}\} \in \mathrm{fn}(P)$ such that $x \in \mathrm{fn}(P_i)$ and $\bar{x} \in \mathrm{fn}(P_j)$ (for some $1 \le i, j \le n$), there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that*

1. $P = \mathscr{E}[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]]$, *or*
2. $P = \mathscr{E}[\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i]]$.

Proof. By induction on the structure of $\mathscr{C}[\cdot]$, followed by inversion on P and the corresponding typing derivation.

There are two cases:

- Case $\mathscr{C}^n[\cdot]$ is of the form $(\nu y \bar{y})\mathscr{C}[\cdot]$ (reusing $\mathscr{C}[\cdot]$).

  By the induction on $\mathscr{C}[\cdot]$, there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $(\nu y \bar{y})\square$ to $\mathscr{E}$.

- Case $\mathscr{C}^n[\cdot]$ is of the form $\mathscr{D}_1^n[\cdot] \parallel \mathscr{D}_2^m[\cdot]$ (reusing n).

  By inversion, $P = \mathscr{D}_1^n[P_1, ..., P_n] \parallel \mathscr{D}_2^m[P_{n+1}, ..., P_{n+m}]$. There are four subcases, depending on whether P_i and P_j are on the same or different sides of the parallel composition.

  - Subcase $1 \le i, j \le n$.

    By induction on $\mathscr{D}_1^n[\cdot]$, there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $\square \parallel \mathscr{D}_2^m[P_{n+1}, ..., P_{n+m}]$ to $\mathscr{E}$.

  - Subcase $n < i, j \le m$.

    By induction on $\mathscr{D}_2^m[\cdot]$, there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $\mathscr{D}_1^n[P_1, ..., P_n] \parallel \square$ to $\mathscr{E}$.

  - Subcase $1 \le i \le n < j \le m$.

    Let $\mathscr{E} = \square$, $\mathscr{F}_i = \mathscr{D}_1[P_1, \square_i, P_n]$, and $\mathscr{F}_j = \mathscr{D}_2[P_{n+1}, \square_j, P_{n+m}]$.

    The result follows as (1).

  - Subcase $1 \le j \le n < i \le m$.

    Let $\mathscr{E} = \square$, $\mathscr{F}_i = \mathscr{D}_1[P_{n+1}, \square_i, P_{n+m}]$, and $\mathscr{F}_j = \mathscr{D}_2[P_1, \square_j, P_n]$.

    The result follows as (2).

□

Lemma 3.22 (Separation). *If $P \vdash \mathscr{G}$, and $\mathscr{C}^n[\cdot]$ is a configuration context such that $P = \mathscr{C}^n[P_1, ..., P_n]$ (for some $n \ge 2$), and there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$ such that $x \in \mathrm{fn}(P_i)$ and $\bar{x} \in \mathrm{fn}(P_j)$ (for some $1 \le i, j \le n$), there exist $\mathscr{E}_1$, $\mathscr{E}_2$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that either*

1. $P = \mathscr{E}_1[(\nu x\bar{x})(\mathscr{E}_2[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]])]$,
2. $P = \mathscr{E}_1[(\nu \bar{x}x)(\mathscr{E}_2[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]])]$,
3. $P = \mathscr{E}_1[(\nu x\bar{x})(\mathscr{E}_2[\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i]])]$, *or*
4. $P = \mathscr{E}_1[(\nu \bar{x}x)(\mathscr{E}_2[\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i]])]$.

Proof. By induction on the structure of $\mathscr{C}[\cdot]$, followed by inversion on P and the corresponding typing derivation.

There are two cases:

- Case $\mathscr{C}^n[\cdot]$ is of the form $(\nu y\bar{y})\mathscr{C}^n[\cdot]$ (reusing $\mathscr{C}[\cdot]$).

  By inversion, $P = (\nu y\bar{y})(\mathscr{D}_1^n[P_i, ..., P_n])$.

  There are three subcases:

  - Subcase $x = y$ and $\bar{x} = \bar{y}$.

    By Lemma 3.21, there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that:

    a. $P = \mathscr{E}[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]]$, or

b. $P = \mathscr{E}[\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i]]$.

Let $\mathscr{E}_1 = \Box$ and $\mathscr{E}_2 = \mathscr{E}$.

In case (a), the result follows as case (1).

In case (b), the result follows as case (3).

– Subcase $x = \bar{y}$ and $\bar{x} = y$.

By Lemma 3.21, there exist $\mathscr{E}$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that:

a. $P = \mathscr{E}[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]]$, or
b. $P = \mathscr{E}[\mathscr{F}_j[P_j] \parallel \mathscr{F}_i[P_i]]$.

Let $\mathscr{E}_1 = \Box$ and $\mathscr{E}_2 = \mathscr{E}$.

In case (a), the result follows as case (2).

In case (b), the result follows as case (4).

– Subcase $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$.

By induction on $\mathscr{C}[\cdot]$, there exist $\mathscr{E}_1$, $\mathscr{E}_2$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $(\nu y\bar{y})\Box$ to $\mathscr{E}_1$.

• Case $\mathscr{C}^n[\cdot]$ is of the form $\mathscr{D}_1^n[\cdot] \parallel \mathscr{D}_2^k[\cdot]$ (reusing n).

By inversion, $P = \mathscr{D}_1^n[P_i, \ldots, P_n] \parallel \mathscr{D}_2^k[P_{n+1}, \ldots, P_k]$. By inversion on the fact that $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$, P_i and P_j must be on the same side of the parallel composition. There are two subcases:

– Subcase $1 \le i, j \le n$.

By induction on $\mathscr{D}_1^n[\cdot]$, there exist $\mathscr{E}_1$, $\mathscr{E}_2$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $\Box \parallel \mathscr{D}_2^k[P_{n+1}, \ldots, P_k]$ to $\mathscr{E}_1$.

– Subcase $n < i, j \le k$.

By induction on $\mathscr{D}_2^k[\cdot]$, there exist $\mathscr{E}_1$, $\mathscr{E}_2$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that (1) or (2). The result follows by prepending $\mathscr{D}_1^n[P_i, \ldots, P_n] \parallel \Box$ to $\mathscr{E}_1$.

□

The separation lemma is rather precise, and gives us one of four equalities. However, all cases are equivalent up to structural congruence. Usually, it is easier to forget the exact case.

Corollary 3.23 (Separation). *If $P \vdash \mathscr{G}$, $\mathscr{C}^n[\cdot]$ is a configuration context such that $P = \mathscr{C}^n[P_1, \ldots, P_n]$ (for some $n \ge 2$), and there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$ such that $x \in \mathrm{fn}(P_i)$ and $\bar{x} \in \mathrm{fn}(P_j)$ (for some $1 \le i, j \le n$), there exist $\mathscr{E}_1$, $\mathscr{E}_2$, $\mathscr{F}_i$, and $\mathscr{F}_j$ such that $P \overset{\text{LS}}{\equiv} \mathscr{E}_1[(\nu x\bar{x})(\mathscr{E}_2[\mathscr{F}_i[P_i] \parallel \mathscr{F}_j[P_j]])]$.*

Proof. By Lemma 3.22, SC-NewComm, and SC-ParComm.

□

Evaluation contexts commute with name restriction and parallel composition.

Lemma 3.24. *If* $x, \bar{x} \notin \mathrm{fn}(\mathscr{E}) \cup \mathrm{bn}(\mathscr{E})$, *then* $\mathscr{E}[(\nu x\bar{x})P] \overset{\text{LS}}{\equiv} (\nu x\bar{x})(\mathscr{E}[P])$.

Proof. By induction on the structure of the evaluation context $\mathscr{E}$.

□

Lemma 3.25. *If* $\mathrm{fn}(P) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$, *then* $\mathscr{E}[P \parallel Q] \overset{\text{LS}}{\equiv} P \parallel \mathscr{E}[Q]$.

Proof. By induction on the structure of the evaluation context $\mathscr{E}$.

□

Corollary 3.26. *If* $\mathrm{fn}(P) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$, *then* $\mathscr{E}[P] \overset{\text{LS}}{\equiv} \mathscr{E}[0] \parallel P$.

Proof. By SC-ParNil and Lemma 3.25.

□

3.2.2 Preservation

Structural congruence preserves typing. If some process P is well-typed and is equivalent to some process Q under structural congruence, then Q is well-typed under the same typing environment.

Lemma 3.27. *If* $P \equiv Q$, *then* $P \vdash \mathscr{G}$ *if and only if* $Q \vdash \mathscr{G}$.

Proof. By induction on the derivation of the equivalence $P \equiv Q$.

The case for reflexivity follows immediately. The cases for symmetry and transitivity follow immediately by induction. The case for congruence closure follows by induction and the injectivity of the type derivation rules. The cases for applications of SC-LinkComm, SC-ParNil, SC-ParComm, SC-ParAssoc, SC-NewComm, and SC-ScopeExt are as follows, presented as equivalences on type derivations:

- Case SC-LinkComm:

$$\frac{}{x{\leftrightarrow}y \vdash x : A, y : \overline{A}} \equiv \frac{\dfrac{}{y{\leftrightarrow}x \vdash x : \overline{\overline{A}}, y : \overline{A}}}{y{\leftrightarrow}x \vdash x : A, y : \overline{A}}\ \text{Lemma 2.1}$$

- Case SC-ParNil.

$$\frac{P \vdash \mathscr{G} \qquad \dfrac{}{0 \vdash \circ}}{P \parallel 0 \vdash \mathscr{G}} \equiv P \vdash \mathscr{G}$$

- Case SC-ParComm.

$$\frac{P_1 \vdash \mathscr{G}_1 \qquad P_2 \vdash \mathscr{G}_2}{P_1 \parallel P_2 \vdash \mathscr{G}_1 \parallel \mathscr{G}_2} \equiv \frac{P_2 \vdash \mathscr{G}_2 \qquad P_1 \vdash \mathscr{G}_1}{P_2 \parallel P_1 \vdash \mathscr{G}_1 \parallel \mathscr{G}_2}$$

- Case SC-ParAssoc.

$$\dfrac{P_1 \vdash \mathscr{G}_1 \qquad \dfrac{P_2 \vdash \mathscr{G}_2 \qquad P_3 \vdash \mathscr{G}_3}{P_2 \parallel P_3 \vdash \mathscr{G}_2 \parallel \mathscr{G}_3}}{P_1 \parallel (P_2 \parallel P_3) \vdash \mathscr{G}_1 \parallel \mathscr{G}_2 \parallel \mathscr{G}_3} \equiv \dfrac{\dfrac{P_1 \vdash \mathscr{G}_1 \qquad P_2 \vdash \mathscr{G}_2}{P_1 \parallel P_2 \vdash \mathscr{G}_1 \parallel \mathscr{G}_2} \qquad P_3 \vdash \mathscr{G}_3}{P_1 \parallel (P_2 \parallel P_3) \vdash \mathscr{G}_1 \parallel \mathscr{G}_2 \parallel \mathscr{G}_3}$$

- Case SC-NewComm.

$$\dfrac{P \vdash \mathscr{G} \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}{(\nu x\bar{x})P \vdash \mathscr{G} \parallel \Gamma, \Delta} \equiv \dfrac{\dfrac{P \vdash \mathscr{G} \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}{P \vdash \mathscr{G} \parallel \Gamma, x : \overline{\overline{A}} \parallel \Delta, \bar{x} : \overline{A}}\ \text{Lemma 2.1}}{(\nu \bar{x} x)P \vdash \mathscr{G} \parallel \Gamma, \Delta}$$

- Case SC-NewAssoc.

$$\dfrac{\dfrac{\dfrac{P \vdash \mathscr{G} \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A} \parallel \Delta_1, y : B \parallel \Delta_2, \bar{y} : \overline{B}}{(\nu y\bar{y})P \vdash \mathscr{G} \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A} \parallel \Delta_1, \Delta_2}}{(\nu x\bar{x})(\nu y\bar{y})P \vdash \mathscr{G} \parallel \Gamma_1, \Gamma_2 \parallel \Delta_1, \Delta_2}}{\dfrac{\dfrac{P \vdash \mathscr{G} \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A} \parallel \Delta_1, y : B \parallel \Delta_2, \bar{y} : \overline{B}}{(\nu x\bar{x})P \vdash \mathscr{G} \parallel \Gamma_1, \Gamma_2 \parallel \Delta_1, y : B \parallel \Delta_2, \bar{y} : \overline{B}}}{(\nu y\bar{y})(\nu x\bar{x})P \vdash \mathscr{G} \parallel \Gamma_1, \Gamma_2 \parallel \Delta_1, \Delta_2}}$$

- Case SC-ScopeExt.

$$\dfrac{\dfrac{\dfrac{P \vdash \mathscr{G}_1 \qquad Q \vdash \mathscr{G}_2 \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A}}{P \parallel Q \vdash \mathscr{G}_1 \parallel \mathscr{G}_2 \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A}}}{(\nu x\bar{x})(P \parallel Q) \vdash \mathscr{G}_1 \parallel \mathscr{G}_2 \parallel \Gamma_1, \Gamma_2}}{\dfrac{P \vdash \mathscr{G}_1 \qquad \dfrac{Q \vdash \mathscr{G}_2 \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A}}{(\nu x\bar{x})Q \vdash \mathscr{G}_2 \parallel \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A}}}{P \parallel (\nu x\bar{x})Q \vdash \mathscr{G}_1 \parallel \mathscr{G}_2 \parallel \Gamma_1, \Gamma_2}}$$

  By inversion, these derivations are the only ones, as $x, \bar{x} \notin \mathrm{fn}(P)$.

□

Renaming preserves typing. If a process is well-typed, then renaming any free endpoint does not affect its typing.

Lemma 3.28. *If* $P \vdash \mathscr{G} \parallel \Gamma, x : A$, *then* $P\{w/x\} \vdash \mathscr{G} \parallel \Gamma, w : A$.

Proof. The result follows by induction.

□

Plugging with any form of process context preserves typing.

Lemma 3.29. *If* $P^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \mathscr{G}_n \to \mathscr{G}$ *and* $P_i \vdash \mathscr{G}_i$ *(for* $1 \le i \le n$*),* $P^n[P_1, \ldots, P_n] \vdash \mathscr{G}$.

Proof. By induction on the derivation of $P^n[\cdot] \vdash \mathscr{G}_1 \mid \cdots \mid \Gamma_n \to \mathscr{G}$.

□

Reduction preserves typing. If a process P is well-typed and reduces to some other process Q, then Q is well-typed under the same typing environment.

Proposition 3.30 (Preservation)**.** *If* $P \vdash \mathscr{G}$ *and* $P \longrightarrow Q$*, then* $Q \vdash \mathscr{G}$.

Proof. By induction on the derivation of the reduction $P \longrightarrow Q$.

The case for E-Cong follows by induction and Lemma 3.29. The case for E-Equiv follows by induction and Lemma 3.27. The cases for the rules E-Link, E-Send, E-Close, E-Select$_1$, and E-Select$_2$ are as follows, presented as reductions on type derivations:

- Case E-Link:

$$\dfrac{\dfrac{\dfrac{x{\leftrightarrow}w \vdash x : A, w : \overline{A} \qquad P \vdash \Gamma, \bar{x} : \overline{A}}{x{\leftrightarrow}w \parallel P \vdash x : A, w : \overline{A} \parallel \Gamma, \bar{x} : \overline{A}}}{(\nu x\bar{x})(x{\leftrightarrow}w \parallel P) \vdash \Gamma, w : \overline{A}}}{\dfrac{P \vdash \Gamma, \bar{x} : \overline{A}}{P\{w/\bar{x}\} \vdash \Gamma, w : \overline{A}}\ \text{Lemma 2.25}}$$

- Case E-Send:

$$\dfrac{\dfrac{\dfrac{\dfrac{P \vdash \Gamma_1, y : A \qquad Q \vdash \Gamma_2, x : B}{P \parallel Q \vdash \Gamma_1, y : A \parallel \Gamma_2, x : B}}{x[y].(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : A \otimes B} \qquad \dfrac{R \vdash \Gamma_3, \bar{y} : \overline{A}, \bar{x} : \overline{B}}{\bar{x}(\bar{y}).\, R \vdash \Gamma_3, \bar{x} : \overline{A} \parr \overline{B}}}{x[y].(P \parallel Q) \parallel \bar{x}(\bar{y}).\, R \vdash \Gamma_1, \Gamma_2, x : A \otimes B \parallel \Gamma_3, \bar{x} : \overline{A} \parr \overline{B}}}{(\nu x\bar{x})(x[y].(P \parallel Q) \parallel \bar{x}(\bar{y}).\, R) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

$$\dfrac{\dfrac{P \vdash \Gamma_1, y : A \qquad \dfrac{\dfrac{Q \vdash \Gamma_2, x : B \qquad R \vdash \Gamma_3, \bar{y} : \overline{A}, \bar{x} : \overline{B}}{Q \parallel R \vdash \Gamma_2, x : B \parallel \Gamma_3, \bar{y} : \overline{A}, \bar{x} : \overline{B}}}{(\nu x\bar{x})(Q \parallel R) \vdash \Gamma_2, \Gamma_3, \bar{y} : \overline{A}}}{P \parallel (\nu x\bar{x})(Q \parallel R) \vdash \Gamma_1, y : A \parallel \Gamma_2, \Gamma_3, \bar{y} : \overline{A}}}{(\nu y\bar{y})(P \parallel (\nu x\bar{x})(Q \parallel R)) \vdash \Gamma_1, \Gamma_2, \Gamma_3}$$

- Case E-Close:

$$\dfrac{\dfrac{x[].\,0 \vdash x : \mathbf{1} \qquad \dfrac{Q \vdash \Gamma_2}{\bar{x}().\, Q \vdash \Gamma_2, \bar{x} : \bot}}{x[].\,0 \parallel \bar{x}().\, Q \vdash x : \mathbf{1} \parallel \Gamma_2, \bar{x} : \bot}}{(\nu x\bar{x})(x[].\,0 \parallel \bar{x}().\, Q) \vdash \Gamma_2} \longrightarrow Q \vdash \Gamma_2$$

- Case E-SELECT$_1$:

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : A}{x \triangleleft \mathsf{inl}.\, P \vdash \Gamma_1, x : A \oplus B} \quad \dfrac{Q \vdash \Gamma_2, \bar{x} : \overline{A} \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{\bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\} \vdash \Gamma_2, \bar{x} : \overline{A} \,\&\, \overline{B}}}{\dfrac{x \triangleleft \mathsf{inl}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\} \vdash \Gamma_1, x : A \oplus B \parallel \Gamma_2, \bar{x} : \overline{A} \,\&\, \overline{B}}{(\nu x \bar{x})(x \triangleleft \mathsf{inl}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\}) \vdash \Gamma_1, \Gamma_2}}$$

$$\downarrow$$

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : A \qquad Q \vdash \Gamma_2, \bar{x} : \overline{A}}{P \parallel Q \vdash \Gamma_1, x : A \parallel \Gamma_2, \bar{x} : \overline{A}}}{(\nu x \bar{x})(P \parallel Q) \vdash \Gamma_1, \Gamma_2}$$

- Case E-SELECT$_2$:

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : B}{x \triangleleft \mathsf{inr}.\, P \vdash \Gamma_1, x : A \oplus B} \quad \dfrac{Q \vdash \Gamma_2, \bar{x} : \overline{A} \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{\bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\} \vdash \Gamma_2, \bar{x} : \overline{A} \,\&\, \overline{B}}}{\dfrac{x \triangleleft \mathsf{inr}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\} \vdash \Gamma_1, x : A \oplus B \parallel \Gamma_2, \bar{x} : \overline{A} \,\&\, \overline{B}}{(\nu x \bar{x})(x \triangleleft \mathsf{inr}.\, P \parallel \bar{x} \triangleright \{\mathsf{inl}: Q; \mathsf{inr}: R\}) \vdash \Gamma_1, \Gamma_2}}$$

$$\downarrow$$

$$\dfrac{\dfrac{P \vdash \Gamma_1, x : B \qquad R \vdash \Gamma_2, \bar{x} : \overline{B}}{P \parallel R \vdash \Gamma_1, x : B \parallel \Gamma_2, \bar{x} : \overline{B}}}{(\nu x \bar{x})(P \parallel R) \vdash \Gamma_1, \Gamma_2}$$

□

3.2.3 Progress

What should the canonical forms be for processes in HCP?

Unlike CP, HCP has syntax for the terminated process. We could make the distinction between *normal form* and *neutral form*. The normal form is the terminated process. The neutral forms are processes stuck on a free name, i.e. processes whose communication is blocked on a free channel.

Is that distinction meaningful? No, not in the context of HCP's reduction semantics. HCP processes do not *reduce to* normal form. After all, HCP preserves CP's semantics. Any process that does *anything* is never done.

Let us stick with the terminology "canonical form" for now. CP's definition for canonical form is adapted almost effortlessly. An HCP process P is in canonical form if and only if

1. P contains no link thread ready to act on a bound endpoint; and
2. P contains no two processes ready to act on dual endpoints.

Case (1) rules out α-reduction—any link evaluating as α-renaming. Case (2) rules out β-reduction—any other reduction. (The definitions of α- and β-reduction are unchanged from § 2.2.3.)

Definition 3.31 (α-Reduction)**.** *A process* P *α-reduces to* Q*, written* $P \longrightarrow_\alpha Q$*, if there exists a reduction* $P \longrightarrow Q$ *which only uses the rules E-Link, E-Cong, and E-Equiv.*

Definition 3.32 (β-Reduction)**.** *A process* P *β-reduces to* Q*, written* $P \longrightarrow_\beta Q$*, if there exists a reduction* $P \longrightarrow Q$ *that does not use the rule E-Link.*

I divide reduction into α-reduction, which captures link evaluating as α-renaming, and β-reduction, which captures all other reduction. In essence, α-reduction captures asynchronous reduction, and β-reduction captures synchronous reduction.

Canonical forms are defined, as described above, by the forms of processes that cannot α- or β-reduce. The definition does not require that terminated processes are normalised to 0, e.g. $0 \parallel 0$ is considered canonical. This simplifies the statement of progress, as we do not need to make allowances for processes such as $0 \parallel 0$ that do not reduce, but are not exactly 0, and lets us defer proving Lemma 3.55. (The definition for canonical forms is adjusted to permit maximum configuration contexts with zero holes, i.e. the canonical forms of terminated processes.)

Definition 3.33 (Canonical Form)**.** *A process* P *is in* canonical form*, written* canonical(P)*, if* P *is of the form* $\mathscr{C}^n[T_1, \ldots, T_n]$ *(for some* $n \geq 0$*) and (for* $1 \leq i, j \leq n$*) and*

1. *no* T_i *is a link thread ready to act on an endpoint* $x \in \mathrm{bn}(\mathscr{C}[\cdot])$*; and*
2. *no* T_i *and* T_j *are ready to act on dual endpoints* $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$*.*

If condition (1) holds, $P \not\longrightarrow_\alpha$*. If condition (2) holds,* $P \not\longrightarrow_\beta$*.*

If a process contains two processes ready to act on dual endpoints, then it can reduce. The following lemma abstracts over the reduction rules for HCP's many dual actions, and is unchanged from CP.

Lemma 3.34 (Reduction)**.** *If* $(\nu x\bar{x})(P \parallel Q) \vdash \mathscr{G}$*, and* P *and* Q *are ready to act on* x *and* $\bar{x}$*, respectively, there exists some* R *such that* $(\nu x\bar{x})(P \parallel Q) \longrightarrow R$*.*

Proof. By inversion on the derivation of $(\nu x\bar{x})(P \parallel Q) \vdash \mathscr{G}$.

There are two cases, where either P or Q is a link, which correspond to E-Link and its symmetric variant under E-Equiv with SC-ParComm.

There are eight cases that correspond exactly to E-Send, E-Close, E-Select$_1$, E-Select$_2$, and their symmetric variants under E-Equiv with SC-ParComm.

□

Progress states that any process is either in canonical form or can reduce. In essence, the proof shows that conditions (1) and (2) of the definition of canonical form correspond to the absence of α- and β-reduction.

Proposition 3.35 (Progress)**.** *If* $P \vdash \mathscr{G}$*, then either* P *is in canonical form, or there exists some* Q *such that* $P \longrightarrow Q$*.*

Proof. The process P is of the form $\mathscr{C}^n[T_1, \ldots, T_n]$ (for some $n \geq 0$).

If P is in canonical form, the result follows.

Otherwise, $n \geq 2$, and there are two cases:

Condition (1) does not hold. Some T_i (for $1 \leq i \leq n$) is a link thread ready to act on an endpoint $x \in \mathrm{bn}(\mathscr{C}[\cdot])$. By SC-LinkComm, $T_i \overset{s}{\equiv} x{\leftrightarrow}y$. By definition, there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$. By Corollary 3.23 and Proposition 3.18, there exists some T_j (for $1 \leq j \leq n$ and $i \neq j$) such that $\bar{x} \in \mathrm{fn}(T_j)$.

$$
\begin{array}{lll}
P & \overset{s}{\equiv} \mathscr{E}_1[(\nu x\bar{x})(\mathscr{E}_2[\mathscr{F}_1[x{\leftrightarrow}y] \parallel \mathscr{F}_2[T_j]])] & \langle\text{by Corollary 3.23 and SC-LinkComm}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[(\nu x\bar{x})(\mathscr{F}_1[x{\leftrightarrow}y] \parallel \mathscr{F}_2[T_j])]] & \langle\text{by Lemma 3.24}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_1[(\nu x\bar{x})(x{\leftrightarrow}y \parallel \mathscr{F}_2[T_j])]]] & \langle\text{by Lemma 3.25}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_1[\mathscr{F}_2[(\nu x\bar{x})(x{\leftrightarrow}y \parallel T_j)]]]] & \langle\text{by Lemma 3.25}\rangle \\
\longrightarrow & \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_1[\mathscr{F}_2[T_j\{y/\bar{x}\}]]]] & \langle\text{by E-Link and E-Cong}\rangle
\end{array}
$$

Condition (2) does not hold. Some T_i and T_j (for $1 \leq i, j \leq n$) are ready to act on dual endpoints $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$.

$$
\begin{array}{lll}
P & \overset{s}{\equiv} \mathscr{E}_1[(\nu x\bar{x})(\mathscr{E}_2[\mathscr{F}_i[T_i] \parallel \mathscr{F}_j[T_j]])] & \langle\text{by Corollary 3.23}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[(\nu x\bar{x})(\mathscr{F}_i[T_i] \parallel \mathscr{F}_j[T_j])]] & \langle\text{by Lemma 3.24}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_i[(\nu x\bar{x})(T_i \parallel \mathscr{F}_j[T_j])]]] & \langle\text{by Lemma 3.25}\rangle \\
& \overset{s}{\equiv} \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_i[\mathscr{F}_j[(\nu x\bar{x})(T_i \parallel T_j)]]]] & \langle\text{by Lemma 3.25}\rangle \\
\longrightarrow & \mathscr{E}_1[\mathscr{E}_2[\mathscr{F}_i[\mathscr{F}_j[R]]]] & \langle\text{by Lemma 3.34 and E-Cong}\rangle
\end{array}
$$

□

3.2.4 Duality, Dependency, and Deadlock

The definition of canonical form (Definition 3.33) requires justification. It defines canonical forms as "processes that do not reduce", which is an easy way to get in trouble and admit processes that are stuck for undesirable reasons such as deadlock as "canonical".

The section follows the same structure as the corresponding section for CP (§ 2.2.4):

- *Dependency Graph and Deadlock Freedom.*
  I define the dependency graph for HCP processes, and prove that HCP processes are deadlock-free because their dependency graph is always essentially acyclic.
- *Adequacy.*
  I define when a process is blocked on a set of endpoints, and prove

that an HCP process cannot β-reduce if and only if it is blocked on a set of free endpoints.

The dependency graph is a mixed graph. I informally revisit the relevant definitions. For a detailed discussion, see § A.1.

- A *mixed graph* G has a set of vertices (denoted V_G, ranged over by u, v), a set of edges (denoted E_G), a set of arcs (denoted A_G). Edges are unordered pairs denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$. The set of edges may not contain loops uu. Arcs are ordered pairs denoted by juxtaposition overset with an arrow to indicate the direction, i.e. $\overrightarrow{uv} \triangleq (u, v)$. The set of arcs may not contain loops $\overrightarrow{uu}$.
- For any graph G with vertices $u, v \in V_G$, u is *adjacent* to v when there exists some edge $uv \in E_G$ or some arc $\overrightarrow{uv} \in A_G$.
- A *walk* w is a sequence of pairwise adjacent vertices.
- A *path* p is a walk with no repeated vertices, except possibly the first and last.
- A *cycle* c is a path that begins and ends at the same vertex.
- A walk is *essentially directed* when it contains at least one arc.
- A graph is *essentially acyclic* when if it contains no essentially directed cycles.
- A graph is *strongly connected* when if there exists a path between any two vertices.
- A graph is *connected* when if the graph formed by replacing all arcs with edges is strongly connected.
- The *subgraph of G induced by U* (denoted by $G[U]$) is the graph formed by taking the subset of vertices U and restricting the edges and arcs according to their correctness criteria, i.e. $E_{G[U]} \triangleq E_G \cap \{uv \mid u, v \in U \wedge u \neq v\}$.
- A *component* of a graph is a maximal connected subgraph.
- The *undirected reachability* relation (denoted by $\sim_G$) is the equivalence closure over E_G.
- The *essentially directed reachability* relation (denoted by $\prec_G$) is the transitive closure over A_G quotiented by $\sim_G$.

The vertices of the dependency graph are endpoint names, which are a proxy for the first action on that endpoint. The edges represent channels, created by either links or name restrictions. The arcs represent dependencies, created by prefixing. For instance, in a(). b[]. 0, the process b[]. 0 is prefixed with the action a(). Hence, the action on b depends on the action on a. (The definition of the dependency graph is unchanged from CP, except to account for the terminated process.)

Definition 3.36 (Dependency Graph)**.** *The* shallow dependency graph *of a process* P*, written* Dep(P)*, is a mixed graph (see § A.1). The process* P *is of the form* $\mathscr{C}^n[\mathsf{T}_1, \ldots, \mathsf{T}_n]$ *(for some* $n \geq 0$*). The shallow dependency*

graph $\mathrm{Dep}(P)$ *is defined as:*

$$V_{\mathrm{Dep}(P)} \triangleq \bigcup_{1\le i\le n} \mathrm{fn}(T_i)$$
$$E_{\mathrm{Dep}(P)} \triangleq \bigcup_{1\le i\le n} \{xy \mid T_i = x\leftrightarrow y\} \cup \{x\bar{x} \mid \{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])\}$$
$$A_{\mathrm{Dep}(P)} \triangleq \bigcup_{1\le i\le n} \{\overrightarrow{xy} \mid x, y \in \mathrm{fn}(T_i), \mathrm{ready}(T_i, x) \wedge \neg\,\mathrm{ready}(T_i, y)\}$$

By Lemma 3.22, $E_{\mathrm{Dep}(P)}$ *and* $A_{\mathrm{Dep}(P)}$ *contain no loops. If* G *is (the subgraph of) some dependency graph, I write* $\mathrm{fn}(G)$ *for its vertices, i.e.* $\mathrm{fn}(G) = V_G$.

The dependency graph gives us *duality* on actions, which is undirected reachability in the dependency graph. (The definition of duality is unchanged from CP.)

Definition 3.37 (Duality)**.** *An endpoint* x *is* dual *to some endpoint* y *in* P*, written* $x \sim_P y$*, if and only if there exists an undirected path from* x *to* y *in* $\mathrm{Dep}(P)$.

If $x \sim_P y$, the corresponding path in $\mathrm{Dep}(P)$ may be arbitrarily long, as undirected edges arise from both cuts and links. Consider the process

$$(\nu x\bar{x})(a\leftrightarrow x \parallel \bar{x}\leftrightarrow b)$$

The duality $a \sim b$ is witnessed by the path $(ax, x\bar{x}, \bar{x}b)$. However, while the paths arising from cuts and links may be arbitrarily long, they must alternate between cut-edges and link-edges and can never branch.

The dependency graph also gives us *dependency* on actions, which is the converse of essentially directed reachability in the dependency graph. (The definition of dependency is unchanged from CP.)

Definition 3.38 (Dependency)**.** *An endpoint* x depends *on some endpoint* y*, written* $x >_P y$*, if and only if there exists an essentially directed path from* y *to* x *in* $\mathrm{Dep}(P)$.

One quirk of using endpoints as a proxy for actions is that the duality and dependency appear to "leak" restricted names, i.e. $\sim_P$ and $>_P$ are not relations over $\mathrm{fn}(P)$, but relations over $\mathrm{fn}(P) \cup \mathrm{bn}(\mathscr{C}[\cdot])$, where $\mathscr{C}[\cdot]$ is the maximum configuration context of P. However, as stated, these relations should be viewed as relations on the first actions on those endpoints, not the endpoints themselves.

A process is in deadlock if the dependency relation is not antisymmetric, or, equivalently, if there is a cycle in the dependency graph that contains at least one arc.

Definition 3.39 (Deadlock)**.** *A process* P *is in deadlock, written* $\mathrm{deadlock}(P)$*, if* $\mathrm{Dep}(P)$ *contains an essential cycle.*

For HCP, the statement that well-typed processes are deadlock free is too weak as an induction hypothesis. Instead, we prove the stronger proposition that (1) the dependency graph is essentially acyclic, and (2) that the components of the dependency graph correspond one-to-one to

the typing environments in the hyper-environment. When specialised to CP, this property becomes deadlock freedom, since every CP process is typed under exactly one typing environment, and its dependency graph is always connected.

Let us start with the base case. If a process is ready, then its dependency graph is essentially acyclic and connected. The latter suffices, since threads are always typed under a single typing environment.

Lemma 3.40. *If* P *is ready, then* $\mathrm{Dep}(P)$ *is essentially acyclic and connected.*

Proof. By case analysis on P.

(For the full proof, see § 3.5.)

□

While the statement of deadlock freedom is stronger, the actual proof does not differ significantly from CP. There is only the small additional burden of maintaining the isomorphism between components and typing environments. The interesting case is the case for name restriction, which, as for CP, relies on Lemma A.2, the property that connecting two essentially acyclic graphs with a single edge yields another essentially acyclic graph.

Proposition 3.41. *If* $P \vdash \mathscr{G}$*, then the dependency graph* $\mathrm{Dep}(P)$ *is essentially acyclic, and there is an isomorphism* f *between the typing environments in* $\mathscr{G}$ *and the components of* $\mathrm{Dep}(P)$ *that preserves* fn*, i.e.* $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$*.*

Proof. By induction on the derivation of $P \vdash \mathscr{G}$. The case where P is ready follows by Lemma 3.40. The case where P is of the form 0 follows vacuously. The case where P is of the form $P_1 \parallel P_2$ follows by taking the union of the induction hypotheses. The case where P is of the form $(\nu x \bar{x})P'$, is the interesting case. By typing, x and $\bar{x}$ are in distinct typing environments By f, these typing environments correspond to distinct components of the dependency graph. By Lemma A.2, connecting distinct components with a single edge preserves acyclicity.

(For the full proof, see § 3.5.)

□

Every well-typed HCP process is deadlock-free. This follows immediately from Proposition 3.41 by forgetting the isomorphism.

Corollary 3.42. *If* $P \vdash \mathscr{G}$*, then* $\neg\, \mathrm{deadlock}(P)$*.*

A *blocking action* is an action that blocks a process from making progress. For instance, in the process

$$(\nu x\bar{x})(a().\, x[].\, 0 \parallel \bar{x}().\, P)$$

the action $a()$ is *blocking*. However, not every ready action is *blocking*. The action $\bar{x}()$ is ready, but not blocking. Rather, it is *blocked*: its dual $x[]$ depends on $a()$, so it cannot reduce until $a()$ does.

As I did with dependency, I approximate blocking actions with blocking endpoints. *Blocking endpoints* are the maxima of the dependency relation, or, equivalently, the sources of the dependency graph. The blocking set of a process is the set of all sources of its dependency graph. Every ready action in a process is blocked on one of the endpoints in the blocking set. (The definition of the blocking set is unchanged from CP.)

Definition 3.43 (Blocking Set)**.** *The blocking set of endpoints of a process* P*, written* $\mathrm{blocking}(P)$*, is the set of sources of* $\mathrm{Dep}(P)$*, i.e.* $\{x \in V_{\mathrm{Dep}(P)} \mid \nexists y.\, x >_P y\}$.

The blocking set is closed under duality.

Lemma 3.44. *If* $P \vdash \Gamma$ *and* $x \sim_P y$*, then* $x \in \mathrm{blocking}(P) \implies y \in \mathrm{blocking}(P)$.

Each endpoint in the blocking set corresponds to a ready action.

Lemma 3.45. *If* $P \vdash \Gamma$ *and* $x \in \mathrm{blocking}(P)$*, then* $P = \mathscr{E}[T]$ *and* $\mathrm{ready}(T, x)$.

Due to the dualities generated by links, the blocking set may contain more endpoints than necessary. For instance, the blocking set of the process

$$(\nu x\bar{x})(x{\leftrightarrow}a \parallel \bar{x}().\, P)$$

is $\{x, \bar{x}, a\}$. An action in P is blocked on all of these endpoints. However, I want to be able to say that any action in is blocked on a free name, and, in this case, the set $\{a\}$ suffices. A process is blocked on a set of endpoints if any action is blocked on at least one endpoint in that set. (The definition of blocking is unchanged from CP.)

Definition 3.46 (Blocked)**.** *A process* P *is blocked on a set of endpoints* X*, written* $\mathrm{blocked}(P, X)$*, if closing* X *under duality yields the blocking set* $\mathrm{blocking}(P)$*, i.e. if* $x \in X$ *and* $x \sim_P y$*, then* $y \in \mathrm{blocked}(P)$.

Any process is blocked on its blocking set.

Lemma 3.47. *If* $P \vdash \Gamma$*, then* $\mathrm{blocked}(P, \mathrm{blocking}(P))$.

If a process is blocked on some set of endpoints, it is blocked on the set formed by replacing any endpoint in that set with its dual.

Lemma 3.48. *If* $P \vdash \Gamma$ *and* $x \sim_P y$*, then* $\mathrm{blocked}(P, X) \implies \mathrm{blocked}(P, X\{y/x\})$.

If a process cannot β-reduce, then it is blocked on some set of free names.

Proposition 3.49. *If* $P \vdash \Gamma$, *then* $P \not\longrightarrow_\beta \iff \exists A \subseteq \mathrm{fn}(P).\, \mathrm{blocked}(P, A)$.

Proof. There are two cases:

- Case ($\Rightarrow$).

  By contradiction. Assume $x \in \mathrm{blocking}(P)$ and $\nexists a \in \mathrm{fn}(P).x \sim_P a$.

  There are two cases:

  - If $x \in \mathrm{fn}(P)$, then $x \sim_P x$.
  - The process $P = \mathscr{C}^n[T_1, \ldots, T_n]$ (for some $n \geq 0$).

    If $x \in \mathrm{bn}(\mathscr{C}[\cdot])$, then there exists some $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$.

    There are two cases:

    * If $\bar{x} \in \mathrm{blocking}(P)$, there exist processes T_i and T_j that are ready to act on dual endpoints. By Lemma 3.34, P is not β-free.
    * If $\bar{x} \notin \mathrm{blocking}(P)$, there exists some y such that $\bar{x} >_P y$. By definition, $x\bar{x} \in E_{\mathrm{Dep}(P)}$. Hence, $x >_P y$ and $x \notin \mathrm{blocking}(P)$.

- Case ($\Leftarrow$).

  By contradiction.

  Assume $P \longrightarrow_\beta$. By inversion, there exist some T_i and T_j (for $1 \leq i, j \leq n$) that are ready to act on dual endpoints x and $\bar{x}$.

  By definition, x and $\bar{x}$ only have outgoing arcs in $\mathrm{Dep}(P)$, and the only edge connected to either is $x\bar{x}$. Hence, $\{x, \bar{x}\} \subseteq \mathrm{blocking}(P)$ and $\nexists a \in \mathrm{fn}(P).x \sim_P a \vee \bar{x} \sim_P a$.

□

Corollary 3.50. *If* $P \vdash \Gamma$, *then* $\mathrm{canonical}(P) \implies \mathrm{blocking}(P) \subseteq \mathrm{fn}(P)$.

Proof. As $\mathrm{canonical}(P)$, $P \not\longrightarrow_\alpha$ and $P \not\longrightarrow_\beta$. By Proposition 3.49, there exists some $A \subseteq \mathrm{fn}(P)$ such that $\mathrm{blocked}(P, A)$. By definition, $A \subseteq \mathrm{blocking}(P)$. It remains to show that $\mathrm{blocking}(P) \subseteq A$.

By contradiction.

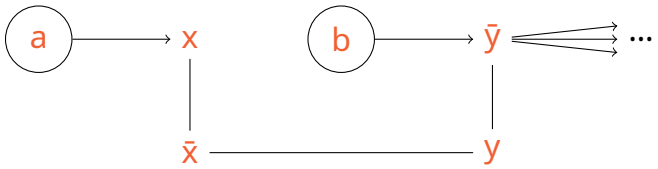
Assume $x \in \mathrm{blocking}(P)$ and $x \notin A$. By definition, there exists some $a \in A$ such that $x \sim_P a$. The duality $x \sim_P a$ corresponds to some undirected path $p_{xa} = (x, \ldots, a)$ in $\mathrm{Dep}(P)$, which must contain at least one edge that connects some *bound* name y to some *free* name b, say, yb. By definition, any edge in $\mathrm{Dep}(P)$ generated by a cut connects two *bound* names. Therefore, yb must be generated by a link. By Lemma 3.45, $P = \mathscr{E}[y \leftrightarrow b]$. Hence, $P \longrightarrow_\alpha$.

□

Unfortunately, as with CP, "blocked on free endpoints" does not characterise canonical forms, as blocking(P) $\subseteq$ fn(P) $\not\Rightarrow$ $P \not\rightarrow_a$. For instance, the process

$$(\nu x\bar{x})(a().\, x[].\, 0 \parallel (\nu y\bar{y})(\bar{x}{\leftrightarrow}y \parallel b().\, \bar{y}().\, P))$$

can α-reduce. Its dependency graph, with blocking endpoints circled, is



In conclusion, my definition of canonical form (Definition 3.33) is *adequate*: any process in canonical form is blocked on some set of free endpoints. Due to the behaviour of links, "blocked on free endpoints" is insufficient to characterise canonical forms. As this would be a desirable property to have, I consider alternative semantics for the link construct in § 3.3. I have not adopted any of these alternatives as standard to maintain backwards compatibility with the work based on Wadler's CP.

3.2.5 Connection and Disentanglement

In this section, I formalise the notion of the *connection graph* of a process, and prove *disentanglement*, the property that any HCP process can be rewritten into the parallel composition of CP-like processes, where I use "CP-like" to mean processes that would be syntactically well-formed and well-typed CP processes, i.e. each name restriction is followed by its corresponding parallel composition, each send action is followed by its corresponding parallel composition in the correct order, and each close action is followed by the terminated process.

The first part of this section follows the same structure as the corresponding section for CP (§ 2.2.5):

- *Connection Forest*.
  I define the connection graph for HCP processes, and prove that the connection graph is always a forest.
- *Right-Branching Forest Form*.
  I define the equivalent of right-branching form for HCP, which is right-branching *forest* form, and prove that any process can be rewritten to right-branching forest form.

The second part of this section proves disentanglement for HCP. In the introduction, I defined disentanglement as a function that converts HCLL proofs into a sequence of CLL proofs:

$$\begin{array}{c} p \\ \vdots \\ \vdash \Gamma_1 \parallel \ldots \parallel \Gamma_n \end{array} \quad \Longrightarrow \quad \begin{array}{c} p_1 \\ \vdots \\ \vdash \Gamma_1 \end{array} \quad , \ldots , \quad \begin{array}{c} p_n \\ \vdots \\ \vdash \Gamma_n \end{array}$$

For the process calculus HCP, I prefer a slightly stronger property. Disentanglement should convert HCP processes to the parallel composition of CP-like processes *and* should be justified by the structural congruence. For any well-typed process $P \vdash \mathcal{G}^k$:

- If $k = 0$, then $P \equiv 0$ and $\mathcal{G}^0 = \circ$.
- If $k \geq 1$, then $P \equiv P_1 \parallel \ldots \parallel P_k$ and $\mathcal{G}^k = \Gamma_1 \parallel \ldots \parallel \Gamma_k$ such that $P_i \vdash \Gamma_i$ (for $1 \leq i \leq k$) and each P_i is CP-like.

The conversion to right-branching forest form satisfies part of this definition. It converts an HCP process to a sequence of processes that are typed under a single typing environment, *and* it is justified by the structural congruence. However, the conversion is *shallow*. It only rewrites the maximum configuration context. Hence, the resulting processes are *shallowly* CP-like. Only the name restrictions and parallel compositions in their maximum configuration contexts are arranged as CP cuts.

I define disentanglement by iterating the conversion to right-branching forest form, which arranges *all* name restrictions and parallel compositions as CP cuts, and subsequently arranging the continuations of send and close actions to match those of CP send and close actions. (I defer the proof that disentangled processes are CP-like to § 3.2.7, where I discuss the translation between HCP processes and multisets of CP processes.)

Let us revisit connection graphs by examining three example processes:

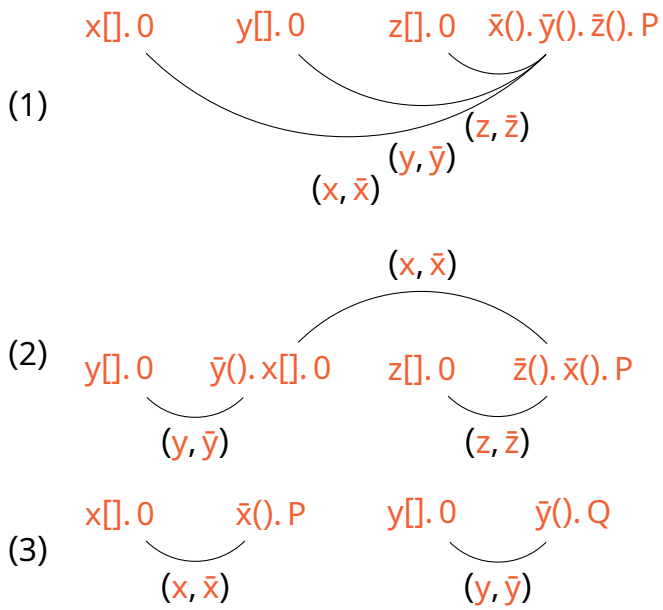
(1) $(\nu x\bar{x})(\nu y\bar{y})(\nu z\bar{z})(\ x[].0 \parallel y[].0 \parallel z[].0 \parallel \bar{x}().\bar{y}().\bar{z}().P\ )$

(2) $(\nu x\bar{x})(\nu y\bar{y})(\nu z\bar{z})(\ y[].0 \parallel \bar{y}().x[].0 \parallel z[].0 \parallel \bar{z}().\bar{x}().P\ )$

(3) $(\nu x\bar{x})(\nu y\bar{y})(\ x[].0 \parallel \bar{x}().P \parallel y[].0 \parallel \bar{y}().Q\ )$

The connection graph of a process is the graph formed of all ready sub-processes and the channels that connect them. Whereas CP's connection graphs are trees, HCP's connection graphs are forests. The connection graphs for processes (1) and (2) are equivalent to those for the example processes discussed in § 2.2.5. Both are fully connected, consisting of four threads, connected by three channels. Process (3) is *not* fully connected.

It consists of four processes, connected in pairs by two channels.



We can use the connection graph to rewrite any process into right-branching forest form, which is the parallel composition of a sequence of processes in right-branching tree form, which is more or less CP's right-branching form:

$$(\nu x_1^1 \bar{x}_1^1)(P_1^1 \parallel \cdots (\nu x_n^1 \bar{x}_n^1)(P_n^1 \parallel P_{n+1}^1) \cdots) \parallel \cdots \parallel (\nu x_1^k \bar{x}_1^k)(P_1^k \parallel \cdots (\nu x_n^k \bar{x}_n^k)(P_n^k \parallel P_{n+1}^k) \cdots)$$

As mentioned, CP's right-branching form is often used to write nice and concise proofs. Unfortunately, HCP's right-branching forest form is a bit too verbose to fill the same niche. Worse, the presentation above does not communicate the case where $k = 0$ and the process is 0.

The procedure to convert processes to right-branching forest form picks a tree from the connection graph, moves *all* the corresponding name restrictions and threads to the top-level, arranges them in right-branching tree form, removes the tree, and continues until the graph is empty.

The procedure to convert processes to right-branching tree form is the same as the procedure for CP. It picks a leaf from the tree, moves the corresponding name restriction and thread to the topmost, leftmost position, removes the leaf, and continues until all of the tree is empty.

A process may have multiple different right-branching forest forms. The following are one possible right-branching forest form for each of the example processes above:

(1) $(\nu x \bar{x})(\, x[].\, 0 \parallel (\nu y \bar{y})(\, y[].\, 0 \parallel (\nu z \bar{z})(\, z[].\, 0 \parallel \bar{x}().\, \bar{y}().\, \bar{z}().\, P\,)\,)\,)$

(2) $(\nu y \bar{y})(\, y[].\, 0 \parallel (\nu x \bar{x})(\, \bar{y}().\, x[].\, 0 \parallel (\nu z \bar{z})(\, z[].\, 0 \parallel \bar{z}().\, \bar{x}().\, P\,)\,)\,)$

(3) $(\nu x \bar{x})(\, x[].\, 0 \parallel \bar{x}().\, P\,) \parallel (\nu y \bar{y})(\, y[].\, 0 \parallel \bar{y}().\, Q\,)$

As in § 2.2.5, the definition of connection graph is *shallow*, rather than deep, as we only account for the connections up to the maximum configuration context. However, in the second part of this section, I will demonstrate that we can use the shallow connection graph to reason about the deep connection structure of a process.

The connection graph is a undirected edge-labelled graph. I informally revisit the relevant definitions. For a detailed discussion, see § A.1.

- A *undirected edge-labelled graph* G has a set of vertices (denoted V_G, ranged over by u, v), a set of edges (denoted E_G), a set of edge labels (denoted $\mathscr{L}_G$), and an edge-labeling function (denoted ℓ_G) that assigns labels to edges. Edges are unordered pairs denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$. The set of edges may not contain loops uu.
  (It suffices to define V_G and ℓ_G, since $E_G \triangleq \mathrm{dom}(\ell_G)$ and $\mathscr{L}_G \triangleq \mathrm{cod}(\ell_G)$.)
- Two vertices $u, v \in V_G$ are *adjacent* when there exists an edge $uv \in E_G$.
- A *walk* w is a sequence of pairwise adjacent vertices.
- A *path* p is a walk with no repeated vertices, except possibly the first and last.
- A *cycle* c is a path that begins and ends at the same vertex.
- The *subgraph of* G *induced by* U (denoted by $G[U]$) is the graph formed by taking the subset of vertices U and restricting the edges, edge labels, and edge-labelling function according to their correctness criteria, i.e. $E_{G[U]} \triangleq E_G \cap \{uv \mid u, v \in U \wedge u \neq v\}$.
- A graph is *acyclic* when it does not contain a cycle.
- A graph is *connected* when there is a path between any two vertices.
- A *component* C of a graph is a maximal connected subgraph.
- A *tree* T is a graph that is connected and acyclic.
- A *forest* F is a graph whose components are trees.

(The definition of the connection graph is unchanged from CP, except to account for the terminated process.)

Definition 3.51 (Connection Graph)**.** *The* shallow connection graph *of a well-typed process* P*, written* $\mathrm{Con}(\mathsf{P})$*, is an undirected edge-labelled graph (see § A.1) where the vertices are threads, the edges are the channels that connect those threads, and the edges are labelled by unordered pairs of their endpoints. The process* P *is of the form* $\mathscr{C}^n[\mathsf{T}_1, \ldots, \mathsf{T}_n]$ *(for some* $n \geq 0$*). The shallow connection graph* $\mathrm{Con}(\mathsf{P})$ *is defined as:*

$$V_{\mathrm{Con}(\mathsf{P})} \triangleq \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$$
$$\ell_{\mathrm{Con}(\mathsf{P})} \triangleq \{\mathsf{T}_i\mathsf{T}_j \mapsto (\mathsf{x}, \bar{\mathsf{x}}) \mid \mathsf{T}_i, \mathsf{T}_j \in V_{\mathrm{Con}(\mathsf{P})}, \{\mathsf{x}, \bar{\mathsf{x}}\} \in \mathrm{dn}(\mathscr{C}[\cdot]), \mathsf{x} \in \mathsf{T}_i \wedge \bar{\mathsf{x}} \in \mathsf{T}_j\}$$

By Lemma 2.21, $E_{\mathrm{Con}(\mathsf{P})}$ *contains no loops. By Proposition 2.18,* $\ell_{\mathrm{Con}(\mathsf{P})}$ *is a function. If* G *is a subgraph of some connection graph, I write* $\mathrm{fn}(G)$ *for the free names in the vertices of* G*, i.e.* $\mathrm{fn}(G) \triangleq \bigcup_{\mathsf{P} \in V_G} \mathrm{fn}(\mathsf{P})$.

In HCP, the connection graph of a process is always a forest. However, that statement by itself is too weak to prove by induction. Instead, we

prove the stronger proposition that (1) the connection graph is a forest, and (2) that the components in the connection graph correspond one-to-one to the typing environments in the hyper-environment. When specialised to CP, this property becomes the property that every connection graph is a tree, since every CP process is typed under exactly one typing environment, and its connection graph is always connected.

The proof does not differ significantly from CP. There is only the small additional burden of maintaining the isomorphism between component and typing environments. The interesting case is the case for name restriction, which, as for CP, relies on the property that connecting two trees with a single edge yields another tree.

Proposition 3.52. *If* $P \vdash \mathscr{G}$*, then* $\mathrm{Con}(P)$ *is a forest, and there is an isomorphism* f *between the typing environments in* $\mathscr{G}$ *and the trees of* $\mathrm{Con}(P)$ *that preserves* fn*, i.e.* $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$*.*

Proof. By induction on the maximum configuration context of P and inversion on P and its typing derivation. The interesting case is for name restriction. The endpoints connected by the name restriction occur in different typing environment. Hence, by the isomorphism f, those endpoints occur in disjoint trees in the connection graph and, by Lemma A.1, the connection graph for the result, formed by connecting those trees with the single edge arising from the name restriction, remains a forest.

(For the full proof, see § 3.5.)

□

A process is in right-branching forest form when it is either 0 or it is the parallel composition of processes in right-branching tree form.

Definition 3.53 (Right-branching Tree Form)**.** *A process* P *is in* right-branching tree form *if* P *is ready or if* P *is of the form (for some* $n \geq 1$*)*

$$(\nu x_1 \bar{x}_1)(T_1 \parallel \cdots (\nu x_n \bar{x}_n)(T_n \parallel T_{n+1}) \cdots)$$

A process is in right-branching tree form is if is a right-branching list of CP cuts connecting threads.

(The definition of right-branching tree form is unchanged, except in name, from CP's definition of right-branching form.)

Definition 3.54 (Right-branching Forest Form)**.** *A process* P *is in* right-branching forest form *if* P *is of the form* 0*, or if* P *is of the form* $P_1 \parallel \cdots \parallel P_n$ *(for some* $n \geq 1$*) such that (for* $1 \leq i \leq n$*) each* P_i *is in right-branching tree form.*

Any well-typed process can be rewritten to right-branching forest form. The proof is given by induction on the structure of the hypersequent, and

is decomposed into two lemmas, which correspond to the two cases of the induction:

- If $P \vdash \circ$, then P can be rewritten to the terminated process.
- If $P \vdash \mathcal{G} \parallel \Gamma$, then P can be rewritten to pull one process in right-branching tree form out to the top level, using a procedure similar to the one used for converting CP processes to right-branching form.

Lemma 3.55. *If* $P \vdash \circ$, *then* $P \overset{\text{ls}}{\equiv} 0$.

Proof. Let G be Con(P). By Proposition 3.52, there is an isomorphism f between the typing environments in $\circ$ and the trees of G that preserves fn, i.e. $\text{fn}(\Gamma) = \text{fn}(f(\Gamma))$. As there are no typing environments in $\circ$, f is the empty function, and G is the null graph. Let $\mathscr{C}^n[\cdot]$ be maximal for P such that $P = \mathscr{C}^n[P_1, \ldots, P_n]$ (for some $n \geq 0$). By definition, $V_G = \{P_1, \ldots P_n\}$. As G is the null graph, $V_G = \varnothing$. Hence, $n = 0$, i.e. the maximum configuration context $\mathscr{C}^0[\cdot]$ contains zero holes. By induction on $\mathscr{C}^0[\cdot]$, the process P is equal to the parallel composition of terminated processes, and is equivalent to 0 by repeated application of SC-ParNil.

□

Lemma 3.56. *If* $P \vdash \mathcal{G} \parallel \Gamma$, *then there exist processes* Q *and* R *such that* $Q \vdash \mathcal{G}$, $R \vdash \Gamma$, $P \overset{\text{ls}}{\equiv} Q \parallel R$, *and* R *is in right-branching tree form.*

Proof. The proof proceeds by picking the tree corresponding to Γ from the connection forest, using the isomorphism constructed by Proposition 3.52, and then iteratively picking its leaves and rewriting the corresponding process to right-branching form, as in Proposition 2.51.

(For the full proof, see § 3.5.)

□

Proposition 3.57. *If* $P \vdash \mathcal{G}$, *then there exists a process* Q, *such that* $P \overset{\text{ls}}{\equiv} Q$, *and* Q *is in right-branching forest form, and there is an isomorphism* f *between the processes in right-branching tree form in* Q *and the typing environments in* $\mathcal{G}$ *that preserves* fn, *i.e.* $\text{fn}(\Gamma) = \text{fn}(f(\Gamma))$.

Proof. The proof proceeds by iteratively picking typing environments in the hyper-environment and moving the corresponding process to the top-level in right-branching tree form using Lemma 3.56.

(For the full proof, see § 3.5.)

□

Connection graphs are an alternative representation for processes, with the interesting property that they represent the maximum configuration context of a process without any spurious ambiguity. Connection

graphs correspond to shallow proof nets for HCLL. (For a more detailed discussion of the correspondence between connection graphs and proof nets, see § 2.2.5.)

Proposition 3.58. *If* P *is well-typed, then* $P \overset{\text{LS}}{\equiv} Q \iff \text{Con}(P) = \text{Con}(Q)$.

Proof. There are two cases:

- Case ($\Rightarrow$).

  By induction on the proof of the structural congruence $P \overset{\text{LS}}{\equiv} Q$. The cases for reflexivity, transitivity, symmetry, and SC-Cong follow by induction and those same properties of equality. The remaining cases follow immediately.

- Case ($\Leftarrow$).

  Let Proc(T) be the set of processes in right-branching forest form obtained from the connection graph G of a well-typed process by Proposition 3.57. (This is a set because Proposition 3.57 defines a non-deterministic procedure.)

  Pick any $R \in \text{Proc}(\text{Con}(P))$. As $\text{Con}(P) = \text{Con}(Q)$, $\text{Proc}(\text{Con}(P)) = \text{Proc}(\text{Con}(Q))$. Hence, $R \in \text{Proc}(\text{Con}(Q))$. By definition, $P \overset{\text{LS}}{\equiv} R$ and $Q \overset{\text{LS}}{\equiv} R$. Hence, $P \overset{\text{LS}}{\equiv} Q$.

□

An HCP process is disentangled when it is the terminated process, or the parallel composition of a sequence of CP-like processes. The formal definition does not make explicit reference to CP syntax, and I defer the proof disentangled HCP processes correspond to CP processes to § 3.2.7.

Definition 3.59 (Disentangled)**.** *A process* P *is* disentangled *when every terminated process matches the form of a CP close, and every parallel composition matches the form of a CP cut or CP send, or is at the top-level.*

Formally, P *is* disentangled *when the following conditions hold:*

1. *If* P *is of the form* $Q[P_1 \parallel P_2]$*, then one of the following holds:*
   a. $Q[\cdot]$ *is of the form* $R[(\nu x \vec{x})\square]$ *such that* $x \in \text{fn}(P_1)$ *and* $\vec{x} \in \text{fn}(P_2)$.
   b. $Q[\cdot]$ *is of the form* $R[x[y].\square]$ *such that* $y \in \text{fn}(P_1)$ *and* $x \in \text{fn}(P_2)$.
   c. $Q[\cdot]$ *is of the form* $\mathscr{E}$ *such that* $\text{bn}(\mathscr{E}) = \varnothing$.
2. *If* P *is of the form* $Q[0]$*, then one of the following holds:*
   a. $Q[\cdot]$ *is of the form* $\square$.
   b. $Q[\cdot]$ *is of the form* $R[x[].\square]$.

The conditions of disentangled form in Definition 3.59 are sufficient to guarantee that the process is CP-like.

Lemma 3.60. *If* $P \vdash \mathcal{G}$ *and* P *is disentangled, then:*

1. $\mathcal{G}$ *is of the form* $\circ$ $\implies$ P *is of the form* 0
2. $\mathcal{G}$ *is of the form* $\mathcal{G} \parallel \Gamma$ $\implies$ P *is of the form* $Q \parallel R$
3. P *is of the form* $Q[(\nu x \tilde{x})R]$ $\implies$ R *is of the form* $R_1 \,{}_x\|_{\tilde{x}}\, R_2$
4. P *is of the form* $Q[x[y].\, R]$ $\implies$ R *is of the form* $R_1 \,{}_y\|_x\, R_2$
5. P *is of the form* $Q[x[].\, R]$ $\implies$ R *is of the form* 0

Proof. There are five cases:

1. By induction on the structure of P. If P is of the form 0, the result follows immediately. If P is of the form $Q \parallel R$, the induction hypotheses give us $Q = 0$ and $R = 0$, which contradicts condition (2) of disentangled form. The remaining cases, where P is ready or P is of the form $(\nu x \tilde{x})P'$ are impossible, by inversion on the typing derivation.
2. By induction on the structure of P. If P is of the form $Q \parallel R$, the result follows immediately. If P is of the form $(\nu x \tilde{x})P'$, the induction hypothesis gives us that P' is of the form $Q' \parallel R'$, which contradicts condition (1) of disentangled form. The remaining cases, where P is ready or P is of the form 0 are impossible, by inversion on the typing derivation.
3. By case (2) and condition (1a) of disentangled form.
4. By case (2) and condition (1b) of disentangled form.
5. By case (1).

□

Every HCP process can be disentangled.

Right-branching forest form is useful for disentangling processes, but not quite sufficient. If we convert every sub-process to right-branching forest form, the resulting process is *nearly* disentangled. Every terminated process matches the form of a CP close, and every parallel composition is at the top-level, matches a CP cut, or *nearly* matches a CP send. The continuation of a send action must be the parallel composition that splits the corresponding endpoints, but the processes need not be in the correct order, i.e. we may have $x[y].\, Q \parallel P$, where Q handles x and P handles y.

To disentangle a process, we convert every sub-process to right-branching forest form, and take special care to correct the order in the continuation of send actions.

Proposition 3.61. *If* $P \vdash \mathcal{G}$*, then there exists some* Q *such that* $P \equiv_{\text{ls}} Q$ *and* Q *is disentangled.*

Proof. By induction on the derivation of $P \vdash \mathcal{G}$ and inversion on P.

If P is ready, the result follows by induction. Most cases can be handled uniformly, but send, offer and the absurd offer are handled separately.

- Case P is of the form $x[y].\, P'$.

  By induction, $P' \overset{\text{LS}}{\equiv} Q'$ for some disentangled process Q'. By Lemma 3.60 (2), Q' is of the form $Q'_1 \parallel Q'_2$. By condition (2) of disentangled form, $Q'_1 \neq 0$ and $Q'_2 \neq 0$. There are two cases:

  - Subcase $y \in \mathrm{fn}(Q'_1)$ and $x \in \mathrm{fn}(Q'_2)$.

    Let Q be $x[y].\, Q'_1 \parallel Q'_2$. The result follows.

  - Subcase $x \in \mathrm{fn}(Q'_1)$ and $y \in \mathrm{fn}(Q'_2)$.

    Let Q be $x[y].\, Q'_2 \parallel Q'_1$. The result follows by SC-ParComm.

- Case P is of the form $x(y).\, P'$, $x[].\, P'$, $x().\, P'$, $x \triangleleft \mathrm{inl}.\, P'$, or $x \triangleleft \mathrm{inr}.\, P'$.

  By induction, $P' \overset{\text{LS}}{\equiv} Q'$ for some disentangled process Q'. Let Q be $x(y).\, Q'$, $x[].\, Q'$, $x().\, Q'$, $x \triangleleft \mathrm{inl}.\, Q'$, or $x \triangleleft \mathrm{inr}.\, Q'$, respectively. The result follows.

- Case P is of the form $x \triangleright \{\mathrm{inl}: P'_1; \mathrm{inr}: P'_2\}$.

  By induction on P'_1, $P'_1 \overset{\text{LS}}{\equiv} Q'_1$ for some disentangled process Q'_1. By induction on P'_2, $P'_2 \overset{\text{LS}}{\equiv} Q'_2$ for some disentangled process Q'_2. Let Q be $x \triangleright \{\mathrm{inl}: Q'_1; \mathrm{inr}: Q'_2\}$. The result follows.

- Case P is of the form $x \leftrightarrow y$ or $x \not\downarrow N$.

  The result follows immediately.

Otherwise, the result follows by converting P to right-branching forest form, taking the maximum configuration context of the resulting process, and converting each ready subprocess by induction.

- Case P is of the form $(\nu x \tilde{x}) P'$, $P_1 \parallel P_2$, or 0.

  By Proposition 3.57, there exists some Q' such that $P \overset{\text{LS}}{\equiv} Q'$ and Q' is in right-branching forest form. By case analysis on Q':

  - If Q' is of the form 0, the result follows immediately.
  - Otherwise, let $\mathscr{C}^n[\cdot]$ be the maximum configuration context of Q' such that $Q' = \mathscr{C}^n[Q'_1, \ldots, Q'_n]$ (for some $n \geq 1$) and (for $1 \leq i \leq n$) each Q'_i is ready. By induction, there exist processes $Q_1, \ldots, Q_n$ such that (for $1 \leq i \leq n$) each $Q'_i \overset{\text{LS}}{\equiv} Q_i$ and Q_i is disentangled. Let Q be $\mathscr{C}^n[Q_1, \ldots, Q_n]$. By transitivity and congruence, $P \overset{\text{LS}}{\equiv} Q$. By induction on the structure of $\mathscr{C}^n[\cdot]$ and inversion on the fact that Q' is in right-branching forest form, Q is disentangled. The result follows.

□

Disentanglement is the procedure defined by the proof of Proposition 3.61. The procedure is non-deterministic and follows the outline in the introduction to this section. (The non-determinism arises from the

arbitrary choice of tree in Proposition 3.61 and the arbitrary choice of leaf in Lemma 3.56.)

The result is a function that converts a process to a *set* of disentangled processes, all of which are equivalent to the original process and to each other by structural congruence.

Definition 3.62 (Disentanglement)**.** *The* disentanglement *of* P*, written* $⦇P⦈$*, is the set of processes obtained from* P *by Proposition 3.61.*

All elements of $⦇P⦈$ *are equivalent to each other and to* P *under link-preserving structural congruence. I use* $⦇·⦈$ *as functional under structural congruence, and write* $⦇P⦈ \overset{L}{\equiv} Q$ *to mean* Q *is an arbitrary element of* $⦇P⦈$*. I extend* $⦇·⦈$ *to configuration and evaluation contexts by preserving holes and otherwise acting as on the corresponding process terms.*

The disentanglement $⦇P⦈$ does not contain *all* disentangled processes equivalent to P, only those which are in right-branching form.

Disentanglement distributes over maximal evaluation contexts.

Lemma 3.63. *If* $\mathscr{E}[T] \vdash \mathscr{G}$*, then* $⦇\mathscr{E}[T]⦈ = \{\mathscr{E}'[T'] \mid \mathscr{E}' \in ⦇\mathscr{E}⦈, T' \in ⦇T⦈\}$.

Disentanglement distributes over the maximum configuration context.

Lemma 3.64. *If* $\mathscr{C}[T_1, \ldots, T_n] \vdash \mathscr{G}$*, then*

$$⦇\mathscr{C}[T_1, \ldots, T_n]⦈ = \{\mathscr{C}'[T'_{\rho(1)}, \ldots, T'_{\rho(n)}] \mid \exists \rho. \mathscr{C}'[\cdot] \in ⦇\mathscr{C}[\cdot]⦈, T'_1 \in ⦇T_1⦈, \ldots, T'_n \in ⦇T_n⦈\}$$

where ρ *is a permutation on the indices* $[1, n]$*.*

The set of right-branching forms of a process is closed under link-preserving shallow structural congruence, as link-preserving shallow structural congruence preserves the connection graph. Likewise, the disentanglement is closed under any link-preserving structural congruence.

Lemma 3.65. *If* $P \vdash \mathscr{G}$ *and* $P \overset{L}{\equiv} Q$*, then* $⦇P⦈ = ⦇Q⦈$.

More importantly, disentanglement is closed under CP's structural congruence, as we will see in § 3.2.7. First, however, we must pause our discussion of disentanglement to introduce Multiset CP, the calculus of multisets of CP processes.

3.2.6 Multiset CP

In this section, I define Multiset CP. At several points in this chapter, I have mentioned that an HCP process corresponds to a multiset of CP processes. However, I have found that it is tedious to relate HCP directly to multisets of CP processes, and much clearer to relate it to another session-typed processes calculus which captures that notion. Multiset CP is that calculus. Its processes are multisets of unconnected CP processes,

presented using process notation. Its processes are typed under hyper-environments, which are multisets of CP typing environments. Its typing derivation, structural congruence, and reduction relations are all some sort of pointwise lifting of the corresponding CP relations. (I will not abbreviate Multiset CP, and continue to write it in full, to avoid confusion with Multiparty CP [MCP, Carbone et al., 2016].)

To avoid syntax highlighting whiplash between sections, the terms and types of *both* Multiset CP and CP are printed in *pink* and *green*, respectively, both are rendered in an italicised font with serif, and any relations, such as typing and reduction, are marked by subscript "$\mathscr{C}$" or "C", respectively.

The processes of Multiset CP are the parallel composition of CP processes, as defined by the following grammar:

$$\mathscr{P}, \mathscr{Q}, \mathscr{R} ::= P \mid 0 \mid \mathscr{P} \parallel \mathscr{Q}$$

The metavariables $\mathscr{P}$, $\mathscr{Q}$, and $\mathscr{R}$ refer to Multiset CP processes, and the metavariables P, Q, and R refer to CP processes, as defined in § 2.1. This matches the convention to use the calligraphic font for multisets.

Multiset CP's parallel composition and terminated process may only occur at the top-level. The syntax for CP processes is not permitted to recurse back into the syntax for Multiset CP processes. The parallel composition and terminated process that are syntactically part of CP's cut, send, and close constructs are distinct from Multiset CP's parallel composition and terminated process.

Multiset CP has no top-level name restriction. Hence, there is no possibility to connect the individual CP processes, and no possibility for them to communicate amongst each other.

The processes of Multiset CP are equivalent up to structural congruence. Concretely, the individual CP processes—i.e. the elements of the multiset—are equivalent up to CP's structural congruence, as defined in § 2.1, and Multiset CP processes—i.e. the multisets themselves—are equivalent up to reordering.

$$\begin{array}{lll} \mathscr{P} \parallel 0 & \equiv_{\mathscr{C}} \mathscr{P} & \textsc{SC-ParNil} \\ \mathscr{P} \parallel \mathscr{Q} & \equiv_{\mathscr{C}} \mathscr{Q} \parallel \mathscr{P} & \textsc{SC-ParComm} \\ \mathscr{P} \parallel (\mathscr{Q} \parallel \mathscr{R}) & \equiv_{\mathscr{C}} (\mathscr{P} \parallel \mathscr{Q}) \parallel \mathscr{R} & \textsc{SC-ParAssoc} \end{array}$$

$$\textsc{SC-Embed}\ \frac{P \equiv_{C} Q}{P \equiv_{\mathscr{C}} Q} \qquad \textsc{SC-ParCong}\ \frac{\mathscr{P} \equiv_{\mathscr{C}} \mathscr{P}' \qquad \mathscr{Q} \equiv_{\mathscr{C}} \mathscr{Q}'}{\mathscr{P} \parallel \mathscr{Q} \equiv_{\mathscr{C}} \mathscr{P}' \parallel \mathscr{Q}'}$$

In essence, the rules SC-ParNil, SC-ParComm, and SC-ParAssoc ensure the process syntax defines a multiset, and the rules SC-Embed and SC-ParCong define the universal pointwise lifting of CP's structural congruence to multisets of equal length. I likewise lift the restricted

versions of structural congruence—$\overset{\text{S}}{\equiv}_{\mathscr{C}}$, $\overset{\text{L}}{\equiv}_{\mathscr{C}}$, $\overset{\text{LS}}{\equiv}_{\mathscr{C}}$, and $\overset{\text{D}}{\equiv}_{\mathscr{C}}$—to Multiset CP (see § 2.2.1 for details).

Reduction for Multiset CP processes is the existential pointwise lifting of CP's reduction, i.e. if $\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{Q}$, then one of the processes $P \in \mathscr{P}$ reduces to one of the processes $Q \in \mathscr{Q}$, and all the other processes are preserved up to structural congruence.

$$\textsc{E-Embed}\ \frac{P \longrightarrow_{C} Q}{P \longrightarrow_{\mathscr{C}} Q} \qquad \textsc{E-Equiv}\ \frac{\mathscr{P} \equiv_{\mathscr{C}} \mathscr{P}' \qquad \mathscr{P}' \longrightarrow_{\mathscr{C}} \mathscr{Q}' \qquad \mathscr{Q}' \equiv_{\mathscr{C}} \mathscr{Q}}{\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{Q}}$$

$$\textsc{E-ParCong}_1\ \frac{\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{P}'}{\mathscr{P} \parallel \mathscr{Q} \longrightarrow_{\mathscr{C}} \mathscr{P}' \parallel \mathscr{Q}} \qquad \textsc{E-ParCong}_2\ \frac{\mathscr{Q} \longrightarrow_{\mathscr{C}} \mathscr{Q}'}{\mathscr{P} \parallel \mathscr{Q} \longrightarrow_{\mathscr{C}} \mathscr{P} \parallel \mathscr{Q}'}$$

Finally, the typing judgement for Multiset CP uses hyper-environments. The definition of hyper-environments follows the definition for HCP. Recall that hyper-environments are multisets of CP typing environments with the additional restriction that endpoint names must be unique across all typing environments.

$$\mathscr{G}, \mathscr{H} ::= \circ \mid \mathscr{G} \parallel \Gamma$$

The typing judgement itself is defined, similar to the structural congruence, as the universal pointwise lifting of CP's typing judgement to multisets of equal size.

$$\textsc{T-Embed}\ \frac{P \vdash_{C} \Gamma}{P \vdash_{\mathscr{C}} \Gamma} \qquad \textsc{T-Par}\ \frac{}{0 \vdash_{\mathscr{C}} \circ} \qquad \textsc{T-Halt}\ \frac{\mathscr{P} \vdash_{\mathscr{C}} \mathscr{G} \qquad \mathscr{Q} \vdash_{\mathscr{C}} \mathscr{H}}{\mathscr{P} \parallel \mathscr{Q} \vdash_{\mathscr{C}} \mathscr{G} \parallel \mathscr{H}}$$

The metatheory for Multiset CP can be constructed from the metatheory for CP, borrowing the occasional bit from Hypersequent CP. (I only discuss preservation and progress, but these should suffice to reassure the reader of that claim.)

Preservation follows from CP's preservation proofs (Lemma 2.24 and Proposition 2.27), and the additional cases for the new rules follow those from HCP's preservation proofs (Lemma 3.27 and Proposition 3.30).

Lemma 3.66. *If $\mathscr{P} \equiv_{\mathscr{C}} \mathscr{Q}$, then $\mathscr{P} \vdash_{\mathscr{C}} \mathscr{G}$ if and only if $\mathscr{Q} \vdash_{\mathscr{C}} \mathscr{G}$.*

Proposition 3.67 (Preservation)**.** *If $\mathscr{P} \vdash_{\mathscr{C}} \mathscr{G}$ and $\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{Q}$, then $\mathscr{Q} \vdash_{\mathscr{C}} \mathscr{G}$.*

Canonical form and progress are lifted directly from CP. A Multiset CP process is in canonical form when each CP process is in canonical form, and progress follows by pointwise application of CP's progress (Proposition 2.32).

Definition 3.68 (Canonical Form)**.** *A process $\mathscr{P}$ is in* canonical form *when each CP process $P \in \mathscr{P}$ is in CP's canonical form (Definition 2.30).*

Proposition 3.69 (Progress)**.** *If $\mathscr{P} \vdash_{\mathscr{C}} \mathscr{G}$, then either $\mathscr{P}$ is in canonical form, or there exists some $\mathscr{Q}$ such that $\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{Q}$.*

I sketched the correspondence between Multiset CP and CP in the introduction to this section. Propositions 3.70 to 3.72 formalise it. The proof of each proposition follows by induction. I believe the verbosity of these statements illustrates the advantage of working with Multiset CP over working directly with multisets of CP processes.

Multiset CP's typing relation is the universal pointwise lifting of CP's typing relation, i.e. a Multiset CP process is well-typed if each individual CP process is well-typed under one of the typing environments.

Proposition 3.70. *If $\mathscr{P} \vdash_{\mathscr{C}} \mathscr{G}$, then there is an isomorphism $f : \mathscr{P} \to \mathscr{G}$ between the elements of the multisets $\mathscr{P}$ and $\mathscr{G}$ such that $P \vdash_C f(P)$ for every $P \in \mathscr{P}$ and $f^{-1}(\Gamma) \vdash_C \Gamma$ for every $\Gamma \in \mathscr{G}$.*

Multiset CP's structural congruence is the universal pointwise lifting of the CP's structural congruence, i.e. one Multiset CP process is equivalent to another if each individual CP process in the one is equivalent to one in the other.

Proposition 3.71. *If $\mathscr{P} \equiv_{\mathscr{C}} \mathscr{Q}$, then there is an isomorphism $f : \mathscr{P} \to \mathscr{Q}$ between the elements of the multisets $\mathscr{P}$ and $\mathscr{Q}$ such that $P \equiv_C f(P)$ for every $P \in \mathscr{P}$ and $f^{-1}(Q) \equiv_C Q$ for every $Q \in \mathscr{Q}$.*

Multiset CP's reduction is the existential pointwise lifting of CP's reduction, i.e. one Multiset CP process reduces to another if one of its component CP processes reduces to one in the other, and all other component CP processes are preserved.

Proposition 3.72. *If $\mathscr{P} \longrightarrow_{\mathscr{C}} \mathscr{Q}$, then there is exactly one pair of processes $(P, Q) \in \mathscr{P} \times \mathscr{Q}$ such that $P \longrightarrow_C Q$, and an isomorphism $f : \mathscr{P}' \to \mathscr{Q}'$ between the remaining elements of the multisets, $\mathscr{P}' = \mathscr{P} - \{\!|P|\!\}$ and $\mathscr{Q}' = \mathscr{Q} - \{\!|Q|\!\}$, such that $P' \equiv_C f(P')$ for every $P' \in \mathscr{P}'$ and $f^{-1}(Q') \equiv_C Q'$ for every $Q' \in \mathscr{Q}'$.*

3.2.7 Fission, Fusion, and Disentanglement

In this section, I discuss the correspondence between Hypersequent CP and Multiset CP. The correspondence has three components—the titular components of this section. Fission is a translation from Multiset CP to Hypersequent CP, and, together, fusion and disentanglement are a translation from Hypersequent CP to Multiset CP. I discuss each component of the correspondence in order, starting with the easy one.

Fission

Fission is a translation from Multiset CP to Hypersequent CP, which splits CP's cut, send, and close into the corresponding HCP processes:

$$\begin{array}{rcl} (\nu x\bar{x})(P \parallel Q) & \text{to} & (\nu x\bar{x})\_(P \parallel Q) \\ x[y].(P \parallel Q) & \text{to} & x[y].\_(P \parallel Q) \\ x[].0 & \text{to} & x[].\_0 \end{array}$$

Visually, its definition resembles an elaborate identity function.

Definition 3.73. *Fission, $[\![P]\!]_H$, translates CP processes into HCP processes by splitting each cut into a separate name restriction and parallel composition, and likewise splitting send and close.*

$$\begin{array}{ll} [\![x \leftrightarrow y]\!]_H & \triangleq x \leftrightarrow y \\ [\![(\nu x\bar{x})(P \parallel Q)]\!]_H & \triangleq (\nu x\bar{x})([\![P]\!]_H \parallel [\![Q]\!]_H) \\ [\![x[y].(P \parallel Q)]\!]_H & \triangleq x[y].([\![P]\!]_H \parallel [\![Q]\!]_H) \\ [\![x(y).P]\!]_H & \triangleq x(y).[\![P]\!]_H \\ [\![x[].0]\!]_H & \triangleq x[].0 \\ [\![x().P]\!]_H & \triangleq x().[\![P]\!]_H \\ [\![x \triangleleft inl.P]\!]_H & \triangleq x \triangleleft inl.[\![P]\!]_H \\ [\![x \triangleleft inr.P]\!]_H & \triangleq x \triangleleft inr.[\![P]\!]_H \\ [\![x \triangleright \{inl: P; inr: Q\}]\!]_H & \triangleq x \triangleright \{inl: [\![P]\!]_H; inr: [\![Q]\!]_H\} \\ [\![x \not\leftarrow N]\!]_H & \triangleq x \not\leftarrow N \end{array}$$

Fission is extended to Multiset CP as follows:

$$\begin{array}{ll} [\![P]\!]_H & \triangleq [\![P]\!]_H \\ [\![\mathscr{P} \parallel \mathscr{Q}]\!]_H & \triangleq [\![\mathscr{P}]\!]_H \parallel [\![\mathscr{Q}]\!]_H \end{array}$$

Fission is extended to process contexts, where it preserves holes and otherwise acts as it does on processes. Fission is the identity on types and typing environments, mapping each connective in CP to the same connective in HCP.

Fission distributes over arbitrary process contexts.

Proposition 3.74. $[\![(P[Q])]\!]_H = ([\![P[\cdot]]\!]_H)[[\![Q]\!]_H]$

Proof. By induction on the structure of the evaluation context $P[\cdot]$.

□

Fission preserves types, structural congruence, and reduction.

Proposition 3.75. *If $P \vdash_C \Gamma$, then $[\![P]\!]_H \vdash [\![\Gamma]\!]_H$.*

Proof. By induction on the structure of the derivation of $P \vdash_C \Gamma$.

- Case T-Cut.

  We have

  $$\frac{P \vdash_c \Gamma, x : A \qquad Q \vdash_c \Delta, \bar{x} : \overline{A}}{(\nu x \bar{x})(P \parallel Q) \vdash_c \Gamma, \Delta}\ \text{T-Cut}$$

  By induction, $[\![P]\!]_H \vdash [\![\Gamma]\!]_H, \mathsf{x} : [\![A]\!]_H$ and $[\![Q]\!]_H \vdash [\![\Delta]\!]_H, \bar{\mathsf{x}} : [\![\overline{A}]\!]_H$.

  The result follows as

  $$\frac{\dfrac{[\![P]\!]_H \vdash [\![\Gamma]\!]_H, \mathsf{x} : [\![A]\!]_H \qquad [\![Q]\!]_H \vdash [\![\Delta]\!]_H, \bar{\mathsf{x}} : [\![\overline{A}]\!]_H}{[\![P]\!]_H \parallel [\![Q]\!]_H \vdash [\![\Gamma]\!]_H, \mathsf{x} : [\![A]\!]_H \parallel [\![\Delta]\!]_H, \bar{\mathsf{x}} : [\![\overline{A}]\!]_H}\ \text{T-Par}}{(\nu \mathsf{x} \bar{\mathsf{x}})([\![P]\!]_H \parallel [\![Q]\!]_H) \vdash [\![\Gamma]\!]_H, [\![\Delta]\!]_H}$$

- Case T-Send.

  We have

  $$\frac{P \vdash_c \Gamma, y : A \qquad Q \vdash_c \Delta, x : B}{x[y].(P \parallel Q) \vdash_c \Gamma, \Delta, x : A \otimes B}\ \text{T-Send}$$

  By induction, $[\![P]\!]_H \vdash \Gamma, y : A$ and $[\![Q]\!]_H \vdash \Delta, x : B$.

  The result follows as

  $$\frac{\dfrac{[\![P]\!]_H \vdash [\![\Gamma]\!]_H, \mathsf{y} : [\![A]\!]_H \qquad [\![Q]\!]_H \vdash [\![\Delta]\!]_H, \mathsf{x} : [\![B]\!]_H}{[\![P]\!]_H \parallel [\![Q]\!]_H \vdash [\![\Gamma]\!]_H, \mathsf{y} : [\![A]\!]_H \parallel [\![\Delta]\!]_H, \mathsf{x} : [\![B]\!]_H}\ \text{T-Par}}{\mathsf{x}[\mathsf{y}].([\![P]\!]_H \parallel [\![Q]\!]_H) \vdash [\![\Gamma]\!]_H, [\![\Delta]\!]_H, \mathsf{x} : [\![A]\!]_H \otimes [\![B]\!]_H}$$

- Case T-Close. We have

  $$\frac{}{x[].\,0 \vdash_c x : \mathbf{1}}\ \text{T-Close}$$

  The result follows as

  $$\frac{\dfrac{}{0 \vdash \circ}}{\mathsf{x}[].\,0 \vdash \mathsf{x} : \mathbf{1}}$$

- Cases T-Link, T-Recv, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, T-Absurd.

  The result follows immediately from the HCP rules T-Link, T-Recv, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, T-Absurd and the induction hypotheses (if any).

□

Proposition 3.76. *If* $P \equiv_c Q$, *then* $[\![P]\!]_H \equiv [\![Q]\!]_H$.

Proof. The cases for reflexivity, symmetry, transitivity, and congruence follow from the same properties of the structural congruence of HCP. The cases for the axioms are as follows:

- Case SC-LinkComm.

  Immediately, by SC-LinkComm.

- Case SC-CutComm.

  Immediately, by SC-NewComm and SC-ParComm.

- Case SC-CutAssoc.

  Immediately, by SC-ScopeExt, SC-NewAssoc, and SC-ParAssoc.

□

Proposition 3.77. *If* $P \longrightarrow_C Q$, *then* $[\![P]\!]_H \longrightarrow [\![Q]\!]_H$.

Proof. By induction on the structure of the derivation $P \longrightarrow_C Q$.

- Cases E-Link, E-Send, E-Close, E-Select$_1$, and E-Select$_2$.

  Immediately, by E-Link, E-Send, E-Close, E-Select$_1$, and E-Select$_2$, respectively.

- Case E-Equiv.

  By E-Equiv, the induction hypothesis, and Proposition 3.76.

- Case E-Cong.

  By E-Cong, the induction hypothesis, and Proposition 3.74.

□

Fusion

Fusion is the partial inverse of fission. It fuses an HCP name restriction and parallel composition into a CP cut, an HCP send and parallel composition into a CP send, and an HCP close and parallel composition into a CP close, but *only* if the HCP process already has the form of a CP process.

$(\nu x \bar{x})\_(P \parallel Q)$	to	$(\nu x \bar{x})(P \parallel Q)$
$x[y].\_(P \parallel Q)$	to	$x[y].(P \parallel Q)$
$x[].\_0$	to	$x[].0$

Visually, its definition likewise resembles an elaborate identity function.

Definition 3.78. *Fusion,* $[\![P]\!]_C$, *translates HCP processes into CP processes by fusing adjacent name restrictions and parallel compositions into cuts,*

and likewise fusing send and close. Fusion is partial.

$$
\begin{array}{ll}
[\![x{\leftrightarrow}y]\!]_c & \triangleq x{\leftrightarrow}y \\
[\![(\nu x\bar{x})(P \parallel Q)]\!]_c & \triangleq (\nu x\bar{x})([\![P]\!]_c \parallel [\![Q]\!]_c) \\
[\![x[y].(P \parallel Q)]\!]_c & \triangleq x[y].([\![P]\!]_c \parallel [\![Q]\!]_c) \\
[\![x(y).P]\!]_c & \triangleq x(y).[\![P]\!]_c \\
[\![x[].0]\!]_c & \triangleq x[].0 \\
[\![x().P]\!]_c & \triangleq x().[\![P]\!]_c \\
[\![x \triangleleft inl.P]\!]_c & \triangleq x \triangleleft inl.[\![P]\!]_c \\
[\![x \triangleleft inr.P]\!]_c & \triangleq x \triangleleft inr.[\![P]\!]_c \\
[\![x \triangleright \{inl: P; inr: Q\}]\!]_c & \triangleq x \triangleright \{inl: [\![P]\!]_c; inr: [\![Q]\!]_c\} \\
[\![x \lightning N]\!]_c & \triangleq x \lightning N
\end{array}
$$

Fusion is extended to Multiset CP as follows:

$$
[\![P]\!]_{\mathscr{C}} \triangleq \begin{cases} [\![Q]\!]_{\mathscr{C}} \parallel [\![R]\!]_{\mathscr{C}} & \text{if } P = Q \parallel R \\ [\![P]\!]_c & \text{otherwise} \end{cases}
$$

Fusion is extended to process contexts, where it preserves holes and otherwise acts as it does on processes. Fusion is the identity on types and typing environments, mapping each connective in HCP to the same connective in CP.

Fusion is the partial inverse of fission.

Fusion distributes over arbitrary process contexts.

Lemma 3.79. $[\![P[Q]]\!]_c = [\![P[\cdot]]\!]_c[[\![Q]\!]_c]$.

Fusion preserves types.

Proposition 3.80. *If* $[\![P]\!]_H \vdash [\![\Gamma]\!]_H$, *then* $P \vdash_c \Gamma$.

Proof. By induction on the structure of the derivation $[\![P]\!]_H \vdash [\![\Gamma]\!]_H$.

- Cases T-New, T-Send, and T-Close.

  The result follows from the CP rules T-Cut, T-Send, and T-Close and inversion on the structure of the CP process P, which establishes that the derivations constructed in Case (⇒) are the only derivations.

- Cases T-Par and T-Halt.

  Impossible, by inversion on the structure of the CP process P.

- Cases T-Link, T-Recv, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, and T-Absurd.

  The result follows from the induction hypotheses (if any) and the CP rule of the same name, i.e. T-Link, T-Recv, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, and T-Absurd, respectively.

□

Fusion is defined for all disentangled processes.

Lemma 3.81. *If* $P \vdash \mathcal{G}$ *and* $Q \in \{\!|P|\!\}$*, then* $[\![Q]\!]_{\mathscr{E}}$ *is defined.*

Proof. By induction on the structure of Q with Lemma 3.60.

□

The fusions of the disentangled processes are equivalent up to CP's link-preserving structural congruence.

Lemma 3.82. *If* $P \vdash \mathcal{G}$ *and* $Q, R \in \{\!|P|\!\}$*, then* $[\![Q]\!]_{\mathscr{E}} \overset{\text{L}}{\equiv}_{\mathscr{E}} [\![R]\!]_{\mathscr{E}}$.

Proof. The proof is an extension to the proof of Proposition 3.61, which proves the following two facts:

1. In the proof for Lemma 3.56, under the case $V_T > 1$, we pick an arbitrary leaf of the connection tree T. Any choice is equivalent up to CP's structural congruence.

   Assume that P_i and P_j are both leaves of T. If we pick P_i and then P_j, we get (a), but if we pick P_j and then P_i, we get (b).

(a)	(b)
$Q \parallel (\nu x\bar{x})(P_i \parallel (\nu y\bar{y})(P_j \parallel R'))$	$Q \parallel (\nu y\bar{y})(P_j \parallel (\nu x\bar{x})(P_i \parallel R'))$

   As P_i and P_j are both leaves of T, $x, \bar{x} \notin \mathrm{fn}(P_j)$ and $y, \bar{y} \notin \mathrm{fn}(P_i)$, which means the two processes are equivalent by SC-CutComm and SC-CutAssoc.

   If we pick P_i, then pick a number of processes that have become leaves by the removal of P_i and successive choices, *and then* pick P_j, we get (a), where there is some (right-branching) evaluation context $\mathscr{E}$ between P_i and P_j, but if we pick P_j, then pick P_i, and then pick the processes in $\mathscr{E}$, we get (b).

(a)	(b)
$Q \parallel (\nu x\bar{x})(P_i \parallel \mathscr{E}[(\nu y\bar{y})(P_j \parallel R')])$	$Q \parallel (\nu y\bar{y})(P_j \parallel (\nu x\bar{x})(P_i \parallel \mathscr{E}[R']))$

   As P_j is a leaf of T, $\mathrm{fn}(P_j) \cap \mathrm{bn}(\mathscr{E}) = \varnothing$, which means the two processes are equivalent by Lemma 3.24 and Lemma 3.25.

2. In the proof for Proposition 3.57, under the case where $\mathcal{G}$ is of the form $\mathcal{G} \parallel \Gamma$, we pick an arbitrary tree of the connection forest G. Any choice is equivalent up to Multiset CP's structural congruence.

   The result follows immediate from the multiset structure of Multiset CP's processes, i.e. by SC-ParComm and SC-ParAssoc.

□

Disentanglement into CP is the composition of disentanglement and fusion.

Definition 3.83 (Disentanglement into CP)**.** *The* disentanglement *of* P *into CP,* $\llparenthesis P \rrparenthesis_{\mathscr{C}}$*, is the composition of disentanglement and fusion. Formally,*

$$\llparenthesis P \rrparenthesis_{\mathscr{C}} \triangleq \{[\![Q]\!]_{\mathscr{C}} \mid Q \in \{\!|P|\!\}\}$$

By Lemma 3.81 and Lemma 3.82, the set $\llparenthesis P \rrparenthesis_{\mathscr{C}}$ *is non-empty for any* P*, and all elements of* $\llparenthesis P \rrparenthesis_{\mathscr{C}}$ *are equivalent to each other under Multiset CP's link-preserving structural congruence. I use* $\llparenthesis \cdot \rrparenthesis_{\mathscr{C}}$ *as functional up to structural congruence, and write* $\llparenthesis P \rrparenthesis_{\mathscr{C}} \mathrel{\overset{!}{\equiv}}_{\mathscr{C}} \mathscr{Q}$ *to mean* $\mathscr{Q}$ *is an arbitrary element of* $\llparenthesis P \rrparenthesis_{\mathscr{C}}$*.*

Disentanglement into CP does not distribute over process contexts, as disentanglement normalises the entire maximum configuration context of a process in a single step. Consequently, when doing proof by induction over the structure of a process, we must generalise the cases for name restriction and parallel composition to cover the entire maximum configuration context, as done in, e.g. the proof of Proposition 3.85.

Disentanglement into CP preserves types, structural congruence, and reduction.

Proposition 3.84. *If* $P \vdash \mathscr{G}$ *and* $\llparenthesis P \rrparenthesis_{\mathscr{C}} \mathrel{\overset{!}{\equiv}}_{\mathscr{C}} \mathscr{P}$*, then* $\mathscr{P} \vdash_{\mathscr{C}} [\![\mathscr{G}]\!]_{\mathscr{C}}$*.*

Proof. By Proposition 3.61, Lemma 3.27, Proposition 3.80, and Lemma 2.24.

□

Proposition 3.85. *If* $P \vdash \mathscr{G}$ *and* $P \equiv Q$*, then* $\llparenthesis P \rrparenthesis_{\mathscr{C}} \equiv_{\mathscr{C}} \llparenthesis Q \rrparenthesis_{\mathscr{C}}$*.*

Proof. By induction on the derivation of $P \vdash \mathscr{G}$ and inversion on P. There are three distinct cases, based on whether P is a link, some other ready process, or some other process construct.

- Case P is of the form $x \leftrightarrow y$.

  By inversion on $P \equiv Q$, there are two cases. Either $Q = x \leftrightarrow y$, in which case the result follows by reflexivity, or $Q = y \leftrightarrow x$, in which case the result follows by SC-LinkComm.

- Case P is of the form $x[y].\,P'$, $x(y).\,P'$, $x[].\,P'$, $x().\,P'$, $x \triangleleft \mathsf{inl}.\,P'$, or $x \triangleleft \mathsf{inr}.\,P'$.

  By inversion on $P \equiv Q$, Q is of the form $x[y].\,Q'$, $x(y).\,Q'$, $x[].\,Q'$, $x().\,Q'$, $x \triangleleft \mathsf{inl}.\,Q'$, or $x \triangleleft \mathsf{inr}.\,Q'$. By the induction hypothesis, $\llparenthesis P' \rrparenthesis_{\mathscr{C}} \equiv_{\mathscr{C}} \llparenthesis Q' \rrparenthesis_{\mathscr{C}}$. The result follows by congruence.

- Case P is of the form $x \triangleright \{\mathsf{inl}: P_1; \mathsf{inr}: P_2\}$.

  As above, but with two appeals to the induction hypothesis.

- Case P is of the form $x \not\leftrightarrow N$.

  By inversion on $P \equiv Q$, $P = Q$. The result follows by reflexivity.

- Case P is of the form $(\nu x\bar{x})P'$, $P_1 \parallel P_2$, or 0.

  By Lemma 3.11, there exists some R such that $P \overset{\text{LS}}{\equiv} R$ and $R \overset{\text{D}}{\equiv} Q$. By Lemma 3.65, $\{\!|P|\!\} = \{\!|R|\!\}$. Hence, $[\![P]\!]_{\mathscr{C}} = [\![R]\!]_{\mathscr{C}}$. We have R is of the form $\mathscr{C}^n[T_1, ..., T_n]$ for some $n \geq 0$. By inversion on the structure of $\mathscr{C}[\cdot]$, decompose $R \overset{\text{D}}{\equiv} Q$ into separate structural congruences for each thread $T_i \equiv T_i'$ for $1 \leq i \leq n$. (The "deep" restriction is not lost. For ready processes, $\overset{\text{D}}{\equiv}$ is exactly $\equiv$.) By the induction hypothesis, $[\![T_i]\!]_{\mathscr{C}} \equiv_{\mathscr{C}} [\![T_i']\!]_{\mathscr{C}}$ for $1 \leq i \leq n$. The result follows by composing these intermediate results using Lemma 3.64.

□

Lemma 3.86. *Any derivation of a reduction $P \longrightarrow Q$ can be rewritten to*

$$\dfrac{P \equiv \mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)] \qquad \dfrac{\dfrac{}{(\nu x\bar{x})(P_1 \parallel P_2) \longrightarrow R}\, r}{\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)] \longrightarrow \mathscr{E}[R]} \qquad \mathscr{E}[R] \equiv Q}{P \longrightarrow Q}$$

where r is E-Link, E-Send, E-Close, E-Select$_1$, or E-Select$_2$.

Proof. By induction on the derivation of the reduction $P \longrightarrow Q$. In the cases for E-Link, E-Send, E-Close, E-Select$_1$, or E-Select$_2$, the result follows immediately. In the case for E-Cong, the result follows by composing the two evaluation contexts. In the case for E-Equiv, the result follows by composing the structural congruences using transitivity.

□

Proposition 3.87. *If $P \vdash \mathscr{G}$ and $P \longrightarrow Q$, then $[\![P]\!]_{\mathscr{C}} \longrightarrow_{\mathscr{C}} [\![Q]\!]_{\mathscr{C}}$.*

Proof. By Lemma 3.86, rewrite the derivation of the reduction $P \longrightarrow Q$ to

$$\dfrac{P \equiv (\nu x\bar{x})(P_1 \parallel P_2) \qquad \dfrac{\dfrac{}{(\nu x\bar{x})(P_1 \parallel P_2) \longrightarrow R}\, r}{\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)] \longrightarrow \mathscr{E}[R]} \qquad \mathscr{E}[R] \equiv Q}{P \longrightarrow Q}$$

where r is E-Link, E-Send, E-Close, E-Select$_1$, or E-Select$_2$.

By Proposition 3.85, $[\![P]\!]_{\mathscr{C}} \equiv_{\mathscr{C}} [\![\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)]]\!]_{\mathscr{C}}$ and $[\![\mathscr{E}[R]]\!]_{\mathscr{C}} \equiv_{\mathscr{C}} [\![Q]\!]_{\mathscr{C}}$.

Pick an arbitrary $P' \in [\![\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)]]\!]_{\mathscr{C}}$.

If r is not E-Link, both P_1 and P_2 are ready, and, by Lemma 3.63, preserved up to link-preserving deep structural congruence, by disentanglement. By Lemma 2.21 and the fact that P' is in right-branching form, P' is either of the form $\mathscr{F}_1[(\nu x\bar{x})(P_1' \parallel \mathscr{F}_2[P_2'])]$ or of the form $\mathscr{F}_2[(\nu \bar{x}x)(P_2' \parallel \mathscr{F}_1[P_1'])]$,

where $P'_1 \in \llbracket P_1 \rrbracket_{\mathscr{C}}$ and $P'_2 \in \llbracket P_2 \rrbracket_{\mathscr{C}}$. In the first case, the CP reduction is constructed as:

$$\begin{array}{ll} \mathscr{F}_1[(\nu x\bar{x})(P'_1 \parallel \mathscr{F}_2[P'_2])] \overset{\text{LS}}{\equiv}_{\mathscr{C}} & \langle\text{by Lemma 2.23}\rangle \\ \mathscr{F}_1[\mathscr{F}_2[(\nu x\bar{x})(P'_1 \parallel P'_2)]] \longrightarrow_{\mathscr{C}} & \langle\text{by } r\rangle \\ \mathscr{F}_1[\mathscr{F}_2[R']] & \end{array}$$

By case analysis on r, it follows that $\mathscr{F}_1[\mathscr{F}_2[R']] \overset{\text{LS}}{\equiv}_C \llbracket \mathscr{E}[R] \rrbracket_{\mathscr{C}}$.

In the second case, the CP reduction is constructed similarly.

If r *is* E-Link, the CP reduction is constructed similarly, since, if P_1 is of the form $x{\leftrightarrow}y$, the thread in P_2 that contains $\bar{x}$ is preserved up to fusion and link-preserving deep structural congruence in P'.

□

3.2.8 Label-Transition System and Harmony

In the previous sections, we got HCP up to speed. We adapted CP's metatheory and showed that HCP enjoys all the same properties, and characterised the exact correspondence between CP and HCP by disentanglement.

In this section, we get to enjoy some of the spoils of all that work. We introduce the beginnings of a behavioural theory for HCP, *just enough* to relate the use of the label-transition system in the following chapter to the reduction semantics, and to use as a springboard to find and fix further flaws in our present formulation of HCP in the discussion.

The label-transition semantics for HCP, first introduced by Montesi and Peressotti [2018], are given by labelled transition, which is a ternary relation on two processes and a label. A label is either the internal action label τ, an observable action label π, or a pair of parallel observable action labels $\pi \parallel \bar{\pi}$.

$$\ell ::= \tau \mid \pi \mid \pi \parallel \bar{\pi}$$

The observable action labels correspond to the action prefixes of processes, except for the offer labels $x \triangleright \mathsf{inl}$ and $x \triangleright \mathsf{inr}$, which denote the offer of inl or inr, respectively, in $x \triangleright \{\mathsf{inl}\colon P; \mathsf{inr}\colon Q\}$.

$$\pi ::= x{\leftrightarrow}y \mid x[y] \mid x(y) \mid x[] \mid x() \mid x \triangleleft \mathsf{inl} \mid x \triangleleft \mathsf{inr} \mid x \triangleright \mathsf{inl} \mid x \triangleright \mathsf{inr}$$

An endpoint occurs free in an action label in the same cases where it would occur free in a process prefixed by that action. Likewise, an endpoint is bound by an action label in the same cases where it would occur bound in a process prefixed by that action. For instance, $x \in \mathrm{fn}(x[y])$ and $y \in \mathrm{bn}(x[y])$. An endpoint occurs free in $\pi \parallel \bar{\pi}$ when it occurs free in either π or $\bar{\pi}$. An endpoint is bound by $\pi \parallel \bar{\pi}$ when it is bound by either π or $\bar{\pi}$. The label τ has no free or bound names.

Definition 3.88 (Label-Transition)**.** *Label-transition, written* $P \xrightarrow{\ell} Q$*, is the smallest ternary relation on two processes and one label defined by the rules in Figure 3.4.*

What does a label-transition mean? If a process has an observable action, i.e. a transition with an observable action label π, it means the action π is observable—some unblocked sub-process is prefixed by the action π on a free endpoint. If a process has pair of parallel observable actions, i.e. a transition with a label $\pi \parallel \bar{\pi}$, it means both actions are observable and independent. If a process has a τ-transition, it has some internal, unobservable reduction. To use familiar terminology:

$$\begin{aligned} P \xrightarrow{\pi} Q &\iff P \equiv \pi.P' \parallel R \\ P \xrightarrow{\pi\parallel\bar{\pi}} Q &\iff P \equiv \pi.P_1 \parallel \bar{\pi}.P_2 \parallel R \\ P \xrightarrow{\tau} \equiv Q &\iff P \longrightarrow Q \end{aligned}$$

These properties are intended to provide an *informal* intuition. They are not well-formed for the offer, since, e.g. "$x \triangleright \mathrm{inl}.\, P$" is a syntactically ill-formed process. However, the intuition still applies:

$$\begin{aligned} P \xrightarrow{x \triangleright \mathrm{inl}} Q &\iff P \equiv x \triangleright \{\mathrm{inl}\colon P_1; \mathrm{inr}\colon P_2\} \parallel R \\ P \xrightarrow{x \triangleright \mathrm{inr}} Q &\iff P \equiv x \triangleright \{\mathrm{inl}\colon P_1; \mathrm{inr}\colon P_2\} \parallel R \\ P \xrightarrow{x \triangleleft \mathrm{inl} \parallel x \triangleright \mathrm{inl}} Q &\iff P \equiv x \triangleleft \mathrm{inl}.\, P_1 \parallel x \triangleright \{\mathrm{inl}\colon P_2; \mathrm{inr}\colon P_3\} \parallel R \\ P \xrightarrow{x \triangleleft \mathrm{inr} \parallel x \triangleright \mathrm{inr}} Q &\iff P \equiv x \triangleleft \mathrm{inr}.\, P_1 \parallel x \triangleright \{\mathrm{inl}\colon P_2; \mathrm{inr}\colon P_3\} \parallel R \end{aligned}$$

Label-transitions are closed under independent evaluation contexts by Str-Cong, which requires that the bound and free endpoints of the evaluation context and the label are disjoint. Consequently, τ-transitions, whose labels have neither free nor bound endpoints, are closed under arbitrary evaluation contexts.

The label-transition system does not use the structural congruence. As a consequence of a label-transition, a process may change *locally* and it may close the channel or open a new channel. However, the structure of any name restrictions and parallel compositions *unrelated to* the communication does not change. On the other hand, the structural congruence does not essentially alter actions. Label-transition and structural congruence are mutually invariant:

$$P \equiv \xrightarrow{\ell} Q \iff P \xrightarrow{\ell} \equiv Q$$

Reductions may contain arbitrary structural congruence, and change the process structure in ways that are not strictly necessary for the reduction itself. Hence, while a τ-transition corresponds to a reduction, a reduction corresponds to a τ-transition *and* a structural congruence.

Label-transition and structural congruence are mutually invariant.

Action Rules

$x \leftrightarrow y \xrightarrow{x \leftrightarrow y} 0$ ACT-LINK$_1$ $\quad x \leftrightarrow y \xrightarrow{y \leftrightarrow x} 0$ ACT-LINK$_2$

$x[y].\,P \xrightarrow{x[y]} P$ ACT-SEND $\quad x \triangleleft \mathrm{inl}.\,P \xrightarrow{x \triangleleft \mathrm{inl}} P$ ACT-SELECT$_1$

$x(y).\,P \xrightarrow{x(y)} P$ ACT-RECV $\quad x \triangleleft \mathrm{inr}.\,P \xrightarrow{x \triangleleft \mathrm{inr}} P$ ACT-SELECT$_2$

$x[].\,P \xrightarrow{x[]} P$ ACT-CLOSE $\quad x \triangleright \{\mathrm{inl}\colon P; \mathrm{inr}\colon Q\} \xrightarrow{x \triangleright \mathrm{inl}} P$ ACT-OFFER$_1$

$x().\,P \xrightarrow{x()} P$ ACT-WAIT $\quad x \triangleright \{\mathrm{inl}\colon P; \mathrm{inr}\colon Q\} \xrightarrow{x \triangleright \mathrm{inr}} Q$ ACT-OFFER$_2$

Structural Rules

STR-PAR

$$\frac{P \xrightarrow{\pi} P' \qquad Q \xrightarrow{\bar{\pi}} Q' \qquad \mathrm{bn}(\pi) \cap \mathrm{bn}(\bar{\pi}) = \varnothing}{P \parallel Q \xrightarrow{\pi \| \bar{\pi}} P' \parallel Q'}$$

STR-CONG

$$\frac{P \xrightarrow{\pi} P' \qquad (\mathrm{bn}(\mathscr{E}) \cup \mathrm{fn}(\mathscr{E})) \cap (\mathrm{bn}(\pi) \cup \mathrm{fn}(\pi)) = \varnothing}{\mathscr{E}[P] \xrightarrow{\pi} \mathscr{E}[P']}$$

Communication Rules

TAU-LINK

$$\frac{P \xrightarrow{x \leftrightarrow y} P'}{(\nu x\bar{x})P \xrightarrow{\tau} P'\{y/\bar{x}\}}$$

TAU-SEND-RECV

$$\frac{P \xrightarrow{x[y] \| \bar{x}(\bar{y})} P'}{(\nu x\bar{x})P \xrightarrow{\tau} (\nu x\bar{x})(\nu y\bar{y})P'}$$

TAU-CLOSE-WAIT

$$\frac{P \xrightarrow{x[] \| \bar{x}()} P'}{(\nu x\bar{x})P \xrightarrow{\tau} P'}$$

TAU-SELECT-OFFER$_1$

$$\frac{P \xrightarrow{x \triangleleft \mathrm{inl} \| \bar{x} \triangleright \mathrm{inl}} P'}{(\nu x\bar{x})P \xrightarrow{\tau} (\nu x\bar{x})P'}$$

TAU-SELECT-OFFER$_2$

$$\frac{P \xrightarrow{x \triangleleft \mathrm{inr} \| \bar{x} \triangleright \mathrm{inr}} P'}{(\nu x\bar{x})P \xrightarrow{\tau} (\nu x\bar{x})P'}$$

Figure 3.4: Label-Transition System for Hypersequent CP

Lemma 3.89. *If* $P \equiv\!\xrightarrow{\ell} Q$, *then* $P \xrightarrow{\ell}\!\equiv Q$.

Proof. By induction on the derivation of the structural congruence $P \equiv Q$ and inversion on the transition $P \xrightarrow{\ell} P'$.

(For the full proof, see § 3.5.)

□

Any derivation of a label-transition can be normalised, by combining successive uses of the Str-Cong rule. Lemmas 3.90 to 3.92 normalise action transitions, parallel action transitions, and τ-transitions, respectively.

Lemma 3.90. *Any derivation of a transition* $P \xrightarrow{\pi} Q$ *can be rewritten to*

$$\dfrac{\dfrac{}{P_1 \xrightarrow{\pi} P_1'}\,a}{\mathscr{F}_1[P_1] \xrightarrow{\pi} \mathscr{F}_1[P_1']}\,\textsc{Str-Cong}$$

where a is Act-Link$_1$, Act-Link$_2$, Act-send, Act-Recv, Act-Close, Act-Wait, Act-Select$_1$, Act-Select$_2$, Act-Offer$_1$, *or* Act-Offer$_2$.

Proof. By induction on the derivation of the transition $P \xrightarrow{\pi} Q$. The cases for Act-Link$_1$, Act-Link$_2$, Act-send, Act-Recv, Act-Close, Act-Wait, Act-Select$_1$, Act-Select$_2$, Act-Offer$_1$, or Act-Offer$_2$ follow immediately. The case for Str-Cong follows by induction, composing the evaluation context introduced by Str-Cong with the evaluation context in the induction hypothesis.

□

Lemma 3.91. *Any derivation of a transition* $P \xrightarrow{\pi\|\bar{\pi}} Q$ *can be rewritten to*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P_1 \xrightarrow{\pi} P_1'}\,a}{\mathscr{F}_1[P_1] \xrightarrow{\pi} \mathscr{F}_1[P_1']}\,\textsc{Str-Cong} \quad \dfrac{\dfrac{}{P_2 \xrightarrow{\bar{\pi}} P_2'}\,\bar{a}}{\mathscr{F}_2[P_2] \xrightarrow{\bar{\pi}} \mathscr{F}_2[P_2']}\,\textsc{Str-Cong}}{\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2] \xrightarrow{\pi\|\bar{\pi}} \mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']}\,\textsc{Str-Par}}{\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]] \xrightarrow{\pi\|\bar{\pi}} \mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]}\,\textsc{Str-Cong}$$

where a and ā are one of Act-Link$_1$, Act-Link$_2$, Act-send, Act-Recv, Act-Close, Act-Wait, Act-Select$_1$, Act-Select$_2$, Act-Offer$_1$, *or* Act-Offer$_2$.

Proof. By induction on the derivation of the transition $P \xrightarrow{\pi} Q$. The case for Str-Par follows by Lemma 3.90. The case for Str-Cong follows by induction, composing the evaluation context introduced by Str-Cong with the evaluation context in the induction hypothesis.

□

Lemma 3.92. *Any derivation of a transition* $P \xrightarrow{\tau} Q$ *can be rewritten to*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P_1 \xrightarrow{\pi} P_1'}\,a}{\mathscr{F}_1[P_1] \xrightarrow{\pi} \mathscr{F}_1[P_1']}\,\textsc{Str-Cong} \quad \dfrac{\dfrac{}{P_2 \xrightarrow{\bar{\pi}} P_2'}\,\bar{a}}{\mathscr{F}_2[P_2] \xrightarrow{\bar{\pi}} \mathscr{F}_2[P_2']}\,\textsc{Str-Cong}}{\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2] \xrightarrow{\pi\|\bar{\pi}} \mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']}\,\textsc{Str-Par}}{\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]] \xrightarrow{\pi\|\bar{\pi}} \mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]}\,\textsc{Str-Cong}}{(\nu x\bar{x})\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]] \xrightarrow{\tau} \mathscr{E}_2[\mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]]}\,t}{\mathscr{E}_1[(\nu x\bar{x})\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]]] \xrightarrow{\tau} \mathscr{E}_1[\mathscr{E}_2[\mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]]]}\,\textsc{Str-Cong}$$

where t is one of Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, *or* Tau-Select-Offer$_2$, *and a and $\bar{a}$ are one of* Act-Link$_1$, Act-Link$_2$, Act-send, Act-Recv, Act-Close, Act-Wait, Act-Select$_1$, Act-Select$_2$, Act-Offer$_1$, *or* Act-Offer$_2$.

Proof. By induction on the structure of the transition. The cases for Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, or Tau-Select-Offer$_2$ follow by Lemma 3.91. The case for Str-Cong follows by induction, composing the evaluation context introduced by Str-Cong with the evaluation context in the induction hypothesis.

□

The equivalences between action transitions and parallel action transitions and structural congruence follow as corollaries from Lemmas 3.90 and 3.91 and Corollary 3.26, and their converses follow from Lemma 3.89.

$$P \xrightarrow{\pi} Q \iff P \equiv \pi.P' \parallel R$$
$$P \xrightarrow{\pi\|\bar{\pi}} Q \iff P \equiv \pi.P_1 \parallel \bar{\pi}.P_2 \parallel R$$

Harmony follows by relating the normal-form derivations of τ-transition (Lemma 3.92) and reduction (Lemma 3.86).

Proposition 3.93 (Harmony)**.** *If* $P \vdash \mathscr{G}$, *then* $P \xrightarrow{\tau} \equiv Q \iff P \longrightarrow Q$.

Proof. There are two cases:

- Case ($\Rightarrow$).

By Lemma 3.92, rewrite the derivation of the transition $P \xrightarrow{\tau} Q$ to

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P_1 \xrightarrow{\pi} P_1'}\,a}{\mathscr{F}_1[P_1] \xrightarrow{\pi} \mathscr{F}_1[P_1']}\,\textsc{Str-Cong} \quad \dfrac{\dfrac{}{P_2 \xrightarrow{\bar{\pi}} P_2'}\,\bar{a}}{\mathscr{F}_2[P_2] \xrightarrow{\bar{\pi}} \mathscr{F}_2[P_2']}\,\textsc{Str-Cong}}{\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2] \xrightarrow{\pi\|\bar{\pi}} \mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']}\,\textsc{Str-Par}}{\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]] \xrightarrow{\pi\|\bar{\pi}} \mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]}\,\textsc{Str-Cong}}{(\nu x\bar{x})\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]] \xrightarrow{\tau} \mathscr{E}_2[\mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]]}\,t}{\mathscr{E}_1[(\nu x\bar{x})\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]]] \xrightarrow{\tau} \mathscr{E}_1[\mathscr{E}_2[\mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]]]}\,\textsc{Str-Cong}$$

where t is one of Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, or Tau-Select-Offer$_2$, and a and $\bar{a}$ are one of Act-Link$_1$, Act-Link$_2$, Act-send, Act-Recv, Act-Close, Act-Wait, Act-Select$_1$, Act-Select$_2$, Act-Offer$_1$, or Act-Offer$_2$. By repeated application of Lemmas 3.24 and 3.25:

$$\mathscr{E}_1[(\nu x\bar{x})\mathscr{E}_3[\mathscr{F}_1[P_1] \parallel \mathscr{F}_2[P_2]]] \equiv \mathscr{E}_1[\mathscr{E}_3[\mathscr{F}_1[\mathscr{F}_2[(\nu x\bar{x})(P_1 \parallel P_2)]]]]$$

By case analysis on t and inversion on a and $\bar{a}$, the structure of the messages π and $\bar{\pi}$, and the processes P_1, P_1', P_2, and P_2', and $\mathscr{E}_2$:

$$\mathscr{E}_1[\mathscr{E}_3[\mathscr{F}_1[\mathscr{F}_2[(\nu x\bar{x})(P_1 \parallel P_2)]]]] \longrightarrow \mathscr{E}_1[\mathscr{E}_3[\mathscr{F}_1[\mathscr{F}_2[\mathscr{E}_2[P_1' \parallel P_2']]]]]$$

By repeated application of Lemmas 3.24 and 3.25:

$$\mathscr{E}_1[\mathscr{E}_3[\mathscr{F}_1[\mathscr{F}_2[\mathscr{E}_2[P_1' \parallel P_2']]]]] \equiv \mathscr{E}_1[\mathscr{E}_2[\mathscr{E}_3[\mathscr{F}_1[P_1'] \parallel \mathscr{F}_2[P_2']]]]$$

The result follows by composing these results with E-Equiv.

- Case ($\Leftarrow$).

  By Lemma 3.86, rewrite the derivation of the reduction $P \longrightarrow Q$ to

  $$\dfrac{P \equiv (\nu x\bar{x})(P_1 \parallel P_2) \qquad \dfrac{\dfrac{}{(\nu x\bar{x})(P_1 \parallel P_2) \longrightarrow R}\,r}{\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)] \longrightarrow \mathscr{E}[R]} \qquad \mathscr{E}[R] \equiv Q}{P \longrightarrow Q}$$

  where r is E-Link, E-Send, E-Close, E-Select$_1$, or E-Select$_2$. By case analysis on r and inversion on the structure of P_1, P_2, and R, $(\nu x\bar{x})(P_1 \parallel P_2) \xrightarrow{\tau} R$. By Str-Cong, $\mathscr{E}[(\nu x\bar{x})(P_1 \parallel P_2)] \xrightarrow{\tau} \mathscr{E}[R]$. By Lemma 3.89 and $P \equiv (\nu x\bar{x})(P_1 \parallel P_2)$, $P \xrightarrow{\tau} \equiv \mathscr{E}[R]$. By transitivity, $P \xrightarrow{\tau} \equiv Q$.

□

3.3 Discussion

This section proceeds as follows:

- In § 3.3.1, I introduce an alternative semantics for the absurd offer based on exception handling.
- In § 3.3.2, I introduce synchronous semantics for link.
- In § 3.3.3, I introduce a variant of HCP that decomposes the binary and n-ary offer into the singleton offer and summation, which allows us to type guarded summation and enables us to factor actions out into their own syntactic category.
- In § 3.3.4, I discuss channel names and endpoint names.
- In § 3.3.4, I discuss the relation between tensor and par and send and receive.
- In § 3.3.5, I discuss the relation between HCP and deep inference logic.
- In § 3.3.6, I discuss the relation between HCP and hypersequent calculus.
- In § 3.3.7, I discuss the relation between the empty hypersequent and hypersequent composition in HCP and the $\textsc{Mix}_0$ and $\textsc{Mix}$ rules.

3.3.1 Zap: the Exceptionally Absurd Offer

In this section, we revisit the semantics for the absurd offer. In § 2.1.6, we discussed the 'inert' semantics for the absurd offer. It does nothing, just sits around. If we think of the absurd offer as an exception handler, the inert semantics make sense. It is waiting for an exception that—by the correctness of the type system—will never arrive. However, the absurd offer is not *quite* an exception handler. An exception handler handles an exceptional case—either the computation succeeds, and we compute a value, or, in some exceptional circumstances, something goes wrong, and the exception handler is invoked. The absurd offer deals with the *certainty* that something will go wrong. If you are certain, why bother waiting? Under this view, we refer to the absurd offer as *zap* or *zapper*. It is not inert, and does not wait. It *zaps* channels, kills processes, and shuts the whole thing down.

Wadler's CP handles zap with a commuting conversion, defined on typing derivations:

$$\dfrac{\dfrac{}{a \triangleright \{\} \vdash_C \Gamma, a : \top, x : A} \qquad P \vdash_C \Delta, \bar{x} : \overline{A}}{(\nu x \bar{x})(a \triangleright \{\} \parallel P) \vdash_C \Gamma, \Delta, a : \top} \longrightarrow_C \dfrac{}{a \triangleright \{\} \vdash_C \Gamma, \Delta, a : \top}$$

As mentioned in § 2.1.6, this reduction rule does not satisfy type erasure, since the process $x \triangleright \{\}$ does not record which sessions it is allowed to kill. In HCP and CP, as presented in this thesis, the absurd offer does record these sessions as part of the process syntax. Hence, we could adapt and repair Wadler's reduction rule. To save on ink and eye strain, let us write N, x to mean $N \cup \{x\}$ and N, M to mean $N \cup M$.

$$\dfrac{\bar{x} \in \mathrm{fn}(P) \qquad M \triangleq \mathrm{fn}(P) \setminus \{\bar{x}\}}{(\nu x \bar{x})(w \lightning N, x \parallel P) \longrightarrow w \lightning N, M}\ \textsc{E-Zap-Now}$$

The above rule is correct, in the sense that it matches Wadler's semantics, but it is a bit over-eager. It permits $w \lightning N$ to kill any process that holds any of the names in N, regardless of whether those processes are currently attempting to communicate on those channels. Let us consider what requirements this places on an implementation.

- If we wanted to implement this as part of our channel implementation, it would require every process to always be listening on all of their endpoints.
- We could implement this using another mechanism entirely. For instance, using POSIX Threads, we could implement this by decorating each channel with the thread ID of the process on the other end of the channel, and implement kill using `pthread_kill` or `pthread_cancel`.

In either case, the adoption of E-Zap-Now significantly complicates an implementation in ways that are not obvious from the description of the formal system.

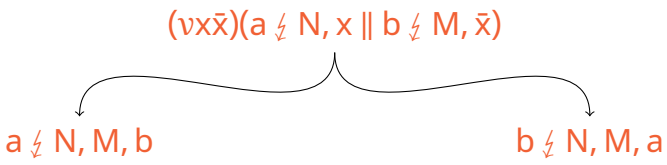
We can address the over-eagerness by restricting the reduction rule to only apply when the other process is ready:

$$\frac{\mathsf{ready}(P, \bar{x}) \qquad M \triangleq \mathsf{fn}(P) \setminus \{\bar{x}\}}{(\nu x \bar{x})(w \lightning N, x \parallel P) \longrightarrow w \lightning N, M} \text{ E-Zap-When-Ready}$$

What does this semantics require of an implementation? Any process only needs to monitor the endpoints it is *ready* on, which is certainly an improvement. What is a zapper ready on? Consider the two processes

$$\text{(a) } (\nu x \bar{x})(a \lightning N, x \parallel \bar{x} \lightning M) \qquad \text{(b) } (\nu x \bar{x})(a \lightning N, x \parallel b \lightning M, \bar{x})$$

For both of these processes to reduce by E-Zap-When-Ready, a zapper $x \lightning N$ must be ready on all its endpoints—x and all the endpoints in N. This reveals an awkwardness: the endpoint x is not special. A zapper may introduce multiple endpoints of type $\top$. From the perspective of the logic, there is no way to tell these apart. The only thing that matters is that there is at least one, and the principal purpose of the syntax $x \lightning N$ is to ensure that. Unfortunately, this makes reduction with E-Zap-When-Ready non-confluent. The second process has two reductions, each of which singles out a different endpoint as special.



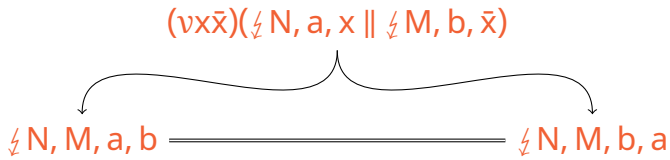
Note that Wadler's semantics has the same problem, but, in his case, the non-confluence for the absurd offer is part of the general non-confluence caused by the commuting conversions (see § 2.3).

The solution is simple: treat all endpoints in a zapper the same. We alter the syntax for zappers to $\lightning\, N$ and alter the typing and reduction rules to reflect the new syntax.

$$P, Q, R ::= \cdots \mid \cancel{x \lightning N} \mid \lightning\, N$$

$$\frac{N = \mathrm{fn}(\Gamma)}{\lightning\, N, x \vdash \Gamma, x : \top}\ \text{T-Zap} \qquad \frac{\mathrm{ready}(P, \tilde{x}) \qquad M \triangleq \mathrm{fn}(P) \setminus \{\tilde{x}\}}{(\nu x \tilde{x})(\lightning\, N, x \parallel P) \longrightarrow \lightning\, N, M}\ \text{E-Zap}$$

The critical pair is resolved. The two processes are definitionally equal:



Alternatively, if one wishes to avoid definitional set equality, the permutation of the endpoints in N could be arranged via the structural congruence. In this case, there is no need to add rules for contraction, since the typing rule guarantees the uniqueness of the endpoints in N.

To reflect the fact that a zapper must be ready on all its endpoints, we alter the definition of *ready*.

Definition 3.94 (Ready)**.** *A process* P *is* ready to act on x, *written* $\mathrm{ready}(P, x)$*, if it is of one of the forms:*

$$\begin{array}{llllll} x{\leftrightarrow}y & x[y].\,P & x[].\,P & x \triangleleft \mathit{inl}.\,P & x \triangleleft \mathit{inr}.\,P & \lightning\, N, x \\ y{\leftrightarrow}x & x(y).\,P & x().\,P & x \triangleright \{\mathit{inl}\colon P; \mathit{inr}\colon Q\} & & \end{array}$$

A process P *is* ready *if it is ready to act on some endpoint.*

The addition of T-Zap and E-Zap preserves all the metatheoretical properties of HCP presented in this chapter:

- The proof of *preservation* requires an additional case for E-Zap (see Proposition 3.95).
- The proof of *progress* requires no changes. All relevant changes are contained in the reduction lemma (see Lemma 3.96).
- The definitions of canonical form and the dependency and connection graphs are unchanged, as are the corresponding proofs of the adequacy of canonical forms, right-branching form, and disentanglement. All relevant changes are contained in the definition of *ready* (see Definition 3.94).
- The calculus continues to be in correspondence with CP, with the caveat that similar changes must be made to CP's syntax and semantics for zappers.
- Finally, we make similar changes to the label-transition system, such that harmony continues to hold.

We presented the updated proofs for preservation and progress and the updated label-transition system.

Proposition 3.95 (Preservation)**.** *If* $P \vdash \mathcal{G}$ *and* $P \longrightarrow Q$*, then* $Q \vdash \mathcal{G}$*.*

Proof. By induction on the derivation of the reduction. The cases for E-Link, E-Send, E-Close, E-Select$_1$, E-Select$_2$, E-Equiv, and E-Cong, are as in Proposition 3.30.

- Case E-Zap.

  The reduction rule guarantees that P is ready on $\bar{x}$. Therefore, P must be typed under a single hyper-environment. Hence, the typing derivation is of the form:

$$\dfrac{\dfrac{\dfrac{N = \mathrm{fn}(\Gamma)}{\lightning N, x \vdash \Gamma, x : A}\,\text{T-Zap} \qquad P \vdash \Delta, \bar{x} : \overline{A}}{\lightning N, x \parallel P \vdash \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}}\,\text{T-Par}}{(\nu x\bar{x})(\lightning N, x \parallel P) \vdash \Gamma, \Delta}\,\text{T-New}$$

  The result is well-typed by the following typing derivation:

$$\dfrac{N, M = \mathrm{fn}(\Gamma, \Delta)}{\lightning N, M \vdash \Gamma, \Delta}\,\text{T-Zap}$$

  The side condition $N, M = \mathrm{fn}(\Gamma, \Delta)$ is satisfied. This follows from the side conditions of typing and reduction, and Proposition 3.18:

$$N = \mathrm{fn}(\Gamma) \quad \text{and} \quad M \triangleq \mathrm{fn}(P) \setminus \{\bar{x}\} = \mathrm{fn}(\Delta)$$

  The other condition is that there must be *some* endpoint $y : \top$ in Γ, Δ. By typing, there must be some endpoint $y : \top$ in $\Gamma, x : A$. There are two cases:

  1. If $x \neq y$, then there exists some endpoint $y : \top \in \Gamma$.
  2. If $x = y$, then $y : \top \notin \Gamma$. By duality, the endpoint $\bar{x}$ must have type $\mathbf{1}$. As there are no introduction forms for $\mathbf{1}$, the endpoint of type $\mathbf{1}$ must have been introduced by T-Zap. Hence, there must be some endpoint $z : \top \in \Delta$.

  The proof for subcase (2) relies on the consistency of HCLL, which we have not explicitly proven, but which follows from disentanglement and the consistency of CLL.

□

Lemma 3.96 (Reduction)**.** *If* $(\nu x\bar{x})(P \parallel Q) \vdash \mathcal{G}$*, and* P *and* Q *are ready to act on* x *and* $\bar{x}$*, respectively, there exists some* R *such that* $(\nu x\bar{x})(P \parallel Q) \longrightarrow R$*.*

Proof. By inversion on the derivation of $(\nu x\bar{x})(P \parallel Q) \vdash \mathcal{G}$.

The ten cases that correspond to E-Link, E-Send, E-Close, E-Select$_1$, E-Select$_2$, and their symmetric variants are as in Lemma 3.34.

There are two new cases, where either P or Q is a zapper, which correspond to E-Zap and its symmetric variant under E-Equiv with SC-ParComm.

□

Proposition 3.97 (Progress)**.** *If* $P \vdash \mathscr{G}$*, then either* P *is in canonical form, or there exists some* Q *such that* $P \longrightarrow Q$*.*

Proof. The proof is unchanged from Proposition 3.35, except that the appeal to Lemma 3.34 is replaced by an appeal to Lemma 3.96.

□

Let us extend the label-transition system and the proof of harmony. We extend the action labels with two new actions.

$$\pi ::= \cdots \mid x \lightning N \mid x \overset{\curvearrowright}{\lightning} N$$

The action $x \lightning N$ denotes sending a kill signal, and receiving the unused endpoints N from the killed process. The action $x \overset{\curvearrowright}{\lightning} N$ denotes the dual, receiving a kill signal, and sending the unused endpoints N.

We add two corresponding action transition rules and one corresponding communication rule.

Action Rules

$$\lightning N, x \xrightarrow{x \lightning M} \lightning N, M \qquad \textsc{Act-Kill}$$

$$P \xrightarrow{x \overset{\curvearrowright}{\lightning} N} 0 \qquad \textsc{Act-Die}$$

if $\mathrm{ready}(P, x)$ and $\neg\, \mathrm{zapper}(P)$ where $N \triangleq \mathrm{fn}(P) \setminus \{x\}$

Communication Rules

$$\textsc{Tau-Kill-Die}\quad \frac{P \xrightarrow{x \lightning N \| \tilde{x} \overset{\curvearrowright}{\lightning} N} P'}{(\nu x \tilde{x})P \xrightarrow{\tau} P'}$$

Proposition 3.98 (Harmony)**.** *If* $P \vdash \mathscr{G}$*, then* $P \xrightarrow{\tau}\!\equiv Q \iff P \longrightarrow Q$*.*

Proof. The proof proceeds as Proposition 3.93. In case ($\Rightarrow$), there is an additional case for Tau-Kill-Die with Act-Kill and Act-Die. In case ($\Leftarrow$), there is an additional case for E-Zap.

□

A Zapper Is Made of Other, Smaller Zappers

A zapper must be ready to act on all its endpoints in parallel. Hence, we should think of the zapper $\lightning x_1, \ldots, x_n$ as a representation of the parallel composition of individual zappers for each endpoint, i.e. $\lightning x_1 \parallel \ldots \parallel \lightning x_n$. Under this view, the permutation of the endpoints in a zapper follows

from the permutation of parallel processes under structural congruence, which is rather pleasing. This is exactly how zappers are represented in Fowler's Exceptional GV [EGV, Fowler et al., 2019]. However, this representation is not well-typed in HCP:

$$\text{(a)}\ \frac{}{\lightning a, b \vdash a : T, b : A} \qquad \frac{\dfrac{}{\lightning a \vdash a : T} \quad \dfrac{}{\lightning b \vdash b : A}}{\lightning a \parallel \lightning b \vdash a : T \parallel b : A}\ \text{(b)}$$

To type (b), we would have to change the local constraint that at least one endpoint introduced by the zapper must have type T to the *non-local* constraint that at least one zapper must introduce an endpoint with type T. Since EGV is not interested in preserving consistency and allows exceptions to occur in any context, it does not require this restriction.

To assign (a) and (b) the same type, we would have to admit Mix, which breaks the connection structure of HCP and the correspondence with CLL. Since EGV is not interested in preserving that correspondence for its runtime terms, it admits Mix in its runtime type system.

While the representation $\lightning x_1 \parallel \ldots \parallel \lightning x_n$ is not a good fit for HCP, it provides an intuition for the implementation of zappers. A zapper is not a single monolithic process that takes ownership of all the unused resources of the processes it kills, and the killed process should not transfer ownership of all their unused endpoints to such a monolithic zapper. Zappers are small processes responsible for cleaning up individual resources. Once a zapper has successfully delivered a kill signal, it may terminate. The killed process, upon receiving a kill signal, simply spawns off one zapper process for each of its unused resources, and then terminates.

Could You to Tell Me When You Are Done?

In this section, we have described an exceptional semantics for the absurd offer in which all cancelled endpoints are treated equally. In this final portion of the section, I hope to convince you that there is no well-behaved interpretation that treats one endpoint specially, as implied by the syntax used in Chapter 2 and Chapter 3, and that this is a direct consequence of the fact that the logical rule corresponding to T-Absurd can introduce multiple propositions T which it cannot distinguish.

Let us consider an alternative interpretation for the zapper $x \lightning N$ where the endpoint x *is* special:

- When some process P receives a kill signal on x, it becomes the zapper $x \lightning N$, where N contains its unused endpoints, i.e. $N = \text{fn}(P) \setminus \{x\}$.
- A zapper $x \lightning N$ does the following for each endpoint in N in parallel:
  - It sends a kill signal.

  - It waits to receive a notification to confirm that the process on the other side has finished cleaning up and has terminated.
  - It closes the channel, and removes the endpoint from N.

- When N is empty, the zapper notifies the process on the other side of x that it has finished cleaning up, and terminates.

Such an interpretation is useful. For instance, if we implement processes as OS threads, then the main thread must not terminate before all the child threads have finished cleaning up.

Under this interpretation, a zapper $x \lightning N$ is ready on all the endpoints in N in parallel, but is blocked on x until N is empty. Consider the following reduction rules:

$$(\nu x\bar{x})(w \lightning N, x \parallel \bar{x} \lightning \varnothing) \longrightarrow w \lightning N \qquad \text{E-Zap-Notify}$$
$$(\nu x\bar{x})(w \lightning N, x \parallel P) \longrightarrow (\nu x\bar{x})(w \lightning N, x \parallel \bar{x} \lightning M) \qquad \text{E-Zap-Kill}$$
$$\text{if } \mathrm{ready}(P, \bar{x}) \text{ and } \neg\, \mathrm{zapper}(P)$$
$$\text{where } M \triangleq \mathrm{fn}(P) \setminus \{\bar{x}\}$$

These rules preserve typing. The rule E-Zap-Kill rebinds x and $\bar{x}$ with types $x : \mathbf{0}$ and $\bar{x} : \top$, but this is usual for session types. However, they are only well-behaved under certain circumstances. Consider the two processes

$$\text{(a) } (\nu x\bar{x})(a \lightning N, x \parallel \bar{x} \lightning M) \qquad (\nu x\bar{x})(a \lightning N, x \parallel b \lightning M, \bar{x}) \text{ (b)}$$

If a process up to structural congruence does not contain (b), then progress, termination, and confluence follow by an argument from the structure of the connection graph. To simplify matters, let us consider a fully-connected process with a single zapper

$$\mathscr{C}[a \lightning N, P_1, \ldots, P_n]$$

where each P_i is ready on some bound endpoint x_i. The process reduces to a single zapper in $2n$ steps.

1. By n applications of E-Zap-Kill, following the structure of the connection tree outward from the zapper $a \lightning N$, we convert each process P_i to a zapper $x_i \lightning N_i$ where $N_i = \mathrm{fn}(P_i) \setminus \{x_i\}$.
2. If P_i is a leaf of the connection graph, it is of the form $x_i \lightning \varnothing$. By n application of E-Zap-Notify, following the structure of the connection tree inwards from the leaves, we convert the entire process to $a \lightning M$, where M contains all the free endpoints from N and each N_i.

If some P_i is blocked on a free endpoint, reduction becomes blocked as usual. If the process contains multiple zappers, but the connection structure for these zappers is always as in (a), then that merely reduces the number of uses of E-Zap-Kill that are needed. If the process is not

fully-connected, any component with at least zapper reduces to become a single zapper, and the remaining components reduce as usual.

Under this interpretation, $\mathbf{0}$ and $\top$ are assigned dual behaviours—an endpoint of type $\mathbf{0}$ is used to send a kill signal and then wait for confirmation, and an endpoint of type $\top$ is used, dually, to receive a kill signal, and then send confirmation and terminate—and therein lies the problem. The typing derivation for process (b)

$$(\nu x\bar{x})(a \lightning N, x \parallel b \lightning M, \bar{x})$$

could assign $x : \mathbf{0}$ and $\bar{x} : \top$, or vice versa, but by E-Zap-Kill, they will be treated the same, not dually. The type system does not have sufficient structure to ensure the desired dual behaviour.

3.3.2 Synchronous Links and Lazy Forwarders

In this section, we revisit the semantics of the link. In § 2.1.2, we mentioned that the semantics of link does not explicitly forward messages, but rather treats links as a suspended α-renaming. Recall that the reduction rule for link is as follows:

$$(\nu x\bar{x})(x{\leftrightarrow}y \parallel P) \longrightarrow P\{\bar{x}/y\}$$

Let us consider what requirements this places on an implementation. The process P is not required to be ready to act on the endpoint $\bar{x}$, which means that link cannot be implemented in terms of synchronous message passing. In essence, link is asynchronous. For instance, if every channel has an associated message buffer, link could be implemented by asynchronously sending a redirection notice.

There is a tension between the asynchronous semantics of link, and the synchronous semantics of all other actions in CP. This is apparent in the definition of canonical form and its adequacy, which have separate cases for link and for all other actions. There are two options to resolve this tension: we either make link synchronous or all other actions asynchronous. CP is easily adapted to be asynchronous [see, e.g. the treatment of synchronous and asynchronous GV in Fowler, 2019]. In this section, we will consider our options for a synchronous link.

Blocking Link

The easiest approach to making link synchronous is to require that the other process is ready on the relevant endpoint.

$$\frac{\mathrm{ready}(P, \bar{x})}{(\nu x\bar{x})(x{\leftrightarrow}y \parallel P) \longrightarrow P\{\bar{x}/y\}}\ \text{E-Link-Ready}$$

The new rule allows strictly fewer reductions, and as such does not affect the proof of preservation. However, it simplifies the definitions of canonical form and the proof of progress, and strengthens the adequacy of canonical forms.

The definition of canonical form no longer requires condition (1).

Definition 3.99 (Canonical Form)**.** *A process* P *is in* canonical form*, written* canonical(P)*, if* P *is of the form* $\mathscr{C}^n[T_1, \ldots, T_n]$ *(for some* $n \geq 0$*) and (for* $1 \leq i, j \leq n$*) no* P_i *and* P_j *are ready to act on dual endpoints* $\{x, \bar{x}\} \in \mathrm{dn}(\mathscr{C}[\cdot])$.

The proof of progress for HCP (Proposition 3.35) has two cases: either condition (1) fails, or condition (2) fails. The proof of progress for HCP with E-Link-Ready no longer requires the first case, as link reduction is fully covered by the second case.

Proposition 3.100 (Progress)**.** *If* $P \vdash \mathscr{G}$*, then either* P *is in canonical form, or there exists some* Q *such that* $P \longrightarrow Q$.

The adequacy of canonical forms is strengthened by the change. In HCP with E-Link-Ready, "blocked on free endpoints" fully characterises canonical forms.

Corollary 3.101. *If* $P \vdash \Gamma$*, then* canonical(P) $\iff$ blocking(P) $\subseteq$ fn(P).

Identity Expansion

We could implement a synchronous link using the process calculus equivalent of *identity expansion*—the procedure that rewrites proofs in the logic to remove non-atomic uses of the axiom. Under this view, link is a macro which statically computes the link process from the type, using the following expansions:

$$x \overset{A \parr B}{\leftrightarrow} y,\ y \overset{A \otimes B}{\leftrightarrow} x \triangleq x(z).\, y[w].\, (z \overset{A}{\leftrightarrow} w \parallel x \overset{B}{\leftrightarrow} y)$$

$$x \overset{\bot}{\leftrightarrow} y,\ y \overset{1}{\leftrightarrow} x \triangleq x().\, y[].\, 0$$

$$x \overset{A \mathbin{\&} B}{\leftrightarrow} y,\ y \overset{A \oplus B}{\leftrightarrow} x \triangleq x \triangleright \{\mathrm{inl}: y \triangleleft \mathrm{inl}.\, x \overset{A}{\leftrightarrow} y;\ \mathrm{inr}: y \triangleleft \mathrm{inr}.\, x \overset{B}{\leftrightarrow} y\}$$

$$x \overset{\top}{\leftrightarrow} y,\ y \overset{0}{\leftrightarrow} x \triangleq x \not\leftrightarrow y$$

(The type written over the arrow is the type of the left endpoint.)

Identity expansion works when the type is statically known, but it breaks when we add polymorphism, and repairing it requires us to compute the link process dynamically and to keep session types around at runtime.

From the perspective of an implementation, identity expansion also inflates the size of the program, from a single link instruction to a number of instructions equal to the size of the type.

Action Rules

$$
\begin{array}{lll}
x{\leftrightarrow}y \xrightarrow{x()} y[].\,0 & & \textsc{Fwd-Close} \\
x{\leftrightarrow}y \xrightarrow{x(z)} y[w].\,(z{\leftrightarrow}w \parallel x{\leftrightarrow}y) & & \textsc{Fwd-Send} \\
x{\leftrightarrow}y \xrightarrow{x \triangleright \mathrm{inl}} y \triangleleft \mathrm{inl}.\, x{\leftrightarrow}y & & \textsc{Fwd-Select}_1 \\
x{\leftrightarrow}y \xrightarrow{x \triangleright \mathrm{inr}} y \triangleleft \mathrm{inr}.\, x{\leftrightarrow}y & & \textsc{Fwd-Select}_2 \\
x{\leftrightarrow}y \xrightarrow{y()} x[].\,0 & & \textsc{Bwd-Close} \\
x{\leftrightarrow}y \xrightarrow{y(z)} x[w].\,(z{\leftrightarrow}w \parallel x{\leftrightarrow}y) & & \textsc{Bwd-Send} \\
x{\leftrightarrow}y \xrightarrow{y \triangleright \mathrm{inl}} x \triangleleft \mathrm{inl}.\, x{\leftrightarrow}y & & \textsc{Bwd-Select}_1 \\
x{\leftrightarrow}y \xrightarrow{y \triangleright \mathrm{inr}} x \triangleleft \mathrm{inr}.\, x{\leftrightarrow}y & & \textsc{Bwd-Select}_2
\end{array}
$$

Figure 3.5: Label-Transition System for Lazy Forwarders

The Lazy Forwarder

We could implement a synchronous link that performs identity expansion, not based on the type, but based on the messages the link receives. In response to a send action, the link unfolds into a receive action followed by a send action, and the process reduces by E-Send.

$$
\begin{aligned}
(\nu x\bar{x})(x[z].\,P \parallel \bar{x}{\leftrightarrow}y) \triangleq\ & (\nu x\bar{x})(x[z].\,P \parallel \bar{x}(\bar{z}).\,y[w].\,(\bar{z}{\leftrightarrow}w \parallel \bar{x}{\leftrightarrow}y)) \\
\longrightarrow\ & (\nu x\bar{x})(\nu z\bar{z})(P \parallel y[w].\,(\bar{z}{\leftrightarrow}w \parallel \bar{x}{\leftrightarrow}y))
\end{aligned}
$$

The operational semantics of the lazy forwarder are easier to understand from the label-transition system, since the reduction rules fold this two-step behaviour into single rule. The label-transition rules for lazy forwarders are in Figure 3.5.

The lazy forwarder has the same semantics as identity expansion, but works in the presence of polymorphism. Lazy forwarders are synchronous, rather than asynchronous, but they behave differently from blocking links. They are directed, as they only forward received messages. Hence, the following process is stuck:

$$(\nu x\bar{x})(x{\leftrightarrow}y \parallel \bar{x}(z).\,P)$$

This is a direct consequence of the correspondence between the process calculus and the sequent calculus for CLL, since the expansion that sends and then receives—i.e. $y[w].\,x(z).\,(z{\leftrightarrow}w \parallel x{\leftrightarrow}y)$—is not typeable. However, with delayed actions [Kokke et al., 2019a], the process can send and then receive, regardless of the order of the two actions. Hence, under delayed actions lazy forwarders and blocking link may behave the same.

Lazy forwarders are type preserving by the correctness of identity expansion, and satisfy progress as well as the adequacy of canonical

forms. However, the canonical forms are quite different, due to the directionality of lazy forwarders.

Firstly, we must alter the definition of ready, since lazy forwarders are only ready on the endpoint on which they receive, which can be inferred from the types of its endpoints.

Definition 3.102 (Ready)**.** *A process* P *is* ready to act on x, *written* ready(P, x)*, if it is of one of the forms:*

$$\overset{A⅋B}{x\leftrightarrow y}\quad \overset{\bot}{x\leftrightarrow y}\quad \overset{A\&B}{x\leftrightarrow y}\quad \overset{\top}{x\leftrightarrow y}\quad x[y].\,P\quad x[].\,P\quad x \triangleleft \mathit{inl}.\,P\quad x \triangleleft \mathit{inr}.\,P\quad x \not\leftrightarrow N$$
$$\overset{A\otimes B}{y\leftrightarrow x}\quad \overset{\mathbf{1}}{y\leftrightarrow x}\quad \overset{A\oplus B}{y\leftrightarrow x}\quad \overset{\mathbf{0}}{y\leftrightarrow x}\quad x(y).\,P\quad x().\,P\quad x \triangleright \{\mathit{inl}\colon P; \mathit{inr}\colon Q\}$$

A process P *is* ready *if it ready to act on some endpoint.*

The definition of canonical form and the proof of progress simplify in the same manner as for blocking links. However, while the text defining canonical form is the same, the actual canonical forms are quite different, due to the different definitions of *ready*. For instance, the previously mentioned stuck process is in canonical form:

$$(\nu x\bar{x})(\overset{A\otimes B}{x\leftrightarrow y} \parallel \bar{x}(z).\,P)$$

While type annotations are unnecessary at runtime, determining whether or not a lazy forwarder is ready requires case analysis on its type to determine its direction. For the process above, we can infer that the endpoint x has type $A \otimes B$, and hence the link is only ready on y.

The asymmetry of lazy forwarders allows us to simplify the definition of the dependency graph, by removing the special case for link. Links no longer generate undirected edges, but directed arcs which align with their direction:

$$\mathrm{Dep}\left(\overset{A⅋B}{x\leftrightarrow y}\right) = \overrightarrow{xy} \qquad \mathrm{Dep}\left(\overset{A\otimes B}{x\leftrightarrow y}\right) = \overrightarrow{yx}$$

The adequacy of canonical forms is strengthened in the same manner as with blocking links, i.e. “blocked on free endpoints” fully characterises canonical forms.

Lazy forwarders continue to work in the presence of polymorphism, since the direction of the link is determined dynamically, in response to communication.

3.3.3 Variant Types and Guarded Summation

In this section, we generalise select and offer to variant types, as mentioned in § 2.1.5, then decompose the *n*-ary offer into guarded summation, which lets us factor out actions into their own syntactic

sort and which enormously simplifies the reduction and label-transition semantics, as well as the associated metatheory. The decomposition guarantees that summations are guarded using a limited form of focusing, which is a technique from proof theory to control the structure of proofs without reducing the expressivity of the logic [see Andreoli, 1992].

The Variant With Variants

To generalise select and offer to variant types, we remove the restriction that labels must always be drawn from {inl, inr} [following Dardha and Gay, 2018]. Let L range over finite sets of labels, and let l and k range over individual labels (not to be confused with ℓ, which ranges over the entirely unrelated labels used in the label-transition system). We write L, l to mean $L \cup \{l\}$. Labels occur both in processes and in types, and as such appear both in blue and red. They are coloured accordingly, and when they appear by themselves, they are coloured according to whether or not they can be erased. Regardless of colour, labels with the same name refer to the same label.

We replace the binary select and offer actions and the corresponding types with *n*-ary labelled variants:

$$
\begin{array}{rcl}
P, Q, R & ::= & \ldots \\
& \mid & \cancel{x \triangleleft \mathsf{inl}.\, P} \mid \cancel{x \triangleleft \mathsf{inr}.\, P} \\
& \mid & \cancel{x \triangleright \{\mathsf{inl}\colon P; \mathsf{inr}\colon Q\}} \\
& \mid & x \triangleleft l.\, P \\
& \mid & x \triangleright \{l\colon P_l\}_{l \in L}
\end{array}
\qquad
\begin{array}{rcl}
A, B & ::= & \ldots \\
& \mid & \cancel{A \oplus B} \\
& \mid & \cancel{A \mathbin{\&} B} \\
& \mid & \bigoplus\{l\colon A_l\}_{l \in L} \\
& \mid & \mathbin{\&}\{l\colon A_l\}_{l \in L}
\end{array}
$$

The variant offers $x \triangleright \{l\colon P_l\}_{l \in L}$ offers a set of alternatives labelled by the labels from some finite set L, and the variant selection $x \triangleleft l.\, P$ selects the alternative labelled by l. The variant selection and offer types, $\bigoplus\{l\colon A_l\}_{l \in L}$ and $\mathbin{\&}\{l\colon A_l\}_{l \in L}$, respectively, associate each label $l \in L$ with the type of the alternative.

The syntax $\{l\colon P_l\}_{l \in L}$ and $\{l\colon A_l\}_{l \in L}$ denote sets of labelled alternatives. Formally, these are functions from labels to processes and session types, respectively. Note that the occurrences of l and l between the curly braces are not free, but bound by the expressions in the subscripts.

For concrete processes and types, we write the set of alternatives in full, e.g. as $x \triangleright \{l_1\colon P_1; \ldots l_n\colon P_n\}$. In these cases, we omit the subscript, since the set of labels can be inferred.

The choice to treat alternatives as sets means that they can be reordered. For instance, $x \triangleright \{\mathsf{inl}\colon P; \mathsf{inr}\colon Q\}$ and $x \triangleright \{\mathsf{inr}\colon Q; \mathsf{inl}\colon P\}$ are definitionally equal. Alternatively, we could treat the sets of labels as ordered, and treat alternatives as ordered sequences.[4]

[4]Dardha and Gay [2018] use a different construction, where both the label and the

The typing rules and reduction semantics for variant selection and offer are standard generalisations of the binary versions.

$$\text{T-SELECT}\ \frac{P \vdash \Gamma, x : A_l \qquad l \in L}{x \triangleleft l.\, P \vdash \Gamma, x : \bigoplus\{l\colon A_l\}_{l\in L}} \qquad \text{T-OFFER}\ \frac{(P_l \vdash \Gamma, x : A_l)_{l\in L}}{x \triangleright \{l\colon P_l\}_{l\in L} \vdash \Gamma, x : \with\{l\colon A_l\}_{l\in L}}$$

$$(\nu x\bar{x})(x \triangleleft l.\, P \parallel \bar{x} \triangleright \{l\colon Q_l\}_{l\in L}) \longrightarrow (\nu x\bar{x})(P \parallel Q_l) \quad \text{E-SELECT}$$

In the premise of T-OFFER, the notation "$(\dots)_{l\in L}$" means that one typing derivation is required for each label and alternative. Should we want to support type inference, then every variant selection on a free endpoint requires one type annotation for each label not selected, though this is no different from the binary left and right selection actions.

There is an awkwardness between variant types and the absurd offer. The absurd offer should be the nullary variant, which would let us replace $\mathbf{0}$ by $\bigoplus\{\}$, $\top$ by $\with\{\}$, and the absurd offer by $x \triangleright \{\}$. As discussed in § 3.1.6 and § 2.1.6, this works in HCP with the inert semantics and in CP with both the inert semantics and Wadler's commuting conversions, though, in both cases, it complicates the statement of linearity. As discussed in § 2.1.6 and § 3.3.1, zappers require that the names of the discarded endpoints are kept at runtime. Morally, zappers are the nullary offer, and the expected propositions hold, e.g. $\with\{l_1\colon A; l_2\colon \top\}$ is equivalent to A. However, zappers require more information than would follow from the nullary case of the generalisation. To combine variant types with zappers, we must rule out empty variants and retain the additive units.

The metatheory of HCP generalises easily to variant types, and all metatheoretical properties of HCP are preserved. In some cases the proofs simplify, as the variant syntax lets us treat selections uniformly. However, we will not belabour this point, as the generalisation to variant types is not novel, and is principally intended as an introduction to the discussion of guarded summation.

Focusing and Guarded Summation

The π-calculus usually defines the syntax of actions separately from the syntax of processes, and the two are combined by prefixing, i.e. $\pi.P$. This leads to a much simpler theory, as we can treat all actions uniformly. As mentioned in § 2.1.5 and § 3.2.8, this cannot easily be done in CP and HCP, as the offer is a single monolithic construct that combines all labels and alternative processes.

In the π-calculus, alternatives are combined by *summation*, e.g. the process $P + Q$ acts either as P or as Q. To ensure that the choice is not

alternative are indexed by some index i drawn from some set I, e.g. $\{l_i\colon P_i\}_{i\in I}$. They do not specify whether these denote a pair of functions from indices to labels and alternatives, or whether these denote ordered sequences of pairs.

arbitrary, each alternative must be *guarded*, which means that it must be ready to act on some distinct endpoint, and the first alternative to receive a message is selected. For instance, when evaluating the process

$$x[].\ 0 \parallel (x().\ P + y().\ Q)$$

the communication on x selects P and discards Q. Such a process calculus, where the selection is made by the choice of endpoint, is difficult to type in formalisms like CP, where endpoints are linear and channels must be bound by a name restriction. (Summation naturally corresponds to a structural *with*, but the introduction of additive hypersequents complicates name restriction, which must account for the fact that each alternative may use some channel at a different type.)

In this section, we introduce an intermediate system, where alternatives are combined by summation, and each alternative must be guarded by an offer action on the same endpoint, but with a different label. Actions are defined as usual, e.g. as in § 3.2.8, with the exception that selection and offer permit arbitrary labels. (We postpone the discussion of link.)

$$\pi ::= x[y] \mid x(y) \mid x[] \mid x() \mid x \lhd l \mid x \rhd l$$

Processes are defined by name restriction, parallel composition, the terminated process, *prefixing*, and *summation*.

$$P, Q, R ::= (\nu x\tilde{x})P \mid P \parallel Q \mid 0 \mid P + Q \mid \pi.P$$

The process $x \rhd l.\ P$ introduces a unary offer, and offers can be combined by summation. The typing rules for the offer and summation use a limited form of *focusing* [see Andreoli, 1992]. Focusing is a technique from proof theory that can be used to heavily restrict the form of proofs without affecting expressivity, and is often used for more efficient proof search. In our case, focusing is only used to enforce guardedness. We add a new typing judgement, $P \vdash \Gamma \Uparrow x : A$, which remembers what endpoint the process is ready to make an offer on—or, if we focus all actions, ready to act on.[5]

$$\text{T-Offer}_1\ \frac{P \vdash \Gamma, x : A}{x \rhd l.\ P \vdash \Gamma \Uparrow x : \&\{l : A\}} \qquad \text{T-Absurd}\ \frac{N = \mathrm{fn}(\Gamma)}{x \not\downarrow N \vdash \Gamma \Uparrow x : \&\{\}} \qquad \text{T-Focus}\ \frac{P \vdash \Gamma \Uparrow x : A}{P \vdash \Gamma, x : A}$$

$$\text{T-Sum}\ \frac{P \vdash \Gamma \Uparrow x : \&\{l : A_l\}_{l \in L} \qquad Q \vdash \Gamma \Uparrow x : \&\{l : A_l\}_{l \in L'}}{P + Q \vdash \Gamma \Uparrow x : \&\{l : A_l\}_{l \in L \cup L'}}$$

The rule T-Offer$_1$ types the unary offer. It is the unary case of the rule for the variant offer, T-Offer, except that it remembers what endpoint the

[5] The upward arrow in "$P \vdash \Gamma \Uparrow x : A$" comes from Andreoli's sequent focused on an asynchronous proposition, which has a dual focused on a synchronous proposition that uses a downward arrow.

offer is made on. The rule T-Focus allows us to forget this information at any point, though, once forgotten, we cannot recover it. The rule T-Sum types guarded summation, which requires that each alternative is ready to make an offer on the same endpoint.

We extend the structural congruence with the following rules, such that summations are associative and commutative, and the absurd offer is the unit for summation:

$$\begin{array}{lll} P + x \not\downarrow N \equiv P & & \text{SC-SumAbsurd} \\ P + Q \equiv Q + P & & \text{SC-SumComm} \\ P + (Q + R) \equiv (P + Q) + R & & \text{SC-SumAssoc} \end{array}$$

The reduction rule for selection chooses the alternative with the matching label, and discards the other alternatives.

$$(\nu x\bar{x})(x \triangleleft l.\, P \parallel (\bar{x} \triangleright l.\, Q + R)) \longrightarrow (\nu x\bar{x})(P \parallel Q) \quad \text{E-Select}$$

The metatheory of HCP generalises easily to the variant with guarded summation. The proofs of preservation for the structural congruence and reduction must be updated by adding cases for the new rules. The proofs for progress, deadlock freedom, and the adequacy of canonical forms are easily updated. Alternatively, these properties follow from the operational correspondence between HCP with binary choice and HCP with guarded summation. We present the updated proofs of preservation and a sketch for the operational correspondence.

The rules SC-SumAbsurd, SC-SumComm, SC-SumAssoc, and E-Select preserve types.

Lemma 3.103. *If* $P \equiv Q$*, then* $P \vdash \mathscr{G}$ *if and only if* $Q \vdash \mathscr{G}$*.*

Proof. By induction on the derivation of the equivalence $P \equiv Q$.

The cases for reflexivity, symmetry, transitivity, and applications of SC-LinkComm, SC-ParNil, SC-ParComm, SC-ParAssoc, SC-NewComm, and SC-ScopeExt are as in Lemma 3.27. The case for congruence closure follows, similarly to Lemma 3.27, by induction and the injectivity of the type derivation rules.

- Case SC-SumAbsurd.

$$\dfrac{P \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l \in L} \qquad \dfrac{N = \mathrm{fn}(\Gamma)}{x \not\downarrow N \vdash \Gamma \Uparrow x : \&\{\}}}{P \parallel x \not\downarrow N \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l \in L}}$$

$$\Downarrow$$

$$P \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l \in L}$$

- Case SC-SumComm.

$$\frac{P \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L} \qquad Q \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L'}}{P + Q \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L\cup L'}}$$

$$\Downarrow$$

$$\frac{Q \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L'} \qquad P \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L}}{Q + P \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L\cup L'}}$$

- Case SC-SumAssoc.

$$\frac{\dfrac{P_1 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_1} \qquad \dfrac{P_2 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_2} \qquad P_3 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_3}}{P_2 + P_3 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_2\cup L_3}}}{P_1 + (P_2 + P_3) \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_1\cup L_2\cup L_3}}$$

$$\Downarrow$$

$$\frac{\dfrac{P_1 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_1} \qquad P_2 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_2}}{P_1 + P_2 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_1\cup L_2}} \qquad P_3 \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_3}}{P_1 + (P_2 + P_3) \vdash \Gamma \Uparrow x : \&\{l\colon A_l\}_{l\in L_1\cup L_2\cup L_3}}$$

□

Proposition 3.104 (Preservation). *If* $P \vdash \mathcal{G}$ *and* $P \longrightarrow Q$, *then* $Q \vdash \mathcal{G}$.

Proof. By induction on the derivation of the reduction. The cases for E-Send, E-Close, E-Equiv, and E-Cong, are as in Proposition 3.30. The case for E-Link, if included, follows by similarly to Proposition 3.30.

- Case E-Select.

  By inversion, the typing derivation for the summation $\bar{x} \triangleright l.\, Q + R$ is of the form

$$\frac{\dfrac{\dfrac{Q \vdash \Delta, \bar{x} : \overline{A_k}}{\bar{x} \triangleright k.\, Q \vdash \Delta \Uparrow \bar{x} : \&\{k\colon \overline{A_k}\}} \qquad R \vdash \Delta \Uparrow \bar{x} : \&\{l\colon \overline{A_l}\}_{l\in L}}{\bar{x} \triangleright k.\, Q + R \vdash \Delta \Uparrow \bar{x} : \&\{l\colon \overline{A_l}\}_{l\in L,k}}}{\bar{x} \triangleright k.\, Q + R \vdash \Delta, \bar{x} : \&\{l\colon \overline{A_l}\}_{l\in L,k}}$$

  By inversion, the typing derivation for the left-hand side of the reduction is of the form

$$\frac{\dfrac{\dfrac{P \vdash \Gamma, x : A_k}{x \triangleleft k.\, P \vdash \Gamma, x : \bigoplus\{l\colon A_l\}_{l\in L,k}} \qquad \bar{x} \triangleright k.\, Q + R \vdash \Delta, \bar{x} : \&\{l\colon \overline{A_l}\}_{l\in L,k}}{x \triangleleft k.\, P \parallel (\bar{x} \triangleright k.\, Q + R) \vdash \Gamma, x : \bigoplus\{l\colon A_l\}_{l\in L,k} \parallel \Delta, \bar{x} : \&\{l\colon \overline{A_l}\}_{l\in L,k}}}{(\nu x\bar{x})(x \triangleleft k.\, P \parallel (\bar{x} \triangleright k.\, Q + R)) \vdash \Gamma, \Delta}$$

  The result follows as

$$\frac{\dfrac{P \vdash \Gamma, x : A_k \qquad Q \vdash \Delta, \bar{x} : \overline{A_k}}{P \parallel Q \vdash \Gamma, x : A_k \parallel \Delta, \bar{x} : \overline{A_k}}}{(\nu x\bar{x})(P \parallel Q) \vdash \Gamma, \Delta}$$

□

The operational correspondence follows disentanglement, by translating each summation to a sequence of binary choices. The translation on processes proceeds as follows:

1. Take the maximum summation context, which is defined by analogy to the maximum configuration context.
2. Eliminate any superfluous absurd offers, which ensures that any use of SC-SUMABSURD holds reflexively under the translation.
3. Normalise the structure of the summation to right-branching form.
4. Fix an arbitrary order on all labels and reorder the unary offers accordingly, which ensures that any use of SC-SUMCOMM and SC-SUMASSOC holds reflexively under the translation.
5. Translate the summation as a series of binary offers.

The translation on session types proceeds similarly, and the order on labels ensures that the translation of the session types matches the translation of the processes.

The translation on processes preserves types—up to the translation on types—and preserves structural congruence and reduction. However, the correspondence between the reduction semantics is not one-to-one. A process that reduces in one step with E-SELECT requires a number of steps with E-SELECT_1 and E-SELECT_2 that is worst-case linear in the number of alternatives. The worst-case linear increase is a consequence of the right-branching form, and we can improve to a logarithmic increase by translating summations as balanced binary trees.

What About Zappers? The exceptional semantics for the absurd offer, as discussed in § 3.3.1, are compatible with guarded summation. The rule E-ZAP-KILL works as written, and it is important *not* to permit any other actions in the summation.

What About Link? There are no issues with link in HCP with guarded summation, but there is a choice on how to implement link. We can either add link as a process or add it as an action:

- *As a process.* From the perspective of the logic, link should be a process constructor, since link, like name restriction and parallel composition, corresponds to a structural rule of the logic, whereas the actions correspond to logical rules. This prevents us from treating link together with the other actions. In HCP, where link is asynchronous and the other actions are synchronous, we are already required to treat link separately. Hence, for HCP, adding link as a process constructor is a good fit.
- *As an action.* On the other hand, when using a synchronous semantics for link, such as those presented in § 3.3.2, we can treat it

> together with the other actions. Hence, for HCP with a synchronous link and for HCP with asynchronous actions, adding link as an action might be a good fit as well, even if it forces us to write "x↔y. 0".

Processes, Summations, and an Absurd Unit Sangiorgi and Walker [2003] define processes and summations as separate syntactic sorts

$$\begin{array}{rcl} P, Q & ::= & S \mid P \parallel Q \mid (\nu x)P \\ S, R & ::= & 0 \mid \pi.P \mid S + R \end{array}$$

which guarantees that summations are guarded *syntactically*. There is no great need for such separation in HCP, as guardedness is guaranteed by the type system. However, examining Sangiorgi and Walker's definition reveals two interesting things.

Firstly, the rule T-Focus corresponds to the syntactic embedding of summations into processes. If we focus the existing typing rules for actions, the typing judgements divide neatly into separate typing judgements for processes and summations

$P \vdash \mathscr{G}$	process typing
$S \vdash \Gamma \Uparrow x : A$	summation typing

with the rules for prefixing and the rule T-Focus moving back and forth between summation typing and process typing.

Secondly, Sangiorgi and Walker [2003] define 0 as a *summation*. Since summations are processes, it does double duty as both the empty summation and the empty process, i.e. as both the unit for summation and parallel composition. In HCP with guarded summation, the absurd offer $x \lightning N$ is the unit for summation, whereas the terminated process 0 is the unit for parallel composition.

3.3.4 Channel or Endpoint Names?

Should names refer to communication channels or to their endpoints? I already revealed my hand in Chapter 2, where CP's names refer to channel endpoints. However, for CP, the choice is not hugely significant. Case in point, names refer to channels in Wadler's CP.

Under the restrictive view, channel names are a natural choice. Endpoint names are incompatible with the restrictive view, as there is no mechanism to tell us which endpoints are connected to the same channel. We could introduce such a mechanism. We could use co-names, where the endpoint name for sending is computed from the endpoint name for receiving by the overbar function, e.g. if x is the endpoint for receiving, then $\bar{x}$ is the endpoint for sending. We could generalise, and compute

the endpoints for any number of participants in a multiparty session from one special endpoint name. However, such a mechanism invariably requires one special endpoint name from which to compute the others, and is there really any difference between such a special endpoint name and a channel name?

Under the creative view, we have a legitimate choice between channel and endpoint names. Let us consider our options with a few questions.

How to Connect Two Unconnected Parallel Processes?

Consider the following example:

$$a\langle c\rangle.P \parallel b(y).Q$$

With channel names, we must rename one of the channels so that their names coincide, then use a name restriction to bind that name—e.g. rename b to a, then use the name restriction (νa).

$$a\langle c\rangle.P \parallel b(y).Q \quad \text{to} \quad a\langle c\rangle.P \parallel a(y).Q \quad \text{to} \quad (\nu a)(a\langle c\rangle.P \parallel a(y).Q)$$

With endpoint names, we only need to use the name restriction (νab).

$$a\langle c\rangle.P \parallel b(y).Q \quad \text{to} \quad (\nu ab)(a\langle c\rangle.P \parallel b(y).Q)$$

With channel names, we need renaming to connect the two processes—an operation in the meta-language, which mutates the process to boot—but with endpoint names, name restriction internalises the operation of connection into the object language.

What Is in A Name?

With channel names, some form of role annotation is necessary to ensure that the various uses of a channel name are coherent.

As discussed, Lπ annotates session types with their role—whether the corresponding channel is used to send, receive, neither, or both. The typing rule for parallel composition checks that the various roles are coherent—i.e. at most one send and one receive. The typing rule for name restriction checks that all roles are fulfilled—i.e. there is exactly one send and one receive.

In Lindley and Morris' GV, similar role annotations are used. The typing rule for parallel composition checks that the various roles for *one* channel are coherent, and requires all other channels names are unique. It combines one positive and one negative use of one channel name into one locked use of that channel—i.e. for exactly one channel name x, $x : S$ and $x : \overline{S}$ are combined into $x : S^{\#}$. The typing rule for name restriction checks that all roles are fulfilled—i.e. that the channel is locked.

An early version of HCP [Kokke et al., 2019b, with errata] used channel names and did not require an explicit role annotation. However, the typing rule for name restriction does require that each name occur at most twice and with dual types.

With endpoint names, such role annotations are unnecessary, as each endpoint name is associated with a unique role. The type system checks that all endpoints are used with a coherent set of roles when checking the corresponding ν-binder.

Do Tensor and Par Correspond to Send and Receive?

Reader familiar with CP may be confused at the assertion that tensor captures some notion of *disjointness*, rather than sending. Wadler [2012] interprets tensor and par as sending and receiving, respectively.

$$\frac{P \vdash_c \Gamma, y : A \qquad Q \vdash_c \Delta, x : B}{x[y].\,(P \parallel Q) \vdash_c \Gamma, \Delta, x : A \otimes B}\ \text{T-Send} \qquad \frac{P \vdash_c \Gamma, y : A, x : B}{x(y).\,P \vdash_c \Gamma, x : A \parr B}\ \text{T-Recv}$$

Carbone et al. [2016, p. 4-5, *May one invert output and input?*] argue that the inverse interpretation, where tensor is interpreted as receive and par is interpreted as send, is nonsense. The argument is that a process should change its behaviour based on the information it receives, not the information it sends. That is a fair argument, but it misses a crucial aspect of Wadler's interpretations for tensor and par. The send and receive actions, $x[y]$ and $x(y)$, implement a restricted form of delegation, but do not transmit information. They are about plumbing, not what is in the pipes.[6] Hence, the argument does not apply.

The crucial part of the interpretation of tensor/par is not the send/receive actions, but the *disjointness* of $P \parallel Q$ and the *jointness* of P. Given a channel with endpoints $x : A \otimes B$ and $\bar{x} : A \parr B$, the process that holds $\bar{x}$ determines the order in which the sub-sessions A and B are resolved. The process that holds x must guarantee that either order works, and the manner in which CP guarantees this is by forcing the sub-sessions to be handled by entirely disjoint processes.

3.3.5 Deep Inference and Display Calculus

Hypersequents add *stratified* structural connectives. HCLL's hyper-environments may contain environments, but its environments may not contain hyper-environments. (In a sense, HCLL's structures are restricted to conjunctive normal form.) As such, we can view HCLL as a stratified

[6] I'm paraphrasing Robert Atkey, who frequently refers to the function of multiplicative as plumbing or throat-clearing. For a proper quote: "Multiplicatives correspond to matters of communication topology, while the additives will correspond to actual data transfer" [Atkey, 2017].

variant of the formulations of CLL as deep inference calculi [e.g. Flat BV, Guglielmi, 2007, p. 51; or LS, Straßburger, 2002] or display calculi [e.g. Belnap, 1989], where every logical connective has a corresponding structural connective without any such stratification.

3.3.6 Hypersequent Calculus

The hypersequents used in HCLL are similar to those used by Avron [1991]. Both are stratified extensions to the structure of the logic. However, their interpretations are rather different. HCLL's hypersequents are linear, multiplicative, and conjunctive, whereas Avron's hypersequents are unrestricted and "usually disjunctive" [Avron, 1996, p. 4].

3.3.7 Hypersequents, Mix, and Mix$_0$

The rules for hyper-environments resemble those for Mix and Mix$_0$.

$$\frac{\vdash \mathscr{G} \qquad \vdash \mathscr{H}}{\vdash \mathscr{G} \parallel \mathscr{H}}\ (\parallel) \qquad \frac{}{\vdash \circ}\ (\circ) \qquad \frac{\vdash \Gamma \qquad \vdash \Delta}{\vdash \Gamma, \Delta}\ \textsc{Mix} \qquad \frac{}{\vdash \varnothing}\ \textsc{Mix}_0$$

However, HCLL is a conservative extension of CLL, but Mix and Mix$_0$ alter the logic: Mix is logically equivalent to $A \otimes B \multimap A \parr B$, and Mix$_0$ is logically equivalent to $\mathbf{1} \multimap \bot$. The correspondence between branching and tensor is illuminating: Mix converts *branching* (corresponding to $\otimes$) into a *comma* (corresponding to $\parr$), and Mix$_0$ converts *having no premises* (corresponding to $\mathbf{1}$) into the empty environment (corresponding to $\bot$). On the contrary, $(\parallel)$ and $(\circ)$ convert structure of the proof tree to structural connectives with the same logical interpretation.

3.4 Conclusion

In this chapter, I introduced Hypersequent CP with its typing rules, reduction semantics, label-transition semantics, and their metatheory. Hypersequent CP is a variant of CP that uses hypersequents to tighten the correspondence with the π-calculus. I proved *preservation* (Proposition 3.30) and *progress* (Proposition 3.35). I proved that CP's processes are deadlock-free (Corollary 3.42), and that its canonical forms are adequate (Corollary 3.50). I proved that HCP's connection graphs are forests (Proposition 3.52), and that any process can be disentangled into a collection of CP processes (Proposition 3.61). I proved harmony between HCP's reduction and label-transition semantics (Proposition 3.93). I defined a pair of inverse translations from CP to HCP (Definition 3.73) and from HCP to CP (Definition 3.83), and proved that they preserve types (Proposition 3.75 and Proposition 3.84) and give rise to a sound and

complete operational correspondence (Proposition 3.77;Proposition 3.87). Finally, I discussed the relation between HCP and similar logical systems, and introduced three variants of HCP with an exceptional semantics for the absurd offer, with synchronous semantics for links, and with guarded summation.

In the future, it would be interesting to extend HCP with the extensions for CP described in previous work. Some of these extensions are already described in the literature. Montesi and Peressotti [2021] describe a variant of HCP extended with the second-order quantifiers and exponentials, which are interpreted as polymorphism and replication. Qian et al. [2021] describe a variant of HCP extended with the exponentials and DiLL's co-exponentials, which are interpreted as replication and client-server interaction. Other extensions, such as Lindley and Morris' fixed points, have not yet been adapted to HCP.

Furthermore, it would be interesting to investigate a variant of Hypersequent CP with logical and structural connectives that capture sequential composition, such as "before" from Retoré's Pomset logic or "seq" from Guglielmi's BV. Any system with sequential composition necessarily deviates from the π-calculus, which only has trivial sequential composition—prefixing a process with an action. However, general sequential composition is important for programming languages. The paper that introduced session types, Honda [1993], describes a type system for a process calculus with general sequential composition. Honda's type system is unsound, in the sense that well-typed processes may deadlock, and additional syntactic constraints are required to guarantee deadlock freedom. In hindsight, this is unsurprising—Tiu [2006] proved that Guglielmi's "seq" cannot be captured by a standard sequent calculus, and this result is assumed to extend to Retoré's "before" and other systems that capture sequential composition. However, the inference rules of Guglielmi's BV and the proof nets of Retoré's Pomset logic do not easily lend themselves to an interpretation as processes. The decorated sequent calculus for Pomset logic, introduced by Slavnov [2019], is an interesting candidate for further study.

3.5 Omitted Proofs

Lemma 3.40. *If* P *is ready, then* Dep(P) *is essentially acyclic and connected.*

Proof. By case analysis on P.

- Case P is of the form x↔y.

  $V_{\mathrm{Dep}(P)} = \{x, y\}$
  $E_{\mathrm{Dep}(P)} = \{xy\}$
  $A_{\mathrm{Dep}(P)} = \varnothing$

Hence, Dep(P) is essentially acyclic, as there are no arcs, and connected, as the two vertices are connected by an edge.

- Case P is of the form $x[y].\,P'$, $x(y).\,P'$, $x[].\,P'$, $x().\,P'$, $x \triangleleft \mathrm{inl}.\,P'$, $x \triangleleft \mathrm{inr}.\,P'$, or $x \triangleright \{\mathrm{inl}: P_1; \mathrm{inr}: P_2\}$.

  $V_{\mathrm{Dep}(P)} = \mathrm{fn}(P)$
  $E_{\mathrm{Dep}(P)} = \varnothing$
  $A_{\mathrm{Dep}(P)} = \{\overrightarrow{xy} \mid y \in \mathrm{fn}(P) \wedge x \neq y\}$

  Hence, Dep(P) is essentially acyclic, as all arcs are out of x, and connected, as every other vertex is connected to x.

□

Proposition 3.41. *If* $P \vdash \mathscr{G}$, *then the dependency graph* Dep(P) *is essentially acyclic, and there is an isomorphism* f *between the typing environments in* $\mathscr{G}$ *and the components of* Dep(P) *that preserves* fn, *i.e.* $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$.

Proof. The process P is of the form $\mathscr{C}^n[T_1, \ldots, T_n]$ (for some $n \geq 0$). By induction on $\mathscr{C}[\cdot]$ and inversion P and the derivation of $P \vdash \mathscr{G}$.

There are four cases:

- Case $\mathscr{C}[\cdot]$ is of the form $\square$.

  Let G be Dep(P). As $\mathscr{C}[\cdot]$ is maximal, P is ready. By Lemma 3.40, G is essentially acyclic and connected. As P is ready, $\mathscr{G}$ is of the form Γ. Let f be the function $\{\Gamma \mapsto G\}$. By Proposition 3.18, $\mathrm{fn}(\Gamma) = \mathrm{fn}(P)$. By definition, $V_G = \mathrm{fn}(P)$. Hence, $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$.

- Case $\mathscr{C}[\cdot]$ is of the form 0.

  By definition, Dep(P) is the null graph. By inversion on the derivation $P \vdash \mathscr{G}$, $\mathscr{G}$ is of the form $\circ$, i.e. there are no typing environments in $\mathscr{G}$. Let f be the empty function $\varnothing$, which vacuously preserves fn.

- Case $\mathscr{C}[\cdot]$ is of the form $(\nu x\bar{x})\mathscr{C}'[\cdot]$.

  By inversion, P is of the form $(\nu x\bar{x})P'$ such that $(\nu x\bar{x})P' \vdash \mathscr{G}' \parallel \Gamma, \Delta$ and $P' \vdash \mathscr{G}' \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}$.

  Let G be Dep(P) and G' be Dep(P′).

  By definition, $V_G = V_{G'}$, $E_G = E_{G'} \cup \{x\bar{x}\}$, and $A_G = A_{G'}$.

  By induction, G' is essentially acyclic, and there is an isomorphism f' between the components of G' and the typing environments in $\mathscr{G}' \parallel \Gamma, x : A \parallel \Delta, \bar{x} : \overline{A}$ that preserves fn.

  Let C_1 be $f'(\Gamma, x : A)$ and C_2 be $f'(\Delta, \bar{x} : \overline{A})$.

By definition, C_1 and C_2 are disjoint and essentially acyclic.

Let C be the graph formed by connecting C_1 and C_2 with the edge $x\bar{x}$.

By definition, C is connected and a component of G.

By Lemma A.2, C and G are essentially acyclic.

Let f be the function $\{\Gamma, \Delta \mapsto C\} \cup f' \lhd \{\Gamma, x : A, \Delta, \bar{x} : \overline{A}\}$ (where $\lhd$ is domain subtraction, see Definition A.4).

The function f is an isomorphism that preserves fn, by definition (for Γ, Δ) and by induction (for the typing environments in $\mathcal{G}'$).

- Case $\mathcal{C}[\cdot]$ is of the form $\mathcal{C}_1[\cdot] \parallel \mathcal{C}_2[\cdot]$.

  By inversion, P is of the form $P_1 \parallel P_2$ such that $P_1 \parallel P_2 \vdash \mathcal{G}_1 \parallel \mathcal{G}_2$, $P_1 \vdash \mathcal{G}_1$, and $P_2 \vdash \mathcal{G}_2$.

  Let G be $\mathrm{Dep}(P)$, G_1 be $\mathrm{Dep}(P_1)$, and G_2 be $\mathrm{Dep}(P_2)$.

  By definition, $V_G = V_{G_1} \cup V_{G_2}$, $E_G = E_{G_1} \cup E_{G_2}$, and $A_G = A_{G_1} \cup A_{G_2}$.

  By induction, G_1 and G_2 are essentially acyclic, and there are isomorphisms f_1 and f_2 between the typing environments in $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, the components of G_1 and G_2, respectively, that preserve fn.

  By T-Par, G_1 and G_2 are disjoint. Hence, G is essentially acyclic.

  Let f be the function $f_1 \cup f_2$.

  As union preserves the relevant properties of f_1 and f_2, f is an isomorphism between the components of G and the typing environments in $\mathcal{G}$ that preserves fn.

□

Proposition 3.52. *If* $P \vdash \mathcal{G}$, *then* $\mathrm{Con}(P)$ *is a forest, and there is an isomorphism* f *between the typing environments in* $\mathcal{G}$ *and the trees of* $\mathrm{Con}(P)$ *that preserves* fn, *i.e.* $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$.

Proof. The process $P = \mathcal{C}^n[T_1, \ldots, T_n]$ (for some $n \geq 0$).

By induction on $\mathcal{C}[\cdot]$ and inversion P and the derivation of $P \vdash \mathcal{G}$.

There are four cases:

- Case $\mathcal{C}[\cdot]$ is of the form $\square$.

  Let G be $\mathrm{Con}(P)$. As $\mathcal{C}[\cdot]$ is maximal, P is ready. By definition, G is the singleton graph, which is a tree. As P is ready, $\mathcal{G}$ is of the form Γ. Let f be the function $\{\Gamma \mapsto G\}$. By Proposition 3.18, $\mathrm{fn}(\Gamma) = \mathrm{fn}(P)$. By definition, $V_G = \{P\}$. Hence, $\mathrm{fn}(\Gamma) = \mathrm{fn}(f(\Gamma))$.

- Case $\mathscr{C}[\cdot]$ is of the form 0.

  By definition, G is the null graph, which is a forest. By inversion on the derivation $P \vdash \mathscr{G}$, $\mathscr{G}$ is of the form $\circ$, i.e. there are no typing environments in $\mathscr{G}$. Let f be the empty function $\varnothing$, which vacuously preserves fn.

- Case $\mathscr{C}[\cdot]$ is of the form $(\nu x\tilde{x})\mathscr{C}'[\cdot]$.

  By inversion, P is of the form $(\nu x\tilde{x})P'$ such that $(\nu x\tilde{x})P' \vdash \mathscr{G}' \parallel \Gamma, \Delta$ and $P' \vdash \mathscr{G}' \parallel \Gamma, x : A \parallel \Delta, \tilde{x} : \overline{A}$.

  Let G be $\mathrm{Con}(P)$ and G' be $\mathrm{Con}(P')$.

  By Proposition 3.18 and Lemma 3.22, there exist unique $T_i, T_j \in V_{G'}$ such that $x \in \mathrm{fn}(T_i)$ and $\tilde{x} \in \mathrm{fn}(T_j)$

  By definition, $V_G = V_{G'}$ and $\ell_G = \ell_{G'} \cup \{T_i T_j \mapsto (x, \tilde{x})\}$.

  By induction, G' is a forest, and there is an isomorphism f' between the components of G' and the typing environments in $\mathscr{G}' \parallel \Gamma, x : A \parallel \Delta, \tilde{x} : \overline{A}$ that preserves fn.

  Let T_1 be $f'(\Gamma, x : A)$ and T_2 be $f'(\Delta, \tilde{x} : \overline{A})$.

  By definition, T_1 and T_2 are disjoint trees.

  Let T be the graph formed by connecting T_1 and T_2 with the edge $P_i P_j$. By Lemma A.1, T is a tree. Hence, G is a forest.

  Let f be the function $\{\Gamma, \Delta \mapsto C\} \cup f' \mathbin{\triangleleft} \{\Gamma, x : A, \Delta, \tilde{x} : \overline{A}\}$ (where $\triangleleft$ is domain subtraction, see Definition A.4).

  The function f is an isomorphism that preserves fn, by definition (for Γ, Δ) and by induction (for the typing environments in $\mathscr{G}'$).

- Case $\mathscr{C}[\cdot]$ is of the form $\mathscr{C}_1[\cdot] \parallel \mathscr{C}_2[\cdot]$.

  By inversion, P is of the form $P_1 \parallel P_2$ such that $P_1 \parallel P_2 \vdash \mathscr{G}_1 \parallel \mathscr{G}_2$, $P_1 \vdash \mathscr{G}_1$, and $P_2 \vdash \mathscr{G}_2$.

  Let G be $\mathrm{Con}(P)$, G_1 be $\mathrm{Con}(P_1)$, and G_2 be $\mathrm{Con}(P_2)$.

  By definition, $V_G = V_{G_1} \cup V_{G_2}$ and $\ell_G = \ell_{G_1} \cup \ell_{G_2}$.

  By induction, G_1 and G_2 are forests, and there are isomorphisms f_1 and f_2 between the typing environments in $\mathscr{G}_1$ and $\mathscr{G}_2$, respectively, the trees of G_1 and G_2, respectively, that preserve fn.

  By T-Par, G_1 and G_2 are disjoint. Hence, G is a forest.

  Let f be the function $f_1 \cup f_2$.

  As union preserves the relevant properties of f_1 and f_2, f is an isomorphism between the components of G and the typing environments in $\mathscr{G}$ that preserves fn.

□

Lemma 3.56. *If* $P \vdash \mathcal{G} \parallel \Gamma$*, then there exist processes* Q *and* R *such that* $Q \vdash \mathcal{G}$*,* $R \vdash \Gamma$*,* $P \overset{\text{LS}}{\equiv} Q \parallel R$*, and* R *is in right-branching tree form.*

Proof. By Proposition 3.52, Con(P) is a forest and some component T is a tree such that $\text{fn}(\Gamma) = \text{fn}(V_T)$. By induction, guarded by the size of T.

- Case $|V_T| = 1$.

  There is some thread R such that $V_T = \{R\}$, and some evaluation context $\mathcal{E}$ such that $P = \mathcal{E}[R]$. As T is a component of Con(P), $\text{fn}(R) \cap \text{bn}(\mathcal{E}) = \varnothing$. By Corollary 3.26, $P \overset{\text{LS}}{\equiv} \mathcal{E}[0] \parallel R$. Let Q be $\mathcal{E}[0]$. Recall that R is in right-branching form, as it is ready.

- Case $|V_T| > 1$.

  Let T_i be any leaf of T. As T_i is a leaf, exactly one free endpoint in T_i is bound in P. Let us name that endpoint x and its dual $\bar{x}$. There exist some $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{F}_i$, $\mathcal{F}_j$, and P_j such that

  $$\begin{aligned} P &\overset{\text{LS}}{\equiv} \mathcal{E}_1[(\nu x\bar{x})(\mathcal{E}_2[\mathcal{F}_i[T_i] \parallel \mathcal{F}_j[P_j]])] && \langle\text{by Corollary 3.23}\rangle \\ &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(\mathcal{E}_1[\mathcal{E}_2[\mathcal{F}_i[T_i] \parallel \mathcal{F}_j[P_j]]]) && \langle\text{by Lemma 3.24}\rangle \\ &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(\mathcal{E}_1[\mathcal{E}_2[\mathcal{F}_i[T_i \parallel \mathcal{F}_j[P_j]]]]) && \langle\text{by Lemma 3.25}\rangle \\ &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(T_i \parallel \mathcal{E}_1[\mathcal{E}_2[\mathcal{F}_i[\mathcal{F}_j[P_j]]]]) && \langle\text{by Lemma 3.25}\rangle \end{aligned}$$

  Let P' be $\mathcal{E}_1[\mathcal{E}_2[\mathcal{F}_i[\mathcal{F}_j[P_j]]]]$. By Lemma 3.27, P' is well-typed.

  Let T' be the tree formed by deleting the vertex T_i and its only edge from T. Clearly, $|V_T| = |V_{T'}| + 1$. Every thread in P is preserved in P', except T_i. Therefore, the connection graph Con(P') is the forest formed by replacing T with T'.

  By induction with P', there exist processes Q and R' such that $P' \overset{\text{LS}}{\equiv} Q \parallel R'$ and R' is in right-branching form. Therefore,

  $$\begin{aligned} P &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(T_i \parallel \mathcal{E}_1[\mathcal{E}_2[\mathcal{F}_i[\mathcal{F}_j[P_j]]]]) && \langle\text{see above}\rangle \\ &\overset{\text{LS}}{\equiv} (\nu x\bar{x})(T_i \parallel (Q \parallel R')) && \langle\text{by induction}\rangle \\ &\overset{\text{LS}}{\equiv} Q \parallel (\nu x\bar{x})(T_i \parallel R') && \langle\text{by Lemma 3.25}\rangle \end{aligned}$$

  Let R be $(\nu x\bar{x})(T_i \parallel R')$. As R' is in right-branching tree form, R is in right-branching tree form.

  It remains to show that the processes Q and R have the correct types.

  For some Γ_i, Γ_j, and A, $T_i \vdash \Gamma_i, x : A$ and $P_j \vdash \Gamma_j, \bar{x} : \overline{A}$.

  Since T_i is a leaf, the evaluation contexts $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{F}_i$ must preserve the contents of its typing environment Γ_i. In essence, we can assign $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{F}_i$ type schemas, rather than types. For some $\mathcal{G}'$, $\mathcal{G}'_i$, $\mathcal{G}'_j$, Γ'',

Γ'', and Γ' the evaluation contexts $\mathscr{E}_1$, $\mathscr{E}_2$, and $\mathscr{F}_i$ have the following type schemas, and $\mathscr{F}_j$ has the following type.

$$\begin{array}{lll}
\forall \Delta.\ \mathscr{E}_1 \vdash \mathscr{G}' \parallel \Gamma'', \Delta & \to \mathscr{G} \parallel \Gamma', \Delta & \text{or} \\
\forall \Delta.\ \mathscr{E}_1 \vdash \mathscr{G}' \parallel \Gamma'', \bar{x} : \overline{A} \parallel \Delta & \to \mathscr{G} \parallel \Gamma', \bar{x} : \overline{A} \parallel \Delta & \\
\forall \Delta.\ \mathscr{E}_2 \vdash \mathscr{G}'_i \parallel \mathscr{G}'_j \parallel \Gamma''', \bar{x} : \overline{A} \parallel \Delta & \to \mathscr{G}' \parallel \Gamma'', \bar{x} : \overline{A} \parallel \Delta & \\
\forall \Delta.\ \mathscr{F}_i \vdash \Delta & \to \mathscr{G}'_i \parallel \Delta & \\
\quad \mathscr{F}_j \vdash \Gamma_j, \bar{x} : \overline{A} & \to \mathscr{G}'_j \parallel \Gamma''', \bar{x} : \overline{A} &
\end{array}$$

Hence, $\Gamma = \Gamma', \Gamma_i$ and $P' \vdash \mathscr{G} \parallel \Gamma', \bar{x} : \overline{A} \parallel \Gamma_i, x : A$.

By induction, $Q \vdash \mathscr{G}$ and $R' \vdash \Gamma', \bar{x} : \overline{A}$. Hence, $R \vdash \Gamma$.

□

Proposition 3.57. *If $P \vdash \mathscr{G}$, then there exists a process Q, such that $P \overset{\text{ls}}{\equiv} Q$, and Q is in right-branching forest form, and there is an isomorphism f between the processes in right-branching tree form in Q and the typing environments in $\mathscr{G}$ that preserves* fn*, i.e.* $\text{fn}(\Gamma) = \text{fn}(f(\Gamma))$.

Proof. By induction on the hyper-environment $\mathscr{G}$.

- Case $\mathscr{G}$ is of the form $\circ$.

  By Lemma 3.55, $P \overset{\text{ls}}{\equiv} 0$. Let Q be 0. The result follows.

- Case $\mathscr{G}$ is of the form $\mathscr{G} \parallel \Gamma$ (reusing $\mathscr{G}$).

  By Lemma 3.56, there exist processes P' and R such that $P' \vdash \mathscr{G}$, $R \vdash \Gamma$, $P \overset{\text{ls}}{\equiv} P' \parallel R$, and R is in right-branching tree form.

  By induction, there exists a process Q' such that $P' \overset{\text{ls}}{\equiv} Q'$ and Q' is in right-branching forest form. There are two cases:

  - If $Q' = 0$, let Q be R. By inversion, $\mathscr{G} = \circ$. The result follows, as, by SC-ParNil, $P \overset{\text{ls}}{\equiv} R$, and R is in right-branching forest form.
  - Otherwise, let Q be $Q' \parallel R$. The result follows, as $P \overset{\text{ls}}{\equiv} Q$ and Q is in right-branching forest form.

□

Lemma 3.89. *If $P \overset{\ell}{\longrightarrow\!\!\!\!\!\equiv} Q$, then $P \overset{\ell}{\longrightarrow} \equiv Q$.*

Proof. Unfolding the compositions of the structural congruence and the transition relations, the goal is as follows:

$$P \equiv P' \wedge P' \overset{\ell}{\to} Q \implies P \overset{\ell}{\to} Q' \wedge Q' \equiv Q$$

I refer to the subgoals $P \overset{\ell}{\to} Q'$ and $Q' \equiv Q$ as results (1) and (2), respectively.

By induction on the derivation of the structural congruence $P \equiv Q$ and inversion on the transition $P \xrightarrow{\ell} P'$. The case for reflexivity follows immediately. The cases for symmetric use of the rules of structural congruence follow by an analogy to the cases below. The case for transitivity follows by induction. The case for congruence follows by induction on the transition.

- Case SC-LinkComm.

  The structural congruence is of the form:

  $$x{\leftrightarrow}y \equiv y{\leftrightarrow}x$$

  By inversion on the transition:

  - Subcase Act-Link$_1$. Result (1) follows by Act-Link$_2$.
  - Subcase Act-Link$_2$. Result (1) follows by Act-Link$_1$.

  Result (2) follows by reflexivity.

- Case SC-ParNil.

  The structural congruence is of the form (reusing P):

  $$P \parallel 0 \equiv P$$

  Result (1) follows by Str-Cong with $\square \parallel 0$.

  Result (2) follows by SC-ParNil.

- Case SC-ParComm.

  The structural congruence is of the form (reusing P and Q):

  $$P \parallel Q \equiv Q \parallel P$$

  By inversion on the transition:

  - Subcase Str-Par.

    Result (1) follows as $\pi \parallel \bar{\pi}$ is unordered.

  - Subcase Str-Cong with $\mathscr{E} \parallel Q$.

    Result (1) follows by Str-Cong with $Q \parallel \mathscr{E}$.

  - Subcase Str-Cong with $P \parallel \mathscr{E}$.

    Result (1) follows by Str-Cong with $\mathscr{E} \parallel P$.

  Result (2) follows by SC-ParComm.

- Case SC-ParAssoc.

  The structural congruence is of the form (reusing P and Q):

  $$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

  By inversion on the transition:

- Subcase Str-Par.

  Result (1) follows as $\pi \parallel \bar{\pi}$ is unordered.

- Subcase Str-Cong with $P \parallel \Box$ then Str-Par.

  Result (1) follows as $\pi \parallel \bar{\pi}$ is unordered.

- Subcase Str-Cong with $\mathscr{E} \parallel (Q \parallel R)$.

  Result (1) follows by Str-Cong with $(\mathscr{E} \parallel Q) \parallel R$.

- Subcase Str-Cong with $P \parallel (\mathscr{E} \parallel R)$.

  Result (1) follows by Str-Cong with $(P \parallel \mathscr{E}) \parallel R$.

- Subcase Str-Cong with $P \parallel (Q \parallel \mathscr{E})$.

  Result (1) follows by Str-Cong with $(P \parallel Q) \parallel \mathscr{E}$.

Result (2) follows by SC-ParAssoc.

- Case SC-NewComm.

  The structural congruence is of the form (reusing P):

  $$(\nu x\bar{x})P \equiv (\nu \bar{x}x)P$$

  By inversion on the transition. In the cases for Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, Tau-Select-Offer$_2$, result (1) follows as $\pi \parallel \bar{\pi}$ is unordered.

  In the cases for Tau-Link and Tau-Close-Wait, result (2) follows by reflexivity. In the cases for Tau-Send-Recv, Tau-Select-Offer$_1$, Tau-Select-Offer$_2$, result (2) follows by SC-NewComm.

- Case SC-NewAssoc.

  The structural congruence is of the form (reusing P):

  $$(\nu x\bar{x})(\nu y\bar{y})P \equiv (\nu y\bar{y})(\nu x\bar{x})P$$

  By inversion on the transition:

  - Subcases Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, or Tau-Select-Offer$_2$ then Str-Cong with $(\nu y\bar{y})\Box$.

    Result (1) follows by Str-Cong with $(\nu y\bar{y})\Box$ then Tau-Link, Tau-Send-Recv, Tau-Close-Wait, Tau-Select-Offer$_1$, or Tau-Select-Offer$_2$.

    In the cases for Tau-Link and Tau-Close-Wait, result (2) follows by reflexivity. In the cases for Tau-Send-Recv, Tau-Select-Offer$_1$, Tau-Select-Offer$_2$, result (2) follows by SC-NewAssoc.

- Subcases STR-CONG with $(\nu x\bar{x})\square$ then TAU-LINK, TAU-SEND-RECV, TAU-CLOSE-WAIT, TAU-SELECT-OFFER$_1$, or TAU-SELECT-OFFER$_2$.

  Result (1) follows by TAU-LINK, TAU-SEND-RECV, TAU-CLOSE-WAIT, TAU-SELECT-OFFER$_1$, or TAU-SELECT-OFFER$_2$ then STR-CONG with $(\nu x\bar{x})\square$.

  In the cases for TAU-LINK and TAU-CLOSE-WAIT, result (2) follows by reflexivity. In the cases for TAU-SEND-RECV, TAU-SELECT-OFFER$_1$, TAU-SELECT-OFFER$_2$, result (2) follows by SC-NEWASSOC.

- Subcase STR-CONG with $(\nu x\bar{x})(\nu y\bar{y})\square$.

  Result (1) follows by STR-CONG with $(\nu y\bar{y})(\nu x\bar{x})\square$.

  Result (2) follows by SC-NEWASSOC.

• Case SC-SCOPEEXT.

The structural congruence is of the form (reusing P and Q):

$$(\nu x\bar{x})(P \parallel Q) \equiv P \parallel (\nu x\bar{x})Q$$

By inversion on the transition:

- Subcases TAU-LINK, TAU-SEND-RECV, TAU-CLOSE-WAIT, TAU-SELECT-OFFER$_1$, or TAU-SELECT-OFFER$_2$ then STR-PAR or STR-CONG with $\mathscr{E} \parallel Q$.

  Impossible. The side condition for SC-SCOPEEXT requires $x, \bar{x} \notin \mathrm{fn}(P)$, but either $x \in \mathrm{fn}(P)$ or $\bar{x} \in \mathrm{fn}(P)$ must hold.

- Subcases TAU-LINK, TAU-SEND-RECV, TAU-CLOSE-WAIT, TAU-SELECT-OFFER$_1$, or TAU-SELECT-OFFER$_2$ then STR-CONG with $P \parallel \mathscr{E}$. Result (1) follows by STR-CONG with $P \parallel \square$, then TAU-LINK, TAU-SEND-RECV, TAU-CLOSE-WAIT, TAU-SELECT-OFFER$_1$, or TAU-SELECT-OFFER$_2$, then STR-CONG with $\mathscr{E}$.

  In the cases for TAU-LINK and TAU-CLOSE-WAIT, result (2) follows by reflexivity. In the cases for TAU-SEND-RECV, TAU-SELECT-OFFER$_1$, TAU-SELECT-OFFER$_2$, result (2) follows by SC-NEWASSOC.

- Subcases STR-CONG with $(\nu x\bar{x})\square$ then STR-PAR.

  Result (1) follows by STR-PAR then STR-CONG with $(\nu x\bar{x})\square$.

  Result (2) follows by SC-SCOPEEXT.

- Subcases STR-CONG with $(\nu x\bar{x})(\mathscr{E} \parallel Q)$

  Result (1) follows by STR-CONG with $\mathscr{E} \parallel (\nu x\bar{x})Q$

  Result (2) follows by SC-SCOPEEXT.

- Subcases STR-CONG with $(\nu x\bar{x})(P \parallel \mathscr{E})$.

Result (1) follows by Str-Cong with $P \parallel (\nu x \bar{x})\mathscr{E}$.

Result (2) follows by SC-ScopeExt.

□

Chapter 4

Hypersequent Good Variation

This chapter presents Hypersequent GV (HGV), concurrent λ-calculus with session-typed concurrency primitives, which has a tight correspondence to Hypersequent CP.

Hypersequent GV was first introduced by Fowler et al. [2021], as a variant of GV that uses hypersequents for its runtime type system. While CP has not undergone many significant changes since its publication by Wadler [2012], GV's history is much more storied:

- Good Variation was first published under that name by Wadler [2012], who adapted the LAST calculus of Gay and Vasconcelos [2010] to fit a correspondence with CP. However, Wadler's GV has neither an operational semantics—except implicitly, by its translation to CP—nor an account of polymorphism or replication.

  This work has an error in its translation from GV to CP, which does not preserve typing in the case for session termination (Theorem 3).

- Lindley and Morris [2014][1] extend GV with operations for forwarding, polymorphism, and servers and clients, rectify the error in Wadler's translation from GV to CP that breaks typing, and describe the first translation from CP to GV.

  This work does not give an operational semantics for GV, and its translation from GV to CP does not preserve reduction.

- Lindley and Morris [2015] give an operational semantics for GV, drop polymorphism and servers and clients, refactor GV so that the concurrency primitives are constant functions rather than term constructors, and switch to an encoding of the choice operators following Dardha et al. [2017].

[1] Lindley and Morris [2014] name their system "Harmonious GV" as the operations in the their version of GV restore the balance between and CP. Later publications in the same line of work refer to their system as GV.

This work has an error in its operational semantics, which does not satisfy the diamond property in the case for link (Theorem 13), and in its translation from GV to CP, which does not preserve the reductions for products and sums[2] (Lemma 21 and Theorems 22 and 23).

- Lindley and Morris [2016b] present μCP and μGV, which adds fixed points to both GV and CP. Orthogonally, this work also rectifies the error in the semantics for link, and adds the direct unit "⊤" and the direct product "&" [Girard and Lafont, 1987].

  This work has the same error in its translation from GV to CP as Lindley and Morris [2015], which does not preserve the reductions for products and sums (Theorem 16).

- Lindley and Morris [2017] present FST, which adds record & variant types with row polymorphism, an account of subkinding for linear and unrestricted types, asynchrony, and access points, and drops link, weak explicit substitution, and the correspondence with CP. This work does not present any of the metatheory of FST.

- Fowler et al. [2019] and Fowler [2019] present EGV, which adds exceptions and handlers to GV. This work also drops link, weak explicit substitution, and the correspondence with CP.

- Fowler et al. [2021] present Hypersequent GV, which rectifies a design flaw that complicates all previous metatheory of GV by dropping lock typing in favour of hypersequents. Orthogonally, this work also rectifies the error in the translation from GV to CP, and presents the first such translation from that both preserves and reflects reduction. (The extended version of this paper is presented in Chapter 4.)

In one sense, Hypersequent GV relates to GV as Hypersequent CP relates to CP. It uses hypersequents to separate name restriction from parallel composition in its configuration typing. In doing so, it removes lock typing and makes the structural congruence type preserving, which significantly simplifies the metatheory of GV.

In another sense, Hypersequent GV is simply GV. Recall that GV separates its language into a static fragment and a runtime fragment, and the static fragment—the user-facing programming language—is unchanged from

[2]Lindley and Morris [2015] use weak explicit substitution in an effort to obtain a translation from GV to CP that preserves reduction, which works in principle, as the translation preserves β-reduction. Unfortunately, their implementation of weak explicit substitution only suspends substitution on λ-abstraction. Hence, the translation does not preserve reductions that involve the other binders, i.e. those for products and sums. Consequently, Lemma 21, Theorem 22, and Theorem 23 fail to hold. Presumably, this is easily solved by either suspending substitution on all binders, or by encoding the eliminators for products and sums as constants and making λ-abstraction the only binder.

GV. Hypersequent CP generalises CP's process structure from trees to forests, and likewise Hypersequent GV generalises GV's configuration structures from trees to forests. However, reduction from a single term—a single user program—only ever reaches tree-structured configurations. That does not mean that we could not use the framework of GV to study configurations arising from multiple user programs—we could, although, since they are disconnected, this is unlikely to provide us any novel insights.

The bulk of this chapter consists of the paper *Separating Sessions Smoothly* by Fowler et al. [2023], hereafter referred to as Paper I. References made from the main body of this thesis into Paper I will be prefixed by an "I", e.g. "Theorem I.3.20". This chapter proceeds as follows:

- In § 4.1, we provide a legend and an errata for Paper I.
- In § 4.2, we present Hypersequent GV and its metatheory, together with translations to and from GV, and an operational correspondence with Hypersequent CP. This section consists entirely of Paper I, and proceeds as follows:
  - In § I.2, we discuss the complications in GV's metatheory that arise from the use of lock typing.
  - In § I.3, we present HGV.
  - In § I.3.1, we present the metatheory for HGV.

    Notably, we prove *preservation* (Theorem I.3.3), the *tree-structure* of connections in configurations (Theorem I.3.14), *global progress* (Theorem I.3.20), the *diamond property* (Theorem I.3.21), and *termination* (Theorem I.3.22).
  - In § I.4, we present the relation between HGV and GV.

    Notably, we prove that any GV configuration is typeable in HGV (Theorem I.4.3), and that any HGV configuration can be rewritten by structural congruence to obtain a configuration that is typeable in GV (Corollary I.4.7).
  - In § I.5, we present the relation between HGV and HCP.

    Notably, we define fine-grain call-by-value HGV (HGV$*$, § I.5, under "HGV$*$"), define a naive translation from HGV to HGV$*$ (§ I.5, Definition I.5.8) and define a translation from HGV$*$ to HCP (§ I.5, Figure I.8), and prove that the latter translation preserves types (§ I.5, Lemma I.5.9) and is a sound and complete operational correspondence (§ I.5, Theorem I.5.11). Finally, we define a translation from HCP to HGV, by composing existing translations (§ I.5, under "Translating HCP to HGV").
  - In § I.6, we present extensions of HGV.

– In § I.7, we discuss the possibility of using hyper-environments in term typing.

– In § I.8, we discuss the related work.

4.1 Legend and Errata

The conventions and terminology in Paper I are different from those used in the rest of this thesis.

- The terms and types of *both* Hypersequent CP and Hypersequent GV are printed in red and blue, respectively, and are rendered in a font with serif. The keywords in HGV's syntax are bolded.
- The names for the rules of structural congruence, reduction, and the type system differ slightly from those used for HCP in Chapter 3.
- Instead of including a single rule for congruence under evaluation contexts, the paper includes separate congruence rules for name restriction and parallel composition (E-Res and E-Par).
- Instead of the *connection graph*, the paper uses the "abstract process structure" (§ I.3.1, Definition I.3.9), which is derived from a hyper-environment and co-name set, as opposed to the configuration.
- Instead of *ready thread*, the paper uses the term "blocked thread" (§ I.3.1, Definition I.3.16.)
- The duality for HCP is written as $A^{\perp}$ rather than $\overline{A}$ (see Figure I.6).
- The lts for HCP (see Figure I.7) combines a number of action rules into Act-Pref, renames Act-Offer$_1$ and Act-Offer$_2$ to Act-Off-Inl and Act-Off-Inr, respectively, makes the distinction between α-transition and β-transition explicit in the rules for τ-transition, rather than a post-facto restriction, replaces the congruence rule Str-Cong with individual congruence rules for name restriction and parallel composition (Str-Res, Str-Par$_1$, and Str-Par$_2$), and renames Str-Par to Str-Syn.

To the best of my knowledge, there are no significant errors in Paper I, but there are a small number of typesetting errors:

- The definition of internal and external choice (§ I.3, under "Choice") define $S \oplus S'$ and $S \,\&\, S'$ in terms of S_1 and S_2, rather than S and S'. The correct definitions are:

$$S \oplus S' \triangleq !\mathbf{1}.!(\overline{S} + \overline{S'}).\mathrm{end}_! \qquad S \,\&\, S' \triangleq ?\mathbf{1}.?(S + S').\mathrm{end}_?$$

- The defintion of an *abstract process structure* (§ I.3.1, Definition I.3.9) uses the function envs without prior definition. The function maps hyper-environments to sets of environments, and may be defined as $\mathrm{envs}(\Gamma_1 \parallel \ldots \parallel \Gamma_n) \triangleq \{\Gamma_1, \ldots, \Gamma_n\}$.

- The definition of a *ground configuration* (§ I.3.1, Definition I.3.19) states that a configuration C is a ground configuration if $\cdot \vdash C : T$, amongst other conditions. This is syntactically ill-formed, as T is not a configuration type, and should be $\cdot \vdash C : \bullet T$.

- One of TG-Connect$_1$ and TG-Connect$_2$ can be elided from GV's configuration typing rules (Figure I.5) without compromising type preservation of reduction, since reduction is only defined on bound endpoints and one can always dualise the type of the bound endpoints. Having both comes at a cost, as parallel compositions no longer have unique typing derivations. Fortunately, the problem is easily addressed by eliding either TG-Connect$_1$ or TG-Connect$_2$.

  The rules TG-Connect$_1$ and TG-Connect$_2$ were mistakenly reproduced from Fowler [2019]. In Fowler's GV, both rules are needed, since reduction is allowed outside of the scope of a name restriction, and therefore, it is not possible to dualise the type of the locked channel without breaking preservation. However, the version of GV presented in this chapter only permits reduction under a name restriction.

- The proof of *global progress* (§ I.3.1, Theorem I.3.20) states that any ground configuration that cannot be reduced can be written as:

$$(\nu x_1 y_1)(\circ\, \mathscr{A}_1 \parallel \ldots \parallel (\nu x_n y_n)(\circ\, \mathscr{A}_n \parallel \bullet V)\ldots)$$

  This is syntactically ill formed, as $\mathscr{A}$ denotes an auxiliary thread, which already includes the thread flag $\circ$. The correct statement is:

$$(\nu x_1 y_1)(\mathscr{A}_1 \parallel \ldots \parallel (\nu x_n y_n)(\mathscr{A}_n \parallel \bullet V)\ldots)$$

- The statement of the *diamond property* (§ I.3.1, Theorem I.3.21) is:

$$\text{If } \mathscr{G} \vdash \mathscr{C} : T,\ \mathscr{C} \longrightarrow \mathscr{D},\ \mathscr{C} \longrightarrow \mathscr{D}', \text{ then } \mathscr{D} \equiv \mathscr{D}'.$$

  This does not hold—and, arguably, is not the diamond property—since it does not permit the result to perform any reductions. Hence, if the reductions $\mathscr{C} \longrightarrow \mathscr{D}$ and $\mathscr{C} \longrightarrow \mathscr{D}'$ eliminate different redexes, there is no leeway to eliminate the redex targeted by $\mathscr{C} \longrightarrow \mathscr{D}$ in $\mathscr{D}'$ and vice versa. The correct statement is:

  > If $\mathscr{G} \vdash \mathscr{C} : T$ and $\mathscr{C} \longrightarrow \mathscr{D}_1$ and $\mathscr{C} \longrightarrow \mathscr{D}_2$, then either $\mathscr{D}_1 \equiv \mathscr{D}_2$ or there exists some $\mathscr{D}_3$ such that $\mathscr{D}_1 \longrightarrow \mathscr{D}_3$ and $\mathscr{D}_2 \longrightarrow \mathscr{D}_3$.

- The definition of *flattening* (§ I.4, Definition I.4.1) translates the empty hyper-environment to the empty hyper-environment, i.e. $\downarrow \varnothing = \varnothing$.
  This is incorrect, as the codomain of flattening is HGV *typing environments*, and the empty typing environment is denoted by $\cdot$. The correct definition is $\downarrow \varnothing = \cdot$.

4.2 Paper I: Separating Sessions Smoothly

This section contains the paper with the same title, written in collaboration with Simon Fowler, Ornela Dardha, Sam Lindley, and J. Garrett Morris, which was originally published in the journal Logical Methods in Computer Science, Volume 19, Issue 3, 2023, and is an extended journal version of the paper with the same title and authors originally published in the proceedings for the 32nd International Conference on Concurrency Theory (CONCUR 2021) as part of the Leibniz International Proceedings in Informatics (LIPIcs) series.

J. Garrett Morris acted as my second Ph.D. supervisor from 2016 to 2017, and Sam Lindley acted as my second Ph.D. supervisor from 2018 to 2024.

The work presented in the paper was conceived of by all the authors. I co-developed Hypersequent GV and was primarily responsible for the correspondence between Hypersequent GV and Hypersequent CP.

Logical Methods in Computer Science
Volume 19, Issue 3, 2023, pp. 3:1–3:52
https://lmcs.episciences.org/

Submitted Apr. 19, 2022
Published Jul. 12, 2023

SEPARATING SESSIONS SMOOTHLY

SIMON FOWLER [a], WEN KOKKE [b], ORNELA DARDHA [a], SAM LINDLEY [c], AND J. GARRETT MORRIS [d]

[a] University of Glasgow, UK
e-mail address: simon.fowler@glasgow.ac.uk, ornela.dardha@glasgow.ac.uk

[b] University of Strathclyde, UK
e-mail address: wen.kokke@strath.ac.uk

[c] The University of Edinburgh, UK
e-mail address: sam.lindley@ed.ac.uk

[d] The University of Iowa, USA
e-mail address: garrett-morris@uiowa.edu

Abstract. This paper introduces Hypersequent GV (HGV), a modular and extensible core calculus for functional programming with session types that enjoys deadlock freedom, confluence, and strong normalisation. HGV exploits hyper-environments, which are collections of type environments, to ensure that structural congruence is type preserving. As a consequence we obtain an operational correspondence between HGV and HCP—a process calculus based on hypersequents and in a propositions-as-types correspondence with classical linear logic (CLL). Our translations from HGV to HCP and vice-versa both preserve and reflect reduction. HGV scales smoothly to support Girard's Mix rule, a crucial ingredient for channel forwarding and exceptions.

1. Introduction

Session types [Hon93, THK94, HVK98] are types used to model and verify communication protocols in concurrent and distributed systems: just as data types rule out dividing an integer by a string, session types rule out sending an unexpected message. Session types originated in process calculi, but there is a gap between process calculi, which model the evolving state of concurrent systems, and the descriptions of these systems in mainstream programming languages. This paper addresses two foundations for session types: (1) a session-typed concurrent lambda calculus called GV [LM15], intended to be a modular and extensible basis for functional programming languages with session types; and, (2) a session-typed process calculus called CP [Wad14], with a propositions-as-types correspondence to classical linear logic (CLL) [Gir87].

Processes in CP correspond exactly to proofs in CLL and deadlock freedom follows from cut-elimination for CLL. However, while CP is strongly tied to CLL, at the same time it departs from the π-calculus. Independent π-calculus features can only appear in combination in CP: CP combines name restriction with parallel composition ($(\nu x)(P \parallel Q)$), corresponding

DOI:10.46298/LMCS-19(3:3)2023

to CLL's cut rule, and combines sending (of bound names only) with parallel composition ($x[y].(P \parallel Q)$), corresponding to CLL's tensor rule. This results in a proliferation of process constructors and prevents the use of standard techniques from concurrency theory, such as labelled-transition semantics and bisimulation, since the expected transitions give rise to ill-typed terms. For example, we cannot write the expected transition rule for output, $x[y].(P \parallel Q) \xrightarrow{x[y]} P \parallel Q$, since $P \parallel Q$ is not a valid CP process. A similar issue arises when attempting to design a synchronisation transition rule for bound output (see [KMP19b] for a detailed discussion). Inspired by Carbone *et al.* [CMS18] who use hypersequents [Avr91] to give a logical grounding to choreographic programming languages [Mon13], Hypersequent CP (HCP) [KMP19a, KMP19b, MP18] restores the independence of these features by factoring out parallel composition into a standalone construct while retaining the close correspondence with CLL proofs. HCP typing reasons about collections of processes using collections of type environments (or *hyper-environments*).

GV extends linear λ-calculus with constants for session-typed communication. Following Gay and Vasconcelos [GV10], Lindley and Morris [LM15] describe GV's semantics by combining a reduction relation on single terms, following standard λ-calculus rules, and a reduction relation on concurrent configurations of terms, following standard π-calculus rules. They give a semantic characterisation of deadlocked processes, an extrinsic [Rey00] type system for configurations, and show that well-typed configurations are deadlock-free. There is, however, a large fly in this otherwise smooth ointment: GV's process equivalence does not preserve typing. As a result, it is not enough for Lindley and Morris to show progress and preservation for well-typed configurations; instead, they must show progress and preservation for *all* configurations *equivalent to* well-typed configurations. This not only complicates the metatheory of GV, but the burden is inherited by any effort to build on GV's account of concurrency [FLMD19].

In this paper, we show that using hyper-environments in the typing of configurations enables a metatheory for GV that, compared to that of Lindley and Morris, is simpler, is more general, and as a result is easier to use and easier to extend. Hypersequent GV (HGV) repairs the treatment of process equivalence—equivalent configurations are equivalently typeable—and avoids the need for formal gimmickry connecting name restriction and parallel composition. HGV admits standard semantic techniques for concurrent programs: we use bisimulation to show that our translations both preserve *and* reflect reduction, whereas Lindley and Morris resort to weak explicit substitutions [LM99] and only show that their translations between GV and CP preserve reduction. HGV is also more easily extensible: we outline three examples, including showing that HGV naturally extends to disconnected sets of communication processes, without any change to the proof of deadlock freedom, and that it serves as a simpler foundation for existing work on exceptions in GV [FLMD19].

Contributions. The paper contributes the following:

- Section 3 introduces Hypersequent GV (HGV), a modular and extensible core calculus for functional programming with session types which uses hyper-environments to ensure that structural congruence is type preserving.
- Section 4 shows that every well-typed GV configuration is also a well-typed HGV configuration, and every tree-structured HGV configuration is equivalent to a well-typed GV configuration.
- Section 5 gives an operational correspondences between HGV and HCP via translations in both directions that preserve and reflect reduction.

- Section 6 demonstrates the extensibility of HGV through: (1) unconnected processes, (2) a simplified treatment of forwarding, and (3) an improved foundation for exceptions.

Section 2 reviews GV and its metatheory, Section 7 discusses why it is difficult to apply hyper-environments to term typing, Section 8 discusses related work, and Section 9 concludes and discusses future work.

This paper is an improved and extended version of a paper published at CONCUR 2021 [FKD+21]. Additional highlights include:

- a more detailed account of process structures;
- a more detailed account of extensions;
- a more detailed account of the metatheory for HCP; and
- a modified formulation of HCP's labelled transition system and the translation of **fork** in Section 5 fixing errors in the operational correspondence result from the CONCUR 2021 paper.

Proofs of all of the technical results are included in the paper.

2. The Equivalence Embroglio

GV programs are deadlock free, which GV ensures by restricting process structures to trees. A *process structure* is an undirected graph where nodes represent processes and edges represent channels shared between the connected nodes. Session-typed programs with an acyclic process structure are deadlock-free by construction. We illustrate this with a session-typed vending machine example written in GV.

Example 2.1. Consider the session type of a vending machine below, which sells chocolate bars and lollipops. If the vending machine is free, the customer can press ① to receive a chocolate bar or ② to receive a lollipop. If the vending machine is busy, the session ends.

$$\mathsf{VendingMachine} \triangleq \oplus \left\{ \begin{array}{ll} \mathsf{Free} & : \ \&\{① : !\mathsf{ChocolateBar}.\mathbf{end}_!, ② : !\mathsf{Lollipop}.\mathbf{end}_!\} \\ \mathsf{Busy} & : \ \mathbf{end}_! \end{array} \right\}$$

The customer's session type is *dual*: where the vending machine sends a ChocolateBar, the customer receives a ChocolateBar, and so forth. Figure 1 shows the vending machine and customer as a GV program with its process structure.

GV establishes the restriction to tree-structured processes by restricting the primitive for spawning processes. In GV, **fork** has type $(S \multimap \mathbf{end}_!) \multimap \overline{S}$. It takes a closure of type $S \multimap \mathbf{end}_!$ as an argument, creates a channel with endpoints of dual types S and $\overline{S}$, spawns the closure as a new process by supplying one of the endpoints as an argument, and then returns the other endpoint. In essence, **fork** is a branching operation on the process structure: it creates a new node connected to the current node by a single edge. Linearity guarantees that the tree structure is preserved, even in the presence of higher-order channels.

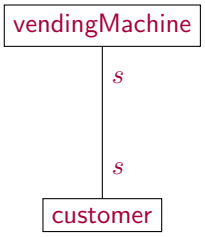
Lindley and Morris [LM15] introduce a semantics for GV, which evaluates programs embedded in process configurations, consisting of embedded programs, flagged as main ($\bullet$) or child ($\circ$) threads, ν-binders to create new channels, and parallel compositions:

$$\mathcal{C}, \mathcal{D} ::= \bullet\, M \mid \circ\, M \mid (\nu x)\mathcal{C} \mid (\mathcal{C} \parallel \mathcal{D})$$

They introduce these process configurations together with a standard structural congruence, which allows, amongst other things, the reordering of processes using commutativity ($\mathcal{C} \parallel \mathcal{C}' \equiv \mathcal{C}' \parallel \mathcal{C}$), associativity ($\mathcal{C} \parallel (\mathcal{C}' \parallel \mathcal{C}'') \equiv (\mathcal{C} \parallel \mathcal{C}') \parallel \mathcal{C}''$), and scope extrusion

```
let vendingMachine = λs.
  let s = select Free s in
  let s = offer s { ① ↦ send chocolateBar
                  ② ↦ send lollipop }
  close s
in let customer = λs.
  offer s { Free ↦ let s = select ① s in
                   let (cb, s) = recv s in
                   wait s; eat cb
          Busy ↦ wait s; hungry }
in let s = fork (λs.vendingMachine s)
in customer s
```

(A) Vending machine and customer as a GV program.



(B) Process structure of Figure 1a.

Figure 1. Example program with acyclic process structure.

($\mathcal{C} \parallel (\nu x)\mathcal{C}' \equiv (\nu x)(\mathcal{C} \parallel \mathcal{C}')$ if $x \notin \mathrm{fv}(\mathcal{C})$). They guarantee acyclicity by defining an extrinsic type system for configurations. In particular, the type system requires that in every parallel composition $\mathcal{C} \parallel \mathcal{D}$, configurations $\mathcal{C}$ and $\mathcal{D}$ must have exactly one channel in common, and that in a name restriction $(\nu x)\mathcal{C}$, channel x cannot be used until it is shared across a parallel composition.

These restrictions are sufficient to guarantee deadlock freedom. Unfortunately, they are *not* preserved by process equivalence. As Lindley and Morris write, (noting that their name restrictions bind *channels* rather than endpoint pairs, and their (νxy) abbreviates $(\nu x)(\nu y)$):

> Alas, our notion of typing is not preserved by configuration equivalence. For example, assume that $\Gamma \vdash (\nu xy)(C_1 \parallel (C_2 \parallel C_3))$, where $x \in \mathrm{fv}(C_1), y \in \mathrm{fv}(C_2)$, and $x, y \in \mathrm{fv}(C_3)$. We have that $C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3$, but $\Gamma \nvdash (\nu xy)((C_1 \parallel C_2) \parallel C_3)$, as both x and y must be shared between the processes $C_1 \parallel C_2$ and C_3.

As a result, standard notions of progress and preservation are not enough to guarantee deadlock freedom, as reduction sequences could include equivalence steps from well-typed to non-well-typed terms. Instead, they must prove a stronger result:

Theorem 3 (Lindley and Morris [LM15])**.** *If $\Gamma \vdash \mathcal{C}$, $\mathcal{C} \equiv \mathcal{C}'$, and $\mathcal{C}' \longrightarrow \mathcal{D}'$, then there exists $\mathcal{D}$ such that $\mathcal{D} \equiv \mathcal{D}'$ and $\Gamma \vdash \mathcal{D}$.*

This is not a one-time cost: languages based on GV must either also give up on type preservation for structural congruence [FLMD19] or admit deadlocks [ITT+19, TV20].

Note that CP only avoids the same issue through its combined $(\nu x)(P \parallel Q)$ term; attempts to split the term into a separate name restriction and parallel composition would also lose typability of equivalence.

3. Hypersequent GV

We present Hypersequent GV (HGV), a linear λ-calculus extended with session types and primitives for session-typed communication. HGV shares its syntax and static typing with GV, but uses hyper-environments for runtime typing to simplify and generalise its semantics.

Typing rules for terms $\boxed{\Gamma \vdash M : T}$

$$\textsc{TM-Var}\ \frac{}{x : T \vdash x : T} \qquad \textsc{TM-Const}\ \frac{}{\cdot \vdash K : T} \qquad \textsc{TM-Lam}\ \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U} \qquad \textsc{TM-App}\ \frac{\Gamma \vdash M : T \multimap U \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M\ N : U}$$

$$\textsc{TM-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \textsc{TM-LetUnit}\ \frac{\Gamma \vdash M : \mathbf{1} \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash \mathbf{let}\ () = M\ \mathbf{in}\ N : T} \qquad \textsc{TM-Pair}\ \frac{\Gamma \vdash M : T \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash (M, N) : T \times U}$$

$$\textsc{TM-LetPair}\ \frac{\Gamma \vdash M : T \times T' \qquad \Delta, x : T, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U} \qquad \textsc{TM-Absurd}\ \frac{\Gamma \vdash M : \mathbf{0}}{\Gamma \vdash \mathbf{absurd}\ M : T} \qquad \textsc{TM-Inl}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash \mathbf{inl}\ M : T + U}$$

$$\textsc{TM-Inr}\ \frac{\Gamma \vdash M : U}{\Gamma \vdash \mathbf{inr}\ M : T + U} \qquad \textsc{TM-CaseSum}\ \frac{\Gamma \vdash L : T + T' \qquad \Delta, x : T \vdash M : U \qquad \Delta, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

Type schemas for communication primitives $\boxed{K : T}$

$$\mathbf{link} : S \times \overline{S} \multimap \mathbf{end}_! \qquad \mathbf{send} : T \times\, !T.S \multimap S \qquad \mathbf{wait} : \mathbf{end}_? \multimap \mathbf{1}$$
$$\mathbf{fork} : (S \multimap \mathbf{end}_!) \multimap \overline{S} \qquad \mathbf{recv} : ?T.S \multimap T \times S$$

Duality $\boxed{\overline{S}}$

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\mathbf{end}_!} = \mathbf{end}_? \qquad \overline{\mathbf{end}_?} = \mathbf{end}_!$$

FIGURE 2. HGV, duality and typing rules for terms.

Types, terms, and static typing. Types (T, U) comprise a unit type ($\mathbf{1}$), an empty type ($\mathbf{0}$), product types ($T \times U$), sum types ($T + U$), linear function types ($T \multimap U$), and session types (S).

$$T, U ::= \mathbf{1} \mid \mathbf{0} \mid T \times U \mid T + U \mid T \multimap U \mid S \qquad S ::= !T.S \mid ?T.S \mid \mathbf{end}_! \mid \mathbf{end}_?$$

Session types (S) comprise output ($!T.S$: send a value of type T, then behave like S), input ($?T.S$: receive a value of type T, then behave like S), and dual end types ($\mathbf{end}_!$ and $\mathbf{end}_?$). The dual endpoints restrict process structure to *trees* [Wad14]; conflating them loosens this restriction to *forests* [ALM16]. We let Γ, Δ range over type environments.

The terms and typing rules are given in Figure 2. The linear λ-calculus rules are standard; communication primitives K are given as constants. Each communication primitive K has a type schema: **link** takes a pair of compatible endpoints and forwards all messages between them; **fork** takes a function, which is passed one endpoint (of type S) of a fresh channel yielding a new child thread, and returns the other endpoint (of type $\overline{S}$); **send** takes a pair of a value and an endpoint, sends the value over the endpoint, and returns an updated endpoint; **recv** takes an endpoint, receives a value over the endpoint, and returns the pair of the received value and an updated endpoint; and **wait** synchronises on a terminated endpoint of type $\mathbf{end}_?$. Output is dual to input, and $\mathbf{end}_!$ is dual to $\mathbf{end}_?$. Duality is involutive, *i.e.*, $\overline{\overline{S}} = S$.

We write $M; N$ for $\mathbf{let}\ () = M\ \mathbf{in}\ N$, $\mathbf{let}\ x = M\ \mathbf{in}\ N$ for $(\lambda x.N)\ M$, $\lambda().M$ for $\lambda z.z; M$, and $\lambda(x, y).M$ for $\lambda z.\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M$. We write $K : T$ for $\cdot \vdash K : T$ in typing derivations.

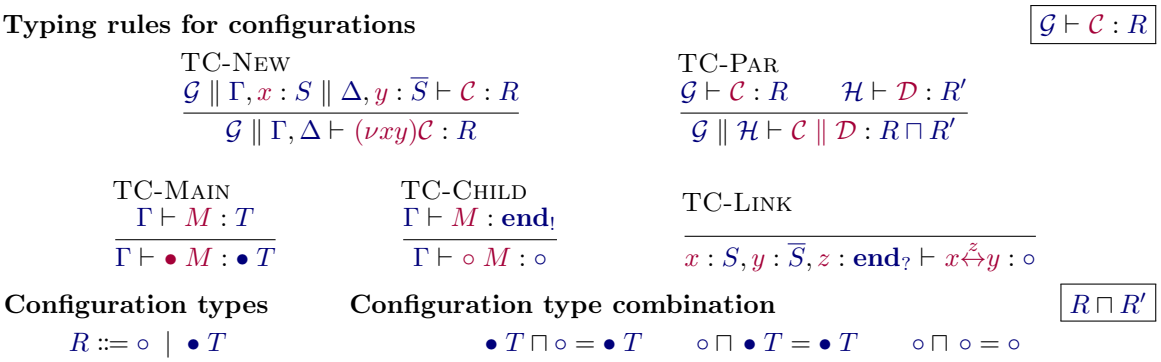


Figure 3. HGV, typing rules for configurations.

Remark 3.1. We include **link** because it is convenient for the correspondence with CP, which interprets CLL's axiom as forwarding. We *can* encode **link** in GV via a type directed translation akin to CLL's *identity expansion*.

Configurations and runtime typing. Process configurations ($\mathcal{C}, \mathcal{D}, \mathcal{E}$) comprise child threads ($\circ\, M$), the main thread ($\bullet\, M$), link threads ($x \overset{z}{\leftrightarrow} y$), name restrictions ($(\nu xy)\mathcal{C}$), and parallel compositions ($\mathcal{C} \parallel \mathcal{D}$). We refer to a configuration of the form $\circ M$ or $x \overset{z}{\leftrightarrow} y$ as an *auxiliary thread*, and a configuration of the form $\bullet M$ as a *main thread*. We let $\mathcal{A}$ range over auxiliary threads and $\mathcal{T}$ range over all threads (auxiliary or main).

$$\phi ::= \bullet \mid \circ \qquad \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \phi\, M \mid x \overset{z}{\leftrightarrow} y \mid \mathcal{C} \parallel \mathcal{D} \mid (\nu xy)\mathcal{C}$$

The configuration language is reminiscent of π-calculus processes, but has some non-standard features. Name restriction uses double binders [Vas12] in which one name is bound to each endpoint of the channel. Link threads [LM16] handle forwarding. A link thread $x \overset{z}{\leftrightarrow} y$ waits for the thread connected to z to terminate before forwarding all messages between x and y.

Configuration typing departs from GV [LM15], exploiting *hypersequents* [Avr91] to recover modularity and extensibility. Inspired by HCP [MP18, KMP19b, KMP19a], configurations are typed under a *hyper-environment*, an unordered collection of disjoint type environments. We let $\mathcal{G}, \mathcal{H}$ range over hyper-environments, writing $\varnothing$ for the empty hyper-environment, $\mathcal{G} \parallel \Gamma$ for disjoint extension of $\mathcal{G}$ with type environment Γ, and $\mathcal{G} \parallel \mathcal{H}$ for disjoint concatenation of $\mathcal{G}$ and $\mathcal{H}$.

The typing rules for configurations are given in Figure 3. Rules TC-New and TC-Par are key to deadlock freedom: TC-New joins two disjoint configurations with a new channel, and merges their type environments; TC-Par combines two disjoint configurations, and registers their disjointness by separating their type environments in the hyper-environment. Rules TC-Main, TC-Child, and TC-Link type main, child, and link threads, respectively; all three require a singleton hyper-environment. A configuration has type $\circ$ if it has no main thread, and $\bullet\ T$ if it has a main thread of type T. The configuration type combination operator ensures that a well-typed configuration has at most one main thread.

Operational semantics. Figure 4 gives the operational semantics for HGV, presented as a deterministic reduction relation on terms and a nondeterministic reduction relation on configurations. HGV values (U, V, W), evaluation contexts (E), and term reduction rules

Values and evaluation contexts

Values	U, V, W	$::=$	$K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$
Evaluation contexts	E	$::=$	$\square$
		$\mid$	$E\ M \mid V\ E$
		$\mid$	$\mathbf{let}\ () = E\ \mathbf{in}\ M$
		$\mid$	$(E, M) \mid (V, E) \mid \mathbf{let}\ (x, y) = E\ \mathbf{in}\ M$
		$\mid$	$\mathbf{inl}\ E \mid \mathbf{inr}\ E \mid \mathbf{case}\ E\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$
Thread contexts	F	$::=$	$\phi\ E$

Term reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

E-Lam	$(\lambda x.M)\ V$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x\}$
E-Unit	$\mathbf{let}\ () = ()\ \mathbf{in}\ M$	$\longrightarrow_{\mathsf{M}}$	M
E-Pair	$\mathbf{let}\ (x, y) = (V, W)\ \mathbf{in}\ M$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x, W/y\}$
E-Inl	$\mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x\}$
E-Inr	$\mathbf{case}\ \mathbf{inr}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$	$\longrightarrow_{\mathsf{M}}$	$N\{V/y\}$
E-Lift	$E[M]$	$\longrightarrow_{\mathsf{M}}$	$E[N]$, if $M \longrightarrow_{\mathsf{M}} N$

Structural congruence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

SC-ParAssoc	$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$	SC-ParComm	$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C}$
SC-NewComm	$(\nu xy)(\nu zw)\mathcal{C} \equiv (\nu zw)(\nu xy)\mathcal{C}$	SC-NewSwap	$(\nu xy)\mathcal{C} \equiv (\nu yx)\mathcal{C}$
SC-ScopeExt	$(\nu xy)(\mathcal{C} \parallel \mathcal{D}) \equiv \mathcal{C} \parallel (\nu xy)\mathcal{D}$, if $x, y \notin \mathrm{fv}(\mathcal{C})$	SC-LinkComm	$x \overset{z}{\leftrightarrow} y \equiv y \overset{z}{\leftrightarrow} x$

Configuration reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

E-Reify-Fork	$F[\mathbf{fork}\ V]$	$\longrightarrow$	$(\nu xx')(F[x] \parallel \circ\ (V\ x'))$, where x, x' fresh
E-Reify-Link	$F[\mathbf{link}\ (x, y)]$	$\longrightarrow$	$(\nu zz')(x \overset{z}{\leftrightarrow} y \parallel F[z'])$, where z, z' fresh
E-Comm-Link	$(\nu zz')(\nu xx')(x \overset{z}{\leftrightarrow} y \parallel \circ\ z' \parallel \phi\ M)$	$\longrightarrow$	$\phi\ (M\{y/x'\})$
E-Comm-Send	$(\nu xy)(F[\mathbf{send}\ (V, x)] \parallel F'[\mathbf{recv}\ y])$	$\longrightarrow$	$(\nu xy)(F[x] \parallel F'[(V, y)])$
E-Comm-Close	$(\nu xy)(\circ\ y \parallel F[\mathbf{wait}\ x])$	$\longrightarrow$	$F[()]$

$$\text{E-Res}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{(\nu xy)\mathcal{C} \longrightarrow (\nu xy)\mathcal{C}'} \qquad \text{E-Par}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D}} \qquad \text{E-Equiv}\ \frac{\mathcal{C} \equiv \mathcal{C}' \quad \mathcal{C}' \longrightarrow \mathcal{D}' \quad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow \mathcal{D}} \qquad \text{E-Lift-M}\ \frac{M \longrightarrow_{\mathsf{M}} M'}{F[M] \longrightarrow F[M']}$$

FIGURE 4. HGV, operational semantics.

$(\longrightarrow_{\mathsf{M}})$ define a standard call-by-value, left-to-right evaluation strategy. A closed term either reduces to a value or is blocked on a communication action.

Thread contexts (F) extend evaluation contexts to threads. The structural congruence rules are standard apart from SC-LinkComm, which ensures links are undirected, and SC-NewSwap, which swaps names in double binders.

The configuration reduction relation gives a semantics for HGV's communication and concurrency constructs. The first two rules, E-Reify-Fork and E-Reify-Link, create child and link threads, respectively. The next three rules, E-Comm-Link, E-Comm-Send, and E-Comm-Close perform communication actions. The final four rules enable reduction under name restriction and parallel composition, rewriting by structural congruence, and term reduction in threads. Two rules handle links: E-Reify-Link creates a new *link thread* $x \overset{z}{\leftrightarrow} y$ which blocks on z of type $\mathbf{end}_?$, one endpoint of a fresh channel. The other endpoint, z' of

type $\mathbf{end}_!$, is placed in the evaluation context of the parent thread. When z' terminates a child thread, E-Comm-Link performs forwarding by substitution.

Remark 3.2. Note that E-Comm-Link does not fire if z' is returned by a main thread. In closed configurations, typing ensures that such a configuration cannot arise: intuitively, a main thread can only obtain endpoints by **fork** or by receiving an endpoint.

Endpoints generated to communicate with forked threads (i.e., those passed to a child thread) will always have a session type terminating with $\mathbf{end}_?$, and a child thread cannot transmit an endpoint ending in $\mathbf{end}_!$, since the endpoint must be returned. Consequently, there is no way for a main thread to obtain endpoints with dual session types as required by the type of **link**. The case for open configurations is accounted for by our open progress result (see Section 3.1).

Choice. HGV does not include constructs for internal and external choice (for example, as shown in the vending machine example in Section 1). Internal and external choice are instead encoded with sum types and session delegation [Kob03, DGS17]. Prior encodings of choice in GV [LM15] are asynchronous. Instead, to encode synchronous choice we add a 'dummy' synchronisation before exchanging the value of sum type, as follows:

$$
\begin{aligned}
S \oplus S' &\triangleq !\mathbf{1}.!(\overline{S_1} + \overline{S_2}).\mathbf{end}_! \\
S \,\&\, S' &\triangleq ?\mathbf{1}.?(S_1 + S_2).\mathbf{end}_? \\
\\
\oplus\{\} &\triangleq !\mathbf{1}.!\mathbf{0}.\mathbf{end}_! \\
\&\{\} &\triangleq ?\mathbf{1}.?\mathbf{0}.\mathbf{end}_?
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{select}\ \ell \triangleq \lambda x.\left(\begin{array}{l}\mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in} \\ \mathbf{fork}\ (\lambda y.\mathbf{send}\ (\ell\ y, x))\end{array}\right) \\
&\mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
&\qquad \triangleq \begin{array}{l}\mathbf{let}\ ((), z) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (w, z) = \mathbf{recv}\ z \\ \mathbf{in}\ \mathbf{wait}\ z; \mathbf{case}\ w\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}\end{array} \\
&\mathbf{offer}\ L\ \{\} \triangleq \begin{array}{l}\mathbf{let}\ ((), c) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{let}\ (z, c) = \mathbf{recv}\ c \\ \mathbf{in}\ \mathbf{wait}\ c; \mathbf{absurd}\ z\end{array}
\end{aligned}
$$

3.1. Metatheory.

HGV enjoys type preservation, deadlock freedom, confluence, and strong normalisation.

Preservation. Hyper-environments enable type preservation under structural congruence, which significantly simplifies the metatheory compared to GV.

Theorem 3.3 (Preservation)**.**

(1) *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$*.*
(2) *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$*.*

Proof. By induction on the derivations of $\mathcal{C} \equiv \mathcal{D}$ and $\mathcal{C} \longrightarrow \mathcal{D}$. See Appendix A. □

Before moving onto progress, we must introduce some technical machinery to allow us to reason about the structure of HGV programs.

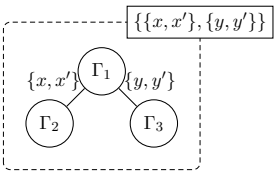
Abstract process structures. Unlike in GV, in HGV we cannot rely on the fact that exactly one channel is split over each parallel composition. Instead, we introduce the notion of an *abstract process structure* (APS). Abstract process structures are a crucial ingredient in showing that HGV configurations can be written in *tree canonical form*, which helps both with establishing progress results and also the correspondence between HGV and GV.

We begin by establishing the intuition behind the notion of an APS, and then describe the formal definitions. An APS is a graph defined over a hyper-environment $\mathcal{G}$ and a set of undirected pairs of co-names (a *co-name set*) $\mathcal{N}$ drawn from the names in $\mathcal{G}$.

The nodes of an APS are the type environments in $\mathcal{G}$. Each edge is labelled by a distinct co-name pair $\{x_1, x_2\} \in \mathcal{N}$, such that $x_1 : S \in \Gamma_1$ and $x_2 : \overline{S} \in \Gamma_2$.

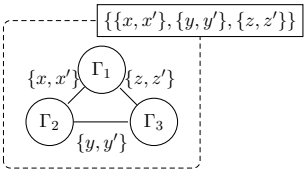
Example 3.4.

Let $\mathcal{G} = \Gamma_1 \parallel \Gamma_2 \parallel \Gamma_3$, where $\Gamma_1 = x : S_1, y : S_2$, $\Gamma_2 = x' : \overline{S_1}, z : T$, and $\Gamma_3 = y' : \overline{S_2}$, and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}\}$. The APS for $\mathcal{G}$ and $\mathcal{N}$ is illustrated to the right.



Example 3.5.

Let $\mathcal{G} = \Gamma_1 \parallel \Gamma_2 \parallel \Gamma_3$, where $\Gamma_1 = x : S_1, z' : \overline{S_3}$, and $\Gamma_2 = x' : \overline{S_1}, y : S_2$, and $\Gamma_3 = y' : \overline{S_2}, z : S_3$, and suppose $\mathcal{N} = \{\{x, x'\}, \{y, y'\}, \{z, z'\}\}$. The APS for $\mathcal{G}$ and $\mathcal{N}$ is illustrated to the right.



Let us now discuss the formal definition of an APS. We begin by recalling the definition of an *undirected edge-labelled multigraph*: an undirected graph that allows multiple edges between vertices.

Definition 3.6 (Undirected Multigraph)**.** An *undirected multigraph* G is a 3-tuple $(\mathcal{V}, \mathcal{E}, r)$ where:

(1) $\mathcal{V}$ is a set of vertices
(2) $\mathcal{E}$ is a set of edge names
(3) r is a function $r : \mathcal{E} \mapsto \{\{v, w\} : v, w \in \mathcal{V}\}$ from edge names to an unordered pair of vertices

Denote the size of a set as $|\cdot|$. A *path* is a sequence of edges connecting two vertices. A multigraph $G = (\mathcal{V}, \mathcal{E}, r)$ is *connected* if $|\mathcal{V}| = 1$, or if for every pair of vertices $v, w \in \mathcal{V}$ there is a path between v and w. A multigraph is *acyclic* if no path forms a cycle. A *leaf* is a vertex connected to the remainder of a graph by a single edge.

Definition 3.7 (Leaf)**.** Given an undirected multigraph $(\mathcal{V}, \mathcal{E}, r)$, a vertex $v \in \mathcal{V}$ is a *leaf* if there exists a single $e \in \mathcal{E}$ such that $v \in r(e)$.

In an undirected tree containing at least two vertices, there must be at least two leaves.

Lemma 3.8. *If $G = (\mathcal{V}, \mathcal{E}, r)$ is an undirected tree where $|V| \geq 2$, then there exist at least two leaves in $\mathcal{V}$.*

Proof. For G to be an undirected tree where $|V| \geq 2$ and have fewer than two leaves, then there would need to be a cycle, contradicting acyclicity. □

With the graph preliminaries in place, we are now ready to introduce the formal definition of an APS.

Definition 3.9 (Abstract process structure)**.** The *abstract process structure of a hyper-environment* $\mathcal{H}$ *with respect to a co-name set* $\mathcal{N} = \{\{x_1, y_1\}, \ldots, \{x_n, y_n\}\}$ is an undirected multigraph $(\mathcal{V}, \mathcal{E}, r)$ defined as follows:

(1) $\mathcal{V} = \mathsf{envs}(\mathcal{H})$
(2) $\mathcal{E} = \mathcal{N}$
(3) $r = (\{x, y\} \mapsto \{\Gamma_1, \Gamma_2\})$ for each $\{x, y\} \in \mathcal{N}$ such that $\Gamma_1 \in \mathsf{envs}(\mathcal{H}), \Gamma_2 \in \mathsf{envs}(\mathcal{H}), x \in \mathrm{fv}(\Gamma_1), y \in \mathrm{fv}(\Gamma_2)$

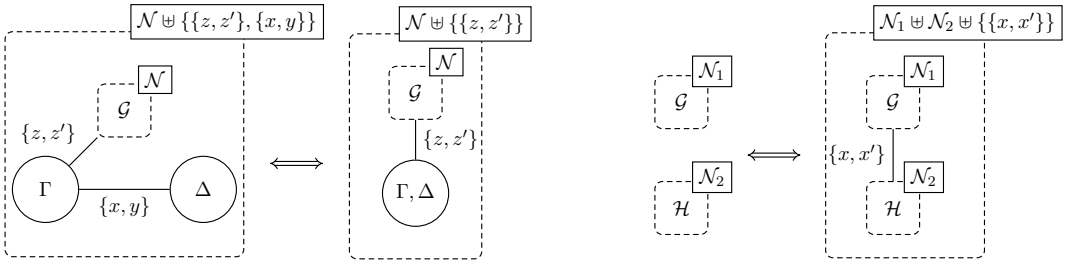
Example 3.10. The formal definition of the APS described in Example 3.4 is defined as:

- $\mathcal{V} = \{\Gamma_1, \Gamma_2, \Gamma_3\}$
- $\mathcal{E} = \{\{x, x'\}, \{y, y'\}\}$
- $r(\{x, x'\}) \mapsto \{\Gamma_1, \Gamma_2\})$
  $r(\{y, y'\}) \mapsto \{\Gamma_1, \Gamma_3\})$

Whereas Example 3.4 is a tree, Example 3.5 contains a cycle. Only configurations typeable under a hyper-environment with a *tree structure* can be written in tree canonical form.

Definition 3.11 (Tree structure)**.** A hyper-environment $\mathcal{H}$ with co-name set $\mathcal{N}$ has a *tree structure*, written $\mathsf{Tree}(\mathcal{H}, \mathcal{N})$, if its APS is connected and acyclic.

An HGV program $\bullet\, M$ has a single type environment, so is tree-structured; the same goes for child and link threads. A key feature of HGV is a subformula principle, which states that all hyper-environments arising in the derivation of an HGV program are tree-structured. It follows that a configuration resulting from the reduction of an HGV program is also tree structured. Read bottom-up, TC-New and TC-Par preserve tree structure, which is illustrated by the following two pictures.



The following lemma states this intuition formally. By analogy to Kleene equality, we write $\mathcal{P} \mathrel{\hat{\iff}} \mathcal{Q}$, to mean that either $\mathcal{P}$ or $\mathcal{Q}$ is undefined, or $\mathcal{P} \iff \mathcal{Q}$.

Lemma 3.12 (Tree structure)**.**

- $\mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, x_1 : S \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N} \uplus \{\{x_1, x_2\}\}) \mathrel{\hat{\iff}} \mathsf{Tree}((\mathcal{H} \parallel \Gamma_1, \Gamma_2)), \mathcal{N})$
- $\mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S), \mathcal{N}_1) \wedge \mathsf{Tree}((\mathcal{H}_2 \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N}_2) \mathrel{\hat{\iff}} \mathsf{Tree}((\mathcal{H}_1 \parallel \Gamma_1, x_1 : S \parallel \mathcal{H}_2 \parallel \Gamma_2, x_2 : \overline{S}), \mathcal{N}_1 \uplus \mathcal{N}_2 \uplus \{\{x_1, x_2\}\})$

Proof. By the definition of $\mathrel{\hat{\iff}}$, we need only consider the cases where both sides of the bi-implication are defined. Both results follow from the observation that adding an edge between two trees results in a tree, and removing an edge from a tree partitions the tree into two subtrees. □

Tree canonical form. We now define a canonical form for configurations that captures the tree structure of an APS. Tree canonical form enables a succinct statement of *open progress* (Lemma 3.17) and a means for embedding HGV in GV (Proposition 4.5).

Definition 3.13 (Tree canonical form)**.** A configuration $\mathcal{C}$ is in *tree canonical form* if it can be written: $(\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$ where $x_i \in \mathrm{fv}(\mathcal{A}_i)$ for $1 \leq i \leq n$.

Every well-typed HGV configuration typeable under a single type environment can be written in tree canonical form.

Theorem 3.14 (Well-typed configurations in tree canonical forms)**.** *If $\Gamma \vdash \mathcal{C} : R$, then there exists some $\mathcal{D}$ such that $\mathcal{C} \equiv \mathcal{D}$ and $\mathcal{D}$ is in tree canonical form.*

Proof. By induction on the number of ν-binders in $\mathcal{C}$. In the case that $n = 0$, it must be the case that $\Gamma \vdash \phi M : R$ for some thread M, since parallel composition is only typeable under a hyper-environment containing two or more type environments. Therefore, $\mathcal{C}$ is in tree canonical form by definition.

In the case that $n \geq 1$, by Theorem 3.3, we can rewrite the configuration as:

$$(\nu x_1 y_1) \cdots (\nu x_n y_n)(\circ M_1 \parallel \cdots \parallel \circ M_n \parallel \phi N)$$

Fix $\mathcal{N} = \{\{x_i, y_i\} \mid 1 \leq i \leq n\}$. By definition, Γ has a tree structure with respect to an empty co-name set. By repeated applications of TC-New, there exists some $\mathcal{G}$ such that $\mathcal{G} \vdash \circ M_1 \parallel \cdots \parallel \circ M_n \parallel \phi N : T$; by Lemma 3.12 (clause 1, right-to-left), $\mathcal{G}$ has a tree structure.

Construct the APS for $\mathcal{G}$ using names $\mathcal{N}$; by Lemma 3.8, there exist $\Gamma_1, \Gamma_2 \in \mathsf{envs}(\mathcal{H})$ such that Γ_1 and Γ_2 are leaves of the tree and therefore by the definition of the APS contain precisely one ν-bound name. By TC-Par, there must exist two threads $\mathcal{C}_1, \mathcal{C}_2$ such that $\Gamma_1 \vdash \mathcal{C}_1 : R_1$ and $\Gamma_2 \vdash \mathcal{C}_2 : R_2$. By runtime type combination, at least one of R_1, R_2 must be $\circ$; without loss of generality assume this is R_1. Suppose (again without loss of generality) that the ν-bound name contained in Γ_1 is x_1 and $L_1 = M_1$.

Let $\mathcal{D} = (\nu x_2 y_2) \cdots (\nu x_n y_n)(\circ M_2 \parallel \cdots \parallel \circ M_n \parallel \phi N)$. By Theorem 3.3 and the fact that x_1 is the only ν-bound variable in M_1, we have that $\mathcal{C} \equiv (\nu x_1 y_1)(\circ M_1 \parallel \mathcal{D})$. By the induction hypothesis, there exists some $\mathcal{D}'$ such that $\mathcal{D} \equiv \mathcal{D}'$ and $\mathcal{D}'$ is in canonical form. By construction we have that $\mathcal{C} \equiv (\nu x_1 y_1)(\circ M_1 \parallel \mathcal{D}')$, which is in tree canonical form as required. □

As hyper-environments capture parallelism, a configuration $\mathcal{C}$ typeable under hyper-environment $\Gamma_1 \parallel \cdots \parallel \Gamma_n$ is equivalent to n independent parallel processes.

Proposition 3.15 (Independence)**.** *If $\Gamma_1 \parallel \cdots \parallel \Gamma_n \vdash \mathcal{C} : R$, then there exist $R_1, \ldots, R_n$ and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ such that $R = R_1 \sqcap \cdots \sqcap R_n$ and $\mathcal{C} \equiv \mathcal{D}_1 \parallel \cdots \parallel \mathcal{D}_n$ and $\Gamma_i \vdash \mathcal{D}_i : R_i$ for each i.*

Proof. By induction on the derivation of $\Gamma_1 \parallel \cdots \parallel \Gamma_n \vdash \mathcal{C} : R$. The cases for TC-Main, TC-Child, and TC-Link follow immediately. The cases for TC-New and TC-Par follow from the IH and structural congruence rules. □

It follows from Theorem 3.14 and Proposition 3.15 that any well-typed HGV configuration can be written as a forest of independent configurations in tree canonical form.

Progress and Deadlock Freedom. With tree canonical forms defined, we can now state a progress result. A thread is *blocked* on an endpoint x if it is ready to perform a communication action on x.

Definition 3.16 (Blocked thread)**.** We say that thread $\mathcal{T}$ is *blocked on variable* z, written $\mathsf{blocked}(\mathcal{T}, z)$, if either: $\mathcal{T} = \circ\, z$; $\mathcal{T} = x \overset{z}{\leftrightarrow} y$, for some x, y; or $\mathcal{T} = F[N]$ for some F, where N is $\mathbf{send}\ (V, z)$, $\mathbf{recv}\ z$, or $\mathbf{wait}\ z$.

We let Ψ range over type environments containing only session-typed variables, *i.e.*, $\Psi ::= \cdot \mid \Psi, x : S$, which lets us reason about configurations that are closed except for runtime names. Using Lemma 3.17 we obtain *open progress* for configurations with free runtime names.

Lemma 3.17 (Open Progress)**.** *Suppose* $\Psi \vdash \mathcal{C} : T$ *where*

$$\mathcal{C} = (\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$$

is in tree canonical form. Either $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*, or:*

(1) *For each* $\mathcal{A}_i$ $(1 \le i \le n)$, $\mathsf{blocked}(\mathcal{A}_i, z)$ *for some* $z \in \{x_i\} \cup \{y_j \mid 1 \le j < i\} \cup \mathrm{fv}(\Psi)$
(2) *Either* N *is a value or* $\mathsf{blocked}(\phi N, z)$ *for some* $z \in \{y_i \mid 1 \le i \le n\} \cup \mathrm{fv}(\Psi)$

Proof. Open progress follows as a direct corollary of a slightly more verbose property which holds on HGV processes, proved by induction on the derivation of an inductive definition of tree canonical forms. See Appendix A for details. □

Closed configurations enjoy a stronger result: if a closed configuration cannot reduce, then each auxiliary thread must either be a value, or be blocked on its neighbouring endpoint.

Lemma 3.18 (Closed Progress)**.** *Suppose* $\Psi \vdash \mathcal{C} : R$ *where*

$$\mathcal{C} = (\nu x_1 y_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\mathcal{A}_n \parallel \phi N) \cdots)$$

is in tree canonical form. Either $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*, or:*

(1) *For each* $\mathcal{A}_j$ *for* $1 \le j \le n$, $\mathsf{blocked}(\mathcal{A}_j, x_j)$
(2) N *is a value*

Proof. Since the environment is closed, by Lemma 3.17, for each $\mathcal{A}_j$ it must be that $\mathsf{blocked}(\mathcal{A}_j, z)$ for some $z \in \{y_i \mid i \in 1..j-1\} \cup \{x_j\}$.

Note that if two names x, y are co-names, and one thread is blocked on x, and another is blocked on y, then due to typing the names must be dual and reduction can occur.

Consider $\mathcal{A}_1$. Since the environment is closed, $\mathcal{A}_1$ must be blocked on x_1. Next, consider $\mathcal{A}_2$; the thread cannot be blocked on y_1 as reduction would occur. By the definition of tree canonical forms, $\mathcal{A}_2$ must contain x_2 and by the typing rules cannot contain y_2, so the thread must be blocked on x_2. The argument extends to the remainder of the configuration. □

Finally, for *ground configurations*, where the main thread does not return a runtime name or capture a runtime name in a closure, we obtain a yet tighter result, *global progress*, which implies deadlock freedom [CDM14].

Definition 3.19 (Ground configuration)**.** A configuration $\mathcal{C}$ is a *ground configuration* if $\cdot \vdash \mathcal{C} : T$, $\mathcal{C}$ is in canonical form, and T does not contain session types or function types.

Our main progress result states that a ground configuration can reduce, or is a value.

Typing rules for configurations $\boxed{\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : T}$

$$\text{TG-New}\ \frac{\Gamma, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} : R}{\Gamma \vdash_{\mathsf{GV}} (\nu xy)\mathcal{C} : R} \qquad \text{TG-Connect}_1\ \frac{\Gamma_1, x : S \vdash_{\mathsf{GV}} \mathcal{C} : R \quad \Gamma_2, y : \overline{S} \vdash_{\mathsf{GV}} \mathcal{D} : R'}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R \sqcap R'}$$

$$\text{TG-Connect}_2\ \frac{\Gamma_1, y : \overline{S} \vdash_{\mathsf{GV}} \mathcal{C} : R \quad \Gamma_2, x : S \vdash_{\mathsf{GV}} \mathcal{D} : R'}{\Gamma_1, \Gamma_2, \langle x, y\rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R \sqcap R'}$$

$$\text{TG-Child}\ \frac{\Gamma \vdash_{\mathsf{GV}} M : \mathbf{end}_!}{\Gamma \vdash_{\mathsf{GV}} \circ M : \circ} \qquad \text{TG-Main}\ \frac{\Gamma \vdash_{\mathsf{GV}} M : T}{\Gamma \vdash_{\mathsf{GV}} \bullet M : \bullet\, T} \qquad \text{TG-Link}\ \frac{}{x : S, y : \overline{S}, z : \mathbf{end}_? \vdash_{\mathsf{GV}} x \overset{z}{\leftrightarrow} y : \circ}$$

FIGURE 5. GV, typing rules for configurations.

Theorem 3.20 (Global progress)**.** *Suppose $\mathcal{C}$ is a ground configuration. Either there exists some $\mathcal{D}$ such that $\mathcal{C} \longrightarrow \mathcal{D}$, or $\mathcal{C} = \bullet V$ for some value V.*

Proof. By Lemma 3.18, either $\mathcal{C}$ can reduce, or $\mathcal{C}$ can be written:

$$(\nu x_1 y_1)(\circ \mathcal{A}_1 \parallel \cdots \parallel (\nu x_n y_n)(\circ \mathcal{A}_n \parallel \bullet V) \cdots)$$

where $\mathsf{blocked}(\mathcal{A}_i, x_i)$ for each $\{x_i \mid i \in 1..n\}$.

Since $\mathcal{C}$ is ground, $\mathrm{fv}(V) = \emptyset$. By definition, tree canonical form ensures that no cycles are present amongst threads, so no auxiliary thread can be blocked. It follows that if $\mathcal{C} \not\longrightarrow$, then there cannot be any auxiliary threads and thus $\mathcal{C} = \bullet V$ for some value V. □

Determinism and Strong Normalisation. HGV enjoys a strong form of determinism known as the diamond property, and due to linearity it enjoys strong normalisation. Unlike with preservation and progress, the addition of hypersequents does not substantially change the arguments from [LM15].

Theorem 3.21 (Diamond property)**.** *If $\mathcal{G} \vdash \mathcal{C} : T$, $\mathcal{C} \longrightarrow \mathcal{D}$, and $\mathcal{C} \longrightarrow \mathcal{D}'$, then $\mathcal{D} \equiv \mathcal{D}'$.*

Proof. Similar to that of GV [LM15, Fow19]: $\longrightarrow_{\mathsf{M}}$ is deterministic, and due to linearity, any overlapping reductions are separate and may be performed in either order. □

Theorem 3.22 (Termination)**.** *If $\mathcal{G} \vdash \mathcal{C} : T$, there are no infinite sequences $\mathcal{C} \longrightarrow\longrightarrow \cdots$.*

Proof. As with GV [LM15, Fow19], due to linearity, HGV has an elementary strong normalisation proof. Let the size of a configuration be the sum of the sizes of all abstract syntax trees of all terms contained in threads. The size of a configuration is invariant under $\equiv$ and strictly decreases under $\longrightarrow$, so no infinite reduction sequences can exist. □

4. Relation between HGV and GV

In this section, we show that well-typed GV configurations are well-typed HGV configurations, and well-typed HGV configurations with tree structure are well-typed GV configurations.

GV. HGV and GV share a common term language and reduction semantics, so only differ in their runtime typing rules. Figure 5 gives the runtime typing rules for GV. We adapt the rules to use a double-binder formulation to concentrate on the essence of the relationship with HGV, but it is trivial to translate GV with single binders into GV with double binders.

GV uses a pseudo-type $S^\sharp$ to type channels. Unlike endpoints, channels cannot appear in terms. Read bottom-up, rule TG-New types a name restriction $(\nu xy)\mathcal{C}$, adding $\langle x, y\rangle : S^\sharp$ to the type environment, which along with TG-Connect$_1$ and TG-Connect$_2$ ensures that a session channel of type S will be split into endpoints x and y over a parallel composition. In turn, this enforces a tree process structure. The remaining typing rules are as in HGV.

A simple embedding of GV into HGV. The simplest embedding of GV in HGV relies on the observation from Section 2 that each parallel composition splits a single channel. Let $\mathcal{C} \parallel_{\langle x,y\rangle} \mathcal{D}$ denote two configurations $\mathcal{C}$ and $\mathcal{D}$ connected by a channel with endpoints x, y. We can write an arbitrary closed GV configuration in the form:

$$\mathcal{C}_1 \parallel_{\langle x_1,y_1\rangle} \cdots \parallel_{\langle x_{n-2},y_{n-2}\rangle} \mathcal{C}_{n-1} \parallel_{\langle x_{n-1},y_{n-1}\rangle} \mathcal{C}_n$$

where each $\mathcal{C}$ does not contain a further parallel composition, and any main thread is in $\mathcal{C}_n$. We can then subsequently embed the configuration in HGV as:

$$(\nu x_1 y_1)(\mathcal{C}_1 \parallel \cdots \parallel (\nu x_{n-2} y_{n-2})(\mathcal{C}_{n-2} \parallel (\nu x_{n-1} y_{n-1})(\mathcal{C}_{n-1} \parallel \mathcal{C}_n)) \cdots)$$

which is well-typed by construction. As a corollary, every well-typed, closed GV configuration is equivalent to a well-typed, closed HGV configuration.

A structure-preserving embedding of GV into HGV. Though the simple embedding of GV into HGV is sound, it does not respect the *intention* of GV. In fact, we can provide a stronger result: every well-typed open GV configuration is exactly a well-typed HGV configuration.

Definition 4.1 (Flattening)**.** Flattening, written $\downarrow$, converts GV type environments and HGV hyper-environments into HGV environments.

$$\begin{array}{lcllcl} \downarrow\cdot & = & \cdot & \downarrow\varnothing & = & \varnothing \\ \downarrow(\Gamma, \langle x, x'\rangle : S^\sharp) & = & \downarrow\Gamma, x : S, x' : \overline{S} & \downarrow(\mathcal{G} \parallel \Gamma) & = & \downarrow\mathcal{G}, \Gamma \\ \downarrow(\Gamma, x : T) & = & \downarrow\Gamma, x : T & & & \end{array}$$

Definition 4.2 (Splitting)**.** Splitting converts GV type environments into hyper-environments. Given channels $\{\langle x_i, x_i'\rangle : S_i^\sharp\}_{i\in 1..n}$ in Γ, a hyper-environment $\mathcal{G}$ is a *splitting* of Γ if $\downarrow\mathcal{G} = \downarrow\Gamma$ and $\exists\Gamma_1, \ldots, \Gamma_{n+1}$ such that $\mathcal{G} = \Gamma_1 \parallel \cdots \parallel \Gamma_{n+1}$, and $\mathsf{Tree}(\mathcal{G}, \{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\})$.

A well-typed GV configuration is typeable in HGV under a splitting of its type environment.

Theorem 4.3 (Typeability of GV configurations in HGV)**.** *If $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$, then there exists some $\mathcal{G}$ such that $\mathcal{G}$ is a splitting of Γ and $\mathcal{G} \vdash \mathcal{C} : R$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : T$ (see Appendix B). □

Example 4.4. Consider a configuration where a child thread pings the main thread:

$$(\nu xy)(\circ\ (\mathbf{send}\ (\mathit{ping}, x)) \parallel \bullet\ (\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y))$$

We can write a GV typing derivation as follows:

$$\dfrac{\dfrac{x : !\mathbf{1}.\mathbf{end}_!, \mathit{ping} : \mathbf{1} \vdash_{\mathsf{GV}} \circ\ (\mathbf{send}\ (\mathit{ping}, x)) : \circ \qquad y : ?\mathbf{1}.\mathbf{end}_? \vdash_{\mathsf{GV}} \bullet\ (\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y) : \bullet\ \mathbf{1}}{\langle x, y\rangle : !\mathbf{1}.\mathbf{end}_!{}^\sharp, \mathit{ping} : \mathbf{1} \vdash_{\mathsf{GV}} (\nu xy)(\circ(\mathbf{send}\ (\mathit{ping}, x)) \parallel \bullet(\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y)) : \mathbf{1}}}{\mathit{ping} : \mathbf{1} \vdash_{\mathsf{GV}} (\nu xy)(\circ(\mathbf{send}\ (\mathit{ping}, x)) \parallel \bullet(\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y)) : \mathbf{1}}$$

The corresponding HGV derivation is:

$$\dfrac{\dfrac{x : !\mathbf{1}.\mathbf{end}_!, \mathit{ping} : \mathbf{1} \vdash \circ\,(\mathbf{send}\ (\mathit{ping}, x)) : \circ \qquad y : ?\mathbf{1}.\mathbf{end}_? \vdash \bullet\,(\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y) : \bullet\,\mathbf{1}}{x : !\mathbf{1}.\mathbf{end}_!, \mathit{ping} : \mathbf{1} \parallel y : ?\mathbf{1}.\mathbf{end}_? \vdash (\nu xy)(\circ(\mathbf{send}\ (\mathit{ping}, x)) \parallel \bullet(\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y)) : \bullet\,\mathbf{1}}}{\mathit{ping} : \mathbf{1} \vdash (\nu xy)(\circ(\mathbf{send}\ (\mathit{ping}, x)) \parallel \bullet(\mathbf{let}\ ((), y) = \mathbf{recv}\ y\ \mathbf{in}\ \mathbf{wait}\ y)) : \bullet\,\mathbf{1}}$$

Note that $x : !\mathbf{1}.\mathbf{end}_!, \mathit{ping} : \mathbf{1} \parallel y : ?\mathbf{1}.\mathbf{end}_?$ is a splitting of $\langle x, y\rangle : (!\mathbf{1}.\mathbf{end}_!)^\sharp, \mathit{ping} : \mathbf{1}$.

Translating HGV to GV. As we saw in §2, unlike in HGV, equivalence in GV is not type-preserving. It follows that HGV types strictly more processes than GV. Let us revisit Lindley and Morris' example from §1 (adapted to use double-binders), where $\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{\mathsf{GV}} (\nu xx')(\nu yy')(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})) : R_1 \sqcap R_2 \sqcap R_3$ with $\Gamma_1, x : S \vdash_{\mathsf{GV}} \mathcal{C} : R_1$, $\Gamma_2, y : S' \vdash_{\mathsf{GV}} \mathcal{D} : R_2$, and $\Gamma_3, x' : \overline{S}, y' : \overline{S'} \vdash_{\mathsf{GV}} \mathcal{E} : R_3$.

The structurally-equivalent term $(\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})$ is not typeable in GV, since we cannot split both channels over a single parallel composition:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, x : S \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2 \qquad \Gamma_3, x' : \overline{S}, \langle y, y'\rangle : S'^\sharp \vdash_{\mathsf{GV}} \mathcal{E} : R_3}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp, \langle y, y'\rangle : S'^\sharp \vdash_{\mathsf{GV}} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, x'\rangle : S^\sharp \vdash_{\mathsf{GV}} (\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{\mathsf{GV}} (\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}$$

However, we *can* type this process in HGV:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x : S \vdash \mathcal{C} : R_1 \qquad \Gamma_2, y : S' \vdash \mathcal{D} : R_2}{\Gamma_1, x : S \parallel \Gamma_2, y : S' \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \qquad \Gamma_3, x' : \overline{S}, y' : \overline{S'} \vdash \mathcal{E} : R_3}{\Gamma_1, x : S \parallel \Gamma_2, y : S' \parallel \Gamma_3, x' : \overline{S}, y' : \overline{S'} \vdash (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, x : S \parallel \Gamma_2, \Gamma_3, x' : \overline{S} \vdash (\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu xx')(\nu yy')((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3}$$

Note in particular the shaded hyper-environment, which includes hyper-environment separators to separate endpoints x and x', as well as y and y'. It follows that, unlike in GV, *both* channels can be split over the same parallel composition. Similarly, the hyper-environment separator allows $\mathcal{C}$ and $\mathcal{D}$ to be composed *without* sharing any channels.

Although HGV types more processes, every well-typed HGV configuration typeable under a singleton hyper-environment Γ is *equivalent* to a well-typed GV configuration, which we show using tree canonical forms.

Proposition 4.5. *Suppose $\Gamma \vdash \mathcal{C} : R$ where $\mathcal{C}$ is in tree canonical form. Then, $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$.*

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C} : R$, making use of an inductive definition of tree canonical forms. See Appendix B for details. □

Remark 4.6. It is not the case that every HGV configuration typeable under an *arbitrary* hyper-environment $\mathcal{H}$ is equivalent to a well-typed GV configuration. This is because open HGV configurations can form *forest* process structures, whereas (even open) GV configurations must form a *tree* process structure.

Since we can write all well-typed HGV configurations in canonical form, and HGV tree canonical forms are typeable in GV, it follows that every well-typed HGV configuration typeable under a single type environment is equivalent to a well-typed GV configuration.

Typing rules for processes $\boxed{P \vdash \mathcal{G}}$

$$\text{TP-Link}\ \frac{}{x{\leftrightarrow}^{A}y \vdash x : A, y : A^{\perp}} \qquad \text{TP-New}\ \frac{P \vdash \mathcal{G} \parallel \Gamma, x : A \parallel \Delta, y : A^{\perp}}{(\nu xy)P \vdash \mathcal{G} \parallel \Gamma, \Delta} \qquad \text{TP-Par}\ \frac{P \vdash \mathcal{G} \qquad Q \vdash \mathcal{H}}{P \parallel Q \vdash \mathcal{G} \parallel \mathcal{H}}$$

$$\text{TP-Halt}\ \frac{}{\mathbf{0} \vdash \varnothing} \qquad \text{TP-Close}\ \frac{P \vdash \varnothing}{x[].P \vdash x : \mathbf{1}}$$

$$\text{TP-Wait}\ \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \perp} \qquad \text{TP-Send}\ \frac{P \vdash \Gamma, y : A \parallel \Delta, x : B}{x[y].P \vdash \Gamma, \Delta, x : A \otimes B} \qquad \text{TP-Recv}\ \frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \parr B}$$

$$\text{TP-Offer-Absurd}\ \frac{}{x \triangleright \{\} \vdash \Gamma, x : \top}$$

$$\text{TP-Select-Inl}\ \frac{P \vdash \Gamma, x : A}{x \triangleleft \mathrm{inl}.P \vdash \Gamma, x : A \oplus B} \qquad \text{TP-Select-Inr}\ \frac{P \vdash \Gamma, x : B}{x \triangleleft \mathrm{inr}.P \vdash \Gamma, x : A \oplus B} \qquad \text{TP-Offer}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \vdash \Gamma, x : A \,\&\, B}$$

Duality $\boxed{A^{\perp}}$

$$\begin{array}{llllllllllll} (A \otimes B)^{\perp} & = & A^{\perp} \parr B^{\perp} & (\mathbf{1})^{\perp} & = & \perp & (A \oplus B)^{\perp} & = & A^{\perp} \,\&\, B^{\perp} & (\mathbf{0})^{\perp} & = & \top \\ (A \parr B)^{\perp} & = & A^{\perp} \otimes B^{\perp} & (\perp)^{\perp} & = & \mathbf{1} & (A \,\&\, B)^{\perp} & = & A^{\perp} \oplus B^{\perp} & (\top)^{\perp} & = & \mathbf{0} \end{array}$$

Figure 6. HCP, duality and typing rules for processes.

Corollary 4.7. *If* $\Gamma \vdash \mathcal{C} : R$*, then there exists some* $\mathcal{D}$ *such that* $\mathcal{C} \equiv \mathcal{D}$ *and* $\Gamma \vdash_{\mathsf{GV}} \mathcal{D} : R$*.*

5. Relation between HGV and HCP

In this section, we explore two translations, from HGV to HCP and from HCP to HGV, together with their operational correspondence results.

Hypersequent CP. HCP [MP18, KMP19b] is a session-typed process calculus with a correspondence to CLL, which exploits hypersequents to fix extensibility and modularity issues with CP.

Types (A, B) consist of the connectives of linear logic: the multiplicative operators ($\otimes$, $\parr$) and units ($\mathbf{1}$, $\perp$) and the additive operators ($\oplus$, $\&$) and units ($\mathbf{0}$, $\top$).

$$A, B ::= \mathbf{1} \mid \perp \mid \mathbf{0} \mid \top \mid A \otimes B \mid A \parr B \mid A \oplus B \mid A \,\&\, B$$

Type environments (Γ, Δ) associate names with types. Hyper-environments ($\mathcal{G}$, $\mathcal{H}$) are collections of type environments. The empty type environment and hyper-environment are written $\cdot$ and $\varnothing$, respectively. Names in type and hyper-environments must be unique and environments may be combined, written Γ, Δ and $\mathcal{G} \parallel \mathcal{H}$, only if they are disjoint.

Processes (P, Q) are a variant of the π-calculus with forwarding [San96, Bor98], bound output [San96], and double binders [Vas12]. The syntax of processes is given by the typing rules (Figure 6), which are standard for HCP [MP18, KMP19b]: $x{\leftrightarrow}^{A}y$ forwards messages between x and y; $(\nu xy)P$ creates a channel with endpoints x and y, and continues as P; $P \parallel Q$ composes P and Q in parallel; $\mathbf{0}$ is the terminated process; $x[y].P$ creates a new channel, outputs one endpoint over x, binds the other to y, and continues as P; $x(y).P$ receives a channel endpoint, binds it to y, and continues as P; $x[].P$ and $x().P$ close x and continue as P; $x \triangleleft \mathrm{inl}.P$ and $x \triangleleft \mathrm{inr}.P$ make a binary choice; $x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}$ offers a binary choice; and $x \triangleright \{\}$ offers a nullary choice. As HCP is synchronous, the only difference between $x[y].P$ and $x(y).P$ is their typing (and similarly for $x[].P$ and $x().P$). We write *unbound* send as $x\langle y\rangle.P$ (short for $x[z].(y{\leftrightarrow}z \parallel P)$), and synchronisation as $\bar{x}.P$ (short for

Action rules

$$\textsc{Act-Pref}\quad \pi.P \xrightarrow{\pi} P \qquad \textsc{Act-Link}_1\quad x{\leftrightarrow}y \xrightarrow{x\leftrightarrow y} \mathbf{0} \qquad \textsc{Act-Link}_2\quad x{\leftrightarrow}y \xrightarrow{y\leftrightarrow x} \mathbf{0} \qquad \textsc{Act-Off-Inl}\quad x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \xrightarrow{x\triangleright\mathrm{inl}} P$$

$$\textsc{Act-Off-Inr}\quad x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \xrightarrow{x\triangleright\mathrm{inr}} Q$$

Communication Rules

$$\textsc{Alp-Link}\quad \frac{P \xrightarrow{x\leftrightarrow z} P'}{(\nu xy)P \xrightarrow{\alpha} P'\{z/y\}} \qquad \textsc{Bet-Send}\quad \frac{P \xrightarrow{x[x']\|y(y')} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)(\nu x'y')P'} \qquad \textsc{Bet-Close}\quad \frac{P \xrightarrow{x[]\|y()} P'}{(\nu xy)P \xrightarrow{\beta} P'}$$

$$\textsc{Bet-Inl}\quad \frac{P \xrightarrow{x\triangleleft\mathrm{inl}\|y\triangleright\mathrm{inl}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'} \qquad \textsc{Bet-Inr}\quad \frac{P \xrightarrow{x\triangleleft\mathrm{inr}\|y\triangleright\mathrm{inr}} P'}{(\nu xy)P \xrightarrow{\beta} (\nu xy)P'}$$

Structural Rules

$$\textsc{Str-Res}\quad \frac{P \xrightarrow{\ell} P' \qquad x, y \notin \mathrm{cn}(\ell)}{(\nu xy)P \xrightarrow{\ell} (\nu xy)P'} \qquad \textsc{Str-Par}_1\quad \frac{P \xrightarrow{\ell} P' \qquad \mathrm{bn}(\ell) \cap \mathrm{fn}(Q) = \varnothing}{P \parallel Q \xrightarrow{\ell} P' \parallel Q}$$

$$\textsc{Str-Par}_2\quad \frac{Q \xrightarrow{\ell} Q' \qquad \mathrm{bn}(\ell) \cap \mathrm{fn}(P) = \varnothing}{P \parallel Q \xrightarrow{\ell} P \parallel Q'} \qquad \textsc{Str-Syn}\quad \frac{P \xrightarrow{\ell} P' \qquad Q \xrightarrow{\ell'} Q' \qquad \mathrm{bn}(\ell) \cap \mathrm{bn}(\ell') = \varnothing}{P \parallel Q \xrightarrow{l\|l'} P' \parallel Q'}$$

FIGURE 7. HCP, label transition semantics.

$x[z].(z[].\mathbf{0} \parallel P)$) and $x.P$ (short for $x(z).z().P$). Duality is standard and is involutive, *i.e.*, $(A^{\perp})^{\perp} = A$.

We define a standard structural congruence ($\equiv$) similar to that of HGV, *i.e.*, parallel composition is commutative and associative, we can commute name restrictions, swap the order of endpoints, swap links, and have scope extrusion (similar to Figure 4). Note that since we base our formal developments on an LTS semantics, structural congruence is not required for reduction.

$$x{\leftrightarrow}^{A}y \equiv y{\leftrightarrow}^{A^{\perp}}x \qquad P \parallel \mathbf{0} \equiv P \qquad P \parallel Q \equiv Q \parallel P \qquad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

$$(\nu xx')(\nu yy')P \equiv (\nu yy')(\nu xx')P \qquad (\nu xy)P \equiv (\nu yx)P$$

$$(\nu xy)(P \parallel Q) \equiv P \parallel (\nu xy)Q \quad \text{if } x, y \notin \mathrm{fv}(P)$$

We define the labelled transition system for HCP as a small refinement of the LTS for the additive-multiplicative fragment of the πLL calculus introduced by Montesi and Peressotti [MP21], in turn inspired by their previous system *CT* [MP18]. The LTS is identical, save for the fact that we distinguish two types of internal actions. Action labels l represent the actions that a process can fire. Prefixes π are a convenient subset of action labels which can be written as prefixes to processes, *i.e.*, $\pi.P$. Transition labels ℓ include

action labels and the parallel composition of two action labels, along with *internal* actions α, β, and τ. The LTS gives rise to two types of internal action: α represents only the evaluation of links as *renaming*, and β represents only *communication*. Labels τ arise only due to saturated transition (Definition 5.4) and are not produced by the rules in the LTS.

$$\begin{array}{lrcl} \text{Prefixes} & \pi & ::= & x[y] \mid x[] \mid x(y) \mid x() \mid x \triangleleft \mathrm{inl} \mid x \triangleleft \mathrm{inr} \\ \text{Action Labels} & l & ::= & \pi \mid x \leftrightarrow y \mid x \triangleright \mathrm{inl} \mid x \triangleright \mathrm{inr} \\ \text{Transition Labels} & \ell & ::= & l \mid l \parallel l' \mid \alpha \mid \beta \end{array}$$

We let ℓ_x range over labels on x: $x \leftrightarrow y$, $x[y]$, $x[]$, *etc.* Labelled transition $\xrightarrow{\ell}$ is defined in Figure 7. We write $\xrightarrow{\ell}\xrightarrow{\ell'}$ for the composition of $\xrightarrow{\ell}$ and $\xrightarrow{\ell'}$, $\xrightarrow{\ell^+}$ for the transitive closure of $\xrightarrow{\ell}$, and $\xrightarrow{\ell^*}$ for the reflexive-transitive closure of $\xrightarrow{\ell}$. We write $\mathrm{bn}(\ell)$ and $\mathrm{fn}(\ell)$ for the bound and free names contained in ℓ, respectively. We write $\mathrm{cn}(\ell)$ for all names in ℓ, *i.e.*, $\mathrm{cn}(\ell) = \mathrm{fn}(\ell) \cup \mathrm{bn}(\ell)$.

Metatheory. Transitions preserve typeability. Since internal actions occur only under binders, they are typable under the same hyper-environment.

Theorem 5.1 (Type Preservation)**.** *Suppose $P \vdash \mathcal{G}$ and $P \xrightarrow{\ell} Q$.*

- *If ℓ is internal, then $Q \vdash \mathcal{G}$.*
- *If ℓ is not internal, then there exists some $\mathcal{H}$ such that $Q \vdash \mathcal{H}$.*

Proof. Following the approach of [KMP19a, MP18, MP21], type preservation is established by defining proof transformations on typing derivations of each reducing process. The only difference with respect to [MP18, MP21] arises due to our separate treatment of α and β actions, which does not materially impact the proof. □

Similarly, our LTS for HCP satisfies progress. Following [KMP19a, MP21], the key intermediate step is to note that for every type environment in a hyper-environment, there is some free name which can be acted upon. Again, the stratification of internal actions does not materially impact the proof.

Theorem 5.2 (Progress)**.** *If $P \vdash \mathcal{H}$ and $P \not\equiv \mathbf{0}$, then there exist some ℓ, Q such that $P \xrightarrow{\ell} Q$.*

Behavioural Theory. The behavioural theory for HCP follows Kokke *et al.* [KMP19a], except that we distinguish two subrelations of weak bisimilarity, following the subtypes of internal actions.

Definition 5.3 (Strong bisimulation and strong bisimilarity)**.** A symmetric relation $\mathcal{R}$ on processes is a *strong bisimulation* if $P \mathcal{R} Q$ implies that if $P \xrightarrow{\ell} P'$, then $Q \xrightarrow{\ell} Q'$ for some Q' such that $P' \mathcal{R} Q'$. *Strong bisimilarity* is the largest relation $\sim$ that is a strong bisimulation.

Definition 5.4 (Saturated transition)**.** The *$\mathcal{L}$-saturated transition relation, for $\mathcal{L} \subseteq \{\alpha, \beta\}$, is the smallest relation $\Longrightarrow_{\mathcal{L}}$ closed under the following rules, with saturated transition labels ℓ ranging over transition labels and the distinguished label τ:*

$$\frac{}{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P} \qquad \frac{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P' \xrightarrow{\ell} Q' \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q \qquad \ell \in \mathcal{L}}{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q} \qquad \frac{P \overset{\tau}{\Longrightarrow}_{\mathcal{L}} P' \xrightarrow{\ell} Q' \overset{\tau}{\Longrightarrow}_{\mathcal{L}} Q \qquad \ell \notin \mathcal{L}}{P \overset{\ell}{\Longrightarrow}_{\mathcal{L}} Q}$$

We write $\Longrightarrow_\ell$ *as shorthand for* $\Longrightarrow_{\{\ell\}}$*, and we write* $\Longrightarrow$ *as shorthand for* $\Longrightarrow_{\{\alpha,\beta\}}$*.*

Definition 5.5 (Weak bisimulation and weak bisimilarity)**.** A symmetric relation $\mathcal{R}$ on processes is an $\mathcal{L}$-bisimulation, for $\mathcal{L} \subseteq \{\alpha, \beta\}$, if $P\,\mathcal{R}\,Q$ implies that if $P \overset{\ell'}{\Longrightarrow}_{\mathcal{L}} P'$, then $Q \overset{\ell'}{\Longrightarrow}_{\mathcal{L}} Q'$ for some Q' such that $P'\,\mathcal{R}\,Q'$. The $\mathcal{L}$-bisimilarity relation is the largest relation $\approx_{\mathcal{L}}$ that is an $\mathcal{L}$-bisimulation. We write $\approx$ as shorthand for $\approx_{\{\alpha,\beta\}}$.

Lemma 5.6. *Structural congruence, strong bisimilarity and the various forms of weak bisimilarity are related as follows:*

$$\equiv \subset \sim \qquad \sim \subset \approx \qquad \sim \subset \approx_\alpha \qquad \sim \subset \approx_\beta$$

Differences with previous version. The LTS in Figure 7 is similar to that in the previous version of this work [FKD+21], with the exception that we have removed the rules Tau-Alp and Tau-Bet:

$$\textsc{Tau-Alp}\ \not{\frac{P \xrightarrow{\alpha} P'}{P \xrightarrow{\tau} P'}} \qquad \textsc{Tau-Bet}\ \not{\frac{P \xrightarrow{\beta} P'}{P \xrightarrow{\tau} P'}}$$

To see why these rules are problematic, consider processes $P = (\nu xy)(z{\leftrightarrow}x \parallel y[].\mathbf{0})$ and $Q = z[].\mathbf{0}$. Following Definition 5.5, P and Q are α-bisimilar, as P only has the α-transition $P \xrightarrow{\alpha} Q$ and Q has no transitions. In the previous version, Tau-Alp gave P the derived τ-transition $P \xrightarrow{\tau} Q$, which meant that $P \not\approx_\alpha Q$, as $Q \overset{\tau}{\not\Longrightarrow} Q$. Therefore Tau-Alp collapses $\approx_\alpha$ to $\sim$ and Tau-Bet collapses $\approx_\beta$ to $\sim$.

The solution we adopted was to remove Tau-Alp and Tau-Bet from the label transition relation $\longrightarrow$, and instead lift α- and β-transitions to τ-transitions in the definition of saturated transition[1].

Translating HGV to HCP. We factor the translation from HGV to HCP into two translations: (1) a translation into HGV$*$, a fine-grain call-by-value [LPT03] variant of HGV, which makes control flow explicit; and (2) a translation from HGV$*$ to HCP. In so doing, we can concentrate on the essence of the translations as opposed to concerning ourselves with administrative reductions.

HGV$*$. We define HGV$*$ as a refinement of HGV in which any non-trivial term must be named by a let-binding before being used. While **let** is syntactic sugar in HGV, it is part of the core language in HGV$*$. Correspondingly, the reduction rule for **let** follows from the encoding in HGV, *i.e.*, $\mathbf{let}\ x = V\ \mathbf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\}$.

Terms	L, M, N	::=	$V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid V\ W$
			$\mid \mathbf{let}\ () = V\ \mathbf{in}\ M \mid \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M$
			$\mid \mathbf{absurd}\ V \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$
Values	V, W	::=	$x \mid K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$
Evaluation contexts	E	::=	$\Box \mid \mathbf{let}\ x = E\ \mathbf{in}\ M$
Thread contexts	$\mathcal{F}$	::=	$\phi\ E$

[1]We thank Marco Peressotti for notifying us of the error and suggesting the fix.

Remark 5.7. Fine-grain call-by-value λ-calculi typically include an explicit $\mathbf{return}\, V$ construct to embed values into the term language. As there is no difference between the shapes of the value and term typing judgements, we allow ourselves to embed values directly for simplicity.

We can *naïvely* translate HGV to HGV$*$ ($(\!|\cdot|\!)$) by let-binding each subterm in a value position, *e.g.*, $(\!|\mathbf{inl}\ M|\!) = \mathbf{let}\ z = (\!|M|\!)\ \mathbf{in}\ \mathbf{inl}\ z$.

Definition 5.8 (Naïve translation of HGV to HGV$*$)**.**

$$
\begin{array}{lcl}
(\!|x|\!) & = & x \\
(\!|\lambda x.M|\!) & = & \lambda x.(\!|M|\!) \\
(\!|L\ M|\!) & = & \mathbf{let}\ x = (\!|L|\!)\ \mathbf{in}\ \mathbf{let}\ y = (\!|M|\!)\ \mathbf{in}\ x\ y \\
(\!|()|\!) & = & () \\
(\!|\mathbf{let}\ () = L\ \mathbf{in}\ M|\!) & = & \mathbf{let}\ z = (\!|L|\!)\ \mathbf{in}\ \mathbf{let}\ () = z\ \mathbf{in}\ (\!|M|\!) \\
(\!|(M, N)|\!) & = & \mathbf{let}\ x = (\!|M|\!)\ \mathbf{in}\ \mathbf{let}\ y = (\!|N|\!)\ \mathbf{in}\ (x, y) \\
(\!|\mathbf{let}\ (x, y) = L\ \mathbf{in}\ M|\!) & = & \mathbf{let}\ z = (\!|L|\!)\ \mathbf{in}\ \mathbf{let}\ (x, y) = z\ \mathbf{in}\ (\!|M|\!) \\
(\!|\mathbf{inl}\ M|\!) & = & \mathbf{let}\ z = (\!|M|\!)\ \mathbf{in}\ \mathbf{inl}\ z \\
(\!|\mathbf{inr}\ M|\!) & = & \mathbf{let}\ z = (\!|M|\!)\ \mathbf{in}\ \mathbf{inr}\ z \\
(\!|\mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}|\!) & = & \mathbf{let}\ z = (\!|L|\!)\ \mathbf{in}\ \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto (\!|M|\!);\ \mathbf{inr}\ y \mapsto (\!|N|\!)\} \\
(\!|\mathbf{absurd}\ L|\!) & = & \mathbf{let}\ z = (\!|L|\!)\ \mathbf{in}\ \mathbf{absurd}\ z
\end{array}
$$

Standard techniques can be used to avoid administrative redexes [Plo75, DMN07]. We give a full definition of HGV$*$ in Appendix C.

HGV$*$ to HCP. The translation from HGV$*$ to HCP is given in Figure 8. All control flow is encapsulated in values and let-bindings. We define a pair of translations on types, $\llbracket\cdot\rrbracket$ and $\llparenthesis\cdot\rrparenthesis$, such that $\llbracket T\rrbracket = \llparenthesis T\rrparenthesis^{\perp}$. We extend these translations pointwise to type environments and hyper-environments. We define translations on configurations ($\llparenthesis\cdot\rrparenthesis^{\mathsf{c}}_r$), terms ($\llparenthesis\cdot\rrparenthesis^{\mathsf{m}}_r$) and values ($\llparenthesis\cdot\rrparenthesis^{\mathsf{v}}_r$), where r is a fresh name denoting a distinguished output channel.

We translate an HGV sequent $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$ as $\llparenthesis\mathcal{C}\rrparenthesis^{\mathsf{c}}_r \vdash \llbracket\mathcal{G}\rrbracket \parallel \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^{\perp}$, where Γ is the type environment corresponding to the main thread. The translation of computations includes synchronisation action in order to faithfully simulate a call-by-value reduction strategy. The (term) translation of a value $\llparenthesis V\rrparenthesis^{\mathsf{m}}_r$ immediately pings the output channel r to announce that it is a value. The translation of a let-binding $\llparenthesis\mathbf{let}\ w = M\ \mathbf{in}\ N\rrparenthesis^{\mathsf{m}}_r$ first evaluates M to a value, which then pings the internal channel x/x' and unblocks the continuation $x.\llparenthesis N\rrparenthesis^{\mathsf{m}}_r$. The translations of main and child threads each make use of an internal result channel. The translation of a child thread consumes the yielded unit endpoint once the child thread has terminated. The translation of the main thread forwards the result value along the external output channel once the main thread has terminated.

There are two changes with respect to the translation of our earlier paper [FKD$^+$21]. First, in the earlier work the translation of the main thread output directly to the external output channel instead of forwarding via an intermediary as in the current translation. This change is purely aesthetic. Second, in the earlier work the translation of **fork** was not sufficiently concurrent. Correspondingly there was an error in the case of the operational correspondence proof which is fixed in the current paper.

Lemma 5.9 (Type Preservation)**.**

(1) *If* $\Gamma \vdash V : T$*, then* $\llparenthesis V\rrparenthesis^{\mathsf{v}}_r \vdash \llbracket\Gamma\rrbracket, r : \llbracket T\rrbracket^{\perp}$.

(2) *If* $\Gamma \vdash M : T$*, then* $\llparenthesis M\rrparenthesis^{\mathsf{m}}_r \vdash \llbracket\Gamma\rrbracket, r : \mathbf{1} \otimes \llbracket T\rrbracket^{\perp}$.

Translation on types $\boxed{\llbracket T \rrbracket \text{ and } \llparenthesis T \rrparenthesis}$

$$\begin{array}{lll} \llbracket !T.S \rrbracket = \llbracket T \rrbracket^{\perp} \otimes \llbracket S \rrbracket & \llbracket \mathbf{end}_! \rrbracket = \mathbf{1} & \llbracket T \rrbracket = \llparenthesis T \rrparenthesis^{\perp}, \\ \llbracket ?T.S \rrbracket = \llbracket T \rrbracket^{\perp} \parr \llbracket S \rrbracket & \llbracket \mathbf{end}_? \rrbracket = \bot & \text{if } T \text{ is not a session type} \end{array}$$

$$\begin{array}{lll} \llparenthesis T \times U \rrparenthesis = \llparenthesis T \rrparenthesis \otimes \llparenthesis U \rrparenthesis & \llparenthesis \mathbf{1} \rrparenthesis = \mathbf{1} & \llparenthesis T \multimap U \rrparenthesis = \llparenthesis T \rrparenthesis^{\perp} \parr (\mathbf{1} \otimes \llparenthesis U \rrparenthesis) \\ \llparenthesis T + U \rrparenthesis = \llparenthesis T \rrparenthesis \oplus \llparenthesis U \rrparenthesis & \llparenthesis \mathbf{0} \rrparenthesis = \mathbf{0} & \llparenthesis S \rrparenthesis = \llbracket S \rrbracket^{\perp} \end{array}$$

Translation on configurations, terms, and values $\boxed{\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r, \llbracket M \rrbracket^{\mathsf{m}}_r, \text{ and } \llbracket V \rrbracket^{\mathsf{v}}_r}$

$$\begin{array}{lll} \llbracket \circ\, M \rrbracket^{\mathsf{c}}_r = (\nu y y')(\llbracket M \rrbracket^{\mathsf{m}}_y \parallel y'.y'[].\mathbf{0}) & \llbracket (\nu x x')\mathcal{C} \rrbracket^{\mathsf{c}}_r = (\nu x x')\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r & \llbracket x \overset{z}{\leftrightarrow} y \rrbracket^{\mathsf{c}}_r = \bar{z}.z().x \leftrightarrow y \\ \llbracket \bullet\, M \rrbracket^{\mathsf{c}}_r = (\nu y y')(\llbracket M \rrbracket^{\mathsf{m}}_y \parallel y'.y' \leftrightarrow r) & \llbracket \mathcal{C} \parallel \mathcal{D} \rrbracket^{\mathsf{c}}_r = \llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \parallel \llbracket \mathcal{D} \rrbracket^{\mathsf{c}}_r & \end{array}$$

$$\begin{array}{lll} \llbracket x \rrbracket^{\mathsf{v}}_r = r \leftrightarrow x & \llbracket () \rrbracket^{\mathsf{v}}_r = r[].\mathbf{0} & \llbracket \mathbf{inl}\ V \rrbracket^{\mathsf{v}}_r = r \triangleleft \mathrm{inl}.\llbracket V \rrbracket^{\mathsf{v}}_r \\ \llbracket \lambda x.M \rrbracket^{\mathsf{v}}_r = r(x).\llbracket M \rrbracket^{\mathsf{m}}_r & \llbracket (V, W) \rrbracket^{\mathsf{v}}_r = r[x].(\llbracket V \rrbracket^{\mathsf{v}}_x \parallel \llbracket W \rrbracket^{\mathsf{v}}_r) & \llbracket \mathbf{inr}\ V \rrbracket^{\mathsf{v}}_r = r \triangleleft \mathrm{inr}.\llbracket V \rrbracket^{\mathsf{v}}_r \end{array}$$

$$\begin{array}{ll} \llbracket V\ W \rrbracket^{\mathsf{m}}_r & = (\nu x x')(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'} \parallel \llbracket W \rrbracket^{\mathsf{v}}_{x'}) \\ \llbracket \mathbf{let}\ () = V\ \mathbf{in}\ M \rrbracket^{\mathsf{m}}_r & = (\nu x x')(x().\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \\ \llbracket \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M \rrbracket^{\mathsf{m}}_r & = (\nu y y')(y(x).\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{y'}) \\ \llbracket \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \rrbracket^{\mathsf{m}}_r & = (\nu x x')(x \triangleright \{\mathrm{inl} : \llbracket M \rrbracket^{\mathsf{m}}_r; \mathrm{inr} : \llbracket N\{x/y\} \rrbracket^{\mathsf{m}}_r\} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \\ \llbracket \mathbf{absurd}\ V \rrbracket^{\mathsf{m}}_r & = (\nu x x')(x \triangleright \{\} \parallel \llbracket V \rrbracket^{\mathsf{v}}_{x'}) \\ \llbracket \mathbf{let}\ x = M\ \mathbf{in}\ N \rrbracket^{\mathsf{m}}_r & = (\nu x x')(x.\llbracket N \rrbracket^{\mathsf{m}}_r \parallel \llbracket M \rrbracket^{\mathsf{m}}_{x'}) \\ \llbracket V \rrbracket^{\mathsf{m}}_r & = \bar{r}.\llbracket V \rrbracket^{\mathsf{v}}_r \end{array}$$

$$\begin{array}{ll} \llbracket \mathbf{link} \rrbracket^{\mathsf{v}}_r & = r(y).y(x).\bar{r}.r().x \leftrightarrow y \\ \llbracket \mathbf{fork} \rrbracket^{\mathsf{v}}_r & = (\nu y y')(r(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \parallel y'(x).x\langle y' \rangle.x.x[].\mathbf{0}) \\ \llbracket \mathbf{send} \rrbracket^{\mathsf{v}}_r & = r(y).y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \\ \llbracket \mathbf{recv} \rrbracket^{\mathsf{v}}_r & = r(x).x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x \\ \llbracket \mathbf{wait} \rrbracket^{\mathsf{v}}_r & = r(x).x().\bar{r}.r[].\mathbf{0} \end{array}$$

FIGURE 8. Translation from HGV$*$ to HCP.

(3) *If* $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$, *where* Γ *is the type environment for the main thread in* $\mathcal{C}$, *then* $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \vdash \llparenthesis \mathcal{G} \rrparenthesis \parallel \llparenthesis \Gamma \rrparenthesis, r : \llparenthesis T \rrparenthesis^{\perp}$.

Lemma 5.10 (Substitution)**.** *If* M *is a well-typed term with* $w \in \mathrm{fv}(M)$, *and* V *is a well-typed value, then* $(\nu w w')(\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{w'}) \approx_\alpha \llbracket M\{V/w\} \rrbracket^{\mathsf{m}}_r$.

Theorem 5.11 (Operational Correspondence)**.** *Suppose* $\mathcal{C}$ *is a well-typed configuration.*

(1) *(Preservation of reductions) If* $\mathcal{C} \longrightarrow \mathcal{C}'$, *then there exists a* P *such that* $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \overset{\beta^+}{\Longrightarrow}_\alpha P$ *and* $P \approx_\alpha \llbracket \mathcal{C}' \rrbracket^{\mathsf{c}}_r$; *and*

(2) *(Reflection of transitions)*
- *if* $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \xrightarrow{\alpha} P$, *then* $P \approx_\alpha \llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r$; *and*
- *if* $\llbracket \mathcal{C} \rrbracket^{\mathsf{c}}_r \xrightarrow{\beta} P$, *then there exists a* $\mathcal{C}'$ *and a* P' *such that* $\mathcal{C} \longrightarrow \mathcal{C}'$ *and* $P \overset{\beta^*}{\Longrightarrow}_\alpha P'$ *and* $P' \approx_\alpha \llbracket \mathcal{C}' \rrbracket^{\mathsf{c}}_r$. *Furthermore,* $\mathcal{C}'$ *is unique up to structural congruence.*

The proof is in Appendix C. One might strive for a tighter operational correspondence here, but our current translation generates multiple administrative β-transitions. The only term reduction that translates to multiple β-transitions is the one for let-bindings. This is because we choose to encode synchronisation using two β-transitions. We could adjust the accounting here by treating synchronisation as a single β-transition or its own special kind of

administrative transition. Many more administrative reductions arise from the configuration translation. These are due to a combination of synchronisations and also the fact that we use constants along with pairs and application for our communication primitives instead of building-in fully-applied communication primitives.

Translating HCP to HGV. We cannot translate HCP processes to HGV terms directly: HGV's term language only supports **fork** (see Section 7 for further discussion), so there is no way to translate an individual name restriction or parallel composition. However, we can still translate HCP into HGV via the composition of known translations.

HCP into CP: We must first reunite each parallel composition with its corresponding name restriction, *i.e.*, translate to CP using the *disentanglement* translation shown by Kokke *et al.* [KMP19b, Lemma 4.7]. The result is a collection of independent CP processes.

CP into GV: Next, we can translate each CP process into a GV configuration using (a variant of) Lindley and Morris' translation [LM15, Figure 8].

GV into HGV: Finally, we can use our embedding of GV into HGV (Theorem 4.3) to obtain a collection of well-typed HGV configurations, which can be composed using TC-Par to result in a single well-typed HGV configuration.

The translation from HCP into CP and the embedding of GV into HGV preserve and reflect reduction. However, as previously mentioned, Lindley and Morris's original translation from CP to GV preserves but does *not* reflect reduction due to an asynchronous encoding of choice. By adapting their translation to use a synchronous encoding of choice (Section 3), we obtain a translation from CP to GV that both preserves and reflects reduction. Thus, composing all three translations together we obtain a translation from HCP to HGV that preserves and reflects reduction.

6. Extensions

In this section, we outline three extensions to HGV that exploit generalising the tree structure of processes to a forest structure. These extensions are of particular interest since HGV already supports a core aspect of forest structure, enabling its full utilisation merely through the addition of a structural rule. In contrast, to extend GV with forest structure one must distinguish two distinct introduction rules for parallel composition [LM15, Fow19]. Other extensions to GV such as shared channels [LM15], polymorphism [LM17], and recursive session types [LM16] adapt to HGV almost unchanged.

From trees to forests. The TC-Par rule allows two processes to be composed in parallel if they are typeable under separate hyper-environments. In a closed program, hyper-environment separators are introduced by TC-Res, meaning that each process must be connected by a channel.

The following TC-Mix rule allows two type environments Γ_1, Γ_2 to be split by a hyper-environment separator *without* a channel connecting them, and is inspired by Girard's Mix rule [Gir87]; in the concurrent setting, Mix can be interpreted as concurrency *without* communication [LM15, ALM16]. TC-Mix admits a much simpler treatment of **link** and provides a crucial ingredient for handling exceptional behaviour.

$$\textsc{TC-Mix}\ \frac{\mathcal{G} \parallel \Gamma_1 \parallel \Gamma_2 \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \mathcal{C} : R}$$

Atkey *et al.* [ALM16] show that conflating the $\mathbf{1}$ and $\bot$ types in CP (which correspond respectively to the $\mathbf{end}_!$ and $\mathbf{end}_?$ types in GV) is logically equivalent to adding the Mix rule and a 0-Mix rule (used to type an empty process). It follows that in the presence of TC-Mix, we use self-dual $\mathbf{end}$ type; in the GV setting, by using a self-dual $\mathbf{end}$ type, we decouple closing a channel from process termination. We therefore refine the TC-Child rule and the type schema for $\mathbf{fork}$ to ensure that each child thread returns the unit value, and replace the $\mathbf{wait}$ constant with a $\mathbf{close}$ constant which eliminates an endpoint of type $\mathbf{end}$.

$$\mathbf{fork} : (S \multimap \mathbf{1}) \multimap \overline{S} \qquad \mathbf{close} : \mathbf{end} \multimap \mathbf{1} \qquad \textsc{TC-Child}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \circ M : \circ}$$

$$\textsc{E-Close}\ (\nu xy)(E[\mathbf{close}\ x] \parallel E'[\mathbf{close}\ y]) \longrightarrow E[()] \parallel E'[()]$$

Given TC-Mix, we might expect a term-level construct $\mathbf{spawn} : (\mathbf{1} \multimap \mathbf{1}) \multimap \mathbf{1}$ which spawns a parallel thread without a connecting channel. We can encode such a construct using $\mathbf{fork}$ and $\mathbf{close}$ (assuming fresh x and y):

$$\mathbf{spawn}\ M \triangleq \mathbf{let}\ x = \mathbf{fork}(\lambda y.\mathbf{close}\ y;\ M)\ \mathbf{in}\ \ \mathbf{close}\ x$$

Assuming the encoded $\mathbf{spawn}$ is running in a main thread, after two reduction steps, we are left with the configuration:

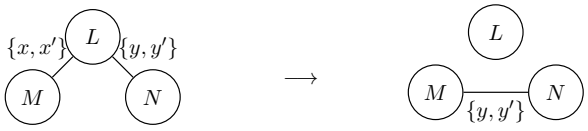
$$\dfrac{\dfrac{\dfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \circ M : \circ}\ \textsc{TC-Child} \qquad \dfrac{\cdot \vdash M : \mathbf{1}}{\cdot \vdash \bullet() : \mathbf{1}}\ \textsc{TC-Main}}{\cdot \parallel \cdot \vdash \circ M \parallel \bullet() : \mathbf{1}}\ \textsc{TC-Par}}{\cdot \vdash \circ M \parallel \bullet() : \mathbf{1}}\ \textsc{TC-Mix}$$

Note the essential use of TC-Mix to insert a hyper-environment separator.

The addition of TC-Mix does not affect preservation or progress. The result follows from routine adaptations of the proof of Theorem 3.3 and Theorem 3.20.

By relaxing the tree process structure restriction using TC-Mix, we can obtain a more efficient treatment of $\mathbf{link}$, and can support the treatment of exceptions advocated by Fowler *et al.* [FLMD19].

A simpler link. The $\mathbf{link}\ (x, y)$ construct forwards messages from x to y and vice-versa. Consider threads $L = F[\mathbf{link}\ (x, y)]$, M, N, where L connects to M by x and to N by y.



The result of link reduction has forest structure. Well-typed closed programs in both GV and HGV must *always* maintain tree structure. Different versions of GV do so in various unsatisfactory ways: one is pre-emptive blocking [LM15], which breaks confluence; another is two-stage linking (Figure 4), which defers forwarding via a special link thread [LM16].

Lindley and Morris [LM15] implement **link** using the following rule (modified here to use a double-binder formulation):

$$(\nu x x')(F[\mathbf{link}\ (x,y)] \parallel \mathcal{F}'[M]) \longrightarrow (\nu x x')(F[x] \parallel \mathcal{F}'[\mathbf{wait}\ x'; M\{y/x'\}]) \quad \text{where } x' \in \mathrm{fv}(M)$$

The first thread will eventually reduce to $\circ x$, at which point the second thread will synchronise to eliminate x and x' and then evaluate the continuation M with endpoint y substituted for x'. Unfortunately, this formulation of **link** preemptively inhibits reduction in the second thread, since the evaluation rule inserts a blocking **wait**. The resulting system does not satisfy the diamond property.

HGV uses the incarnation of **link** advocated by Lindley and Morris [LM16], where linking is split into two stages: the first generates a fresh pair of endpoints z, z' and a link thread of the form $x \overset{z'}{\leftrightarrow} y$, and returns z to the calling thread. Once the calling thread has evaluated to a value (which must by typing be z), then the link substitution can take place. This formulation recovers confluence, but we still lose a degree of concurrency: communication on y is blocked until the linking thread has fully evaluated. In an ideal implementation, the behaviour of the linking thread would be irrelevant to the remainder of the configuration. The operation requires additional runtime syntax and thus complicates the metatheory.

The above issues are symptomatic of the fact that the process structure after a link takes place is a forest rather than a tree. However, with TC-Mix, we can refine the type schema for **link** to $(S \times \overline{S}) \multimap \mathbf{1}$ and we can use the following rule:

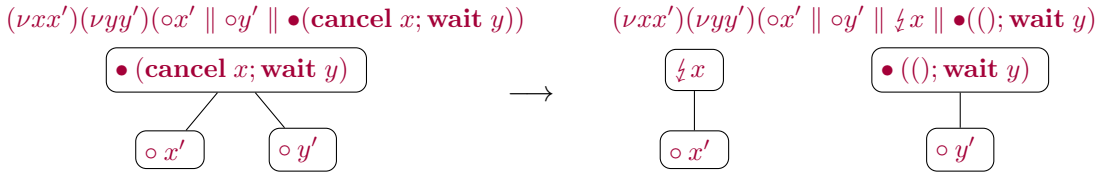
$$(\nu x x')(F[\mathbf{link}\ (x,y)] \parallel \phi N) \longrightarrow F[()] \parallel \phi N\{y/x'\}$$

This formulation enables immediate substitution, maximimising concurrency. A variant of HGV replacing E-Reify-Link and E-Comm-Link with E-Link-Mix retains HGV's metatheory.

Exceptions. In order to support exceptions in the presence of linear endpoints [FLMD19, MV18] we must have a way of *cancelling* an endpoint. Mostrous and Vasconcelos [MV18] describe a process calculus allowing the *explicit cancellation* of a channel endpoint, accounting for exceptional scenarios such as a client disconnecting, or a thread encountering an unrecoverable error. Attempting to communicate with a cancelled endpoint raises an exception. Fowler *et al.* [FLMD19] extend these ideas to the functional setting, introducing Exceptional GV (EGV). EGV supports exceptional behaviour by adding three term-level constructs:

- a new constant, **cancel** : $S \multimap \mathbf{1}$, which allows us to discard an arbitrary session endpoint with type S
- a construct **raise**, which raises an exception
- an exception handling construct **try** L **as** x **in** M **otherwise** N in the style of Benton & Kennedy [BK01], which attempts possibly-failing computation L, binding the result to x in success continuation M if successful and evaluating N if an exception is raised

Cancellation generates a *zapper thread* ($\lightning\, x$) which severs a tree topology into a forest as in the following example.



The configuration on the left has a tree process structure. However, after reduction, we obtain the configuration on the right which is clearly a forest and thus needs TC-Mix to be typeable. We have described a *synchronous* version of EGV, but extending our treatment to asynchrony as in the work of [FLMD19] is a routine adaptation.

7. Can we separate fork?

Hyper-environments allow us to cleanly separate name restriction and parallel composition in process configurations. A natural follow-on question is whether we could use the same technique at the level of *terms* in order to split **fork** into separate constructs for creating a channel and spawning a process. As tantalising a prospect this is, we argue that the disadvantages outweigh the benefits.

Suppose we were to extend term typing to allow hyper-environments, $\mathcal{G} \vdash M : T$, and were to introduce terms $\mathbf{let}\ \langle x, x' \rangle = \mathbf{new}\ \mathbf{in}\ M$ to create a channel and $\mathbf{let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N$ to spawn a thread, with the following typing rules:

$$\textsc{TM-LetNew}\quad \frac{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \vdash M : T}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \vdash \mathbf{let}\ \langle x, x' \rangle = \mathbf{new}\ \mathbf{in}\ M : T} \qquad \textsc{TM-LetSpawn}\quad \frac{\mathcal{G} \vdash M : \mathbf{end}_! \qquad \mathcal{H} \vdash N : T}{\mathcal{G} \parallel \mathcal{H} \vdash \mathbf{let}\ \langle\rangle = \mathbf{spawn}\ M\ \mathbf{in}\ N : T}$$

These rather ad-hoc rules mirror hypersequent cut and hypersequent composition: TM-LetNew creates a new channel with endpoints x and x', and requires them to be used in separate threads in the continuation M; and TM-LetSpawn takes a term M, spawns it as a child thread, and continues as N. Using these rules, we can encode $\mathbf{fork}\ M$ as $\mathbf{let}\ \langle x, x' \rangle = \mathbf{new}\ \mathbf{in}\ \mathbf{let}\ \langle\rangle = \mathbf{spawn}\ (M\ x)\ \mathbf{in}\ x'$.

Where else can we allow hyper-environments? In HCP, we have two options: (1) if we restrict *all logical rules* to singleton hypersequents and allow hyper-environments only in the rules for name restriction and parallel composition, we can use standard sequential semantics [MP18, KMP19b]; but (2) if we allow hyper-environments in *any logical rules*, we must use a semantics which allows the corresponding actions to be delayed [KMP19a]. This is unlikely to be a property of logical rules, but rather due to the fact that the logical rules correspond exactly to the communication actions—which block reduction—and the structural rules to name restriction and parallel composition—which do not. Therefore, we expect the positions where hypersequents can safely occur to follow from the structure of evaluation contexts and whether any blocking term perform a communication action.

Regardless of our choice, we would be left with restrictions on the syntax of terms that seem sensible in a process calculus, but are surprising in a λ-calculus. In the strictest variant, where we disallow hyper-environments in all but the above two rules, uses of TM-LetNew and TM-LetSpawn may be interleaved, but no other construct may appear between a

TM-LetNew and its corresponding TM-LetSpawn. Consider the following terms, where M uses x and y, and N uses x'. Term (7.1) may be well-typed, but (7.2) is always ill-typed:

$$\textbf{let } y = 1 \textbf{ in let } \langle x, x' \rangle = \textbf{new in let } \langle\rangle = \textbf{spawn } M \textbf{ in } N \tag{7.1}$$

$$\textbf{let } \langle x, x' \rangle = \textbf{new in let } y = 1 \textbf{ in let } \langle\rangle = \textbf{spawn } M \textbf{ in } N \tag{7.2}$$

Note that $\textbf{let } \langle x, x' \rangle = \textbf{new in } M$ is a single, monolithic term constructor—exactly what hypersequents were meant to prevent! However, if we attempt to decompose these constructors, we find that these are not the regular product and unit types.

8. Related work

Session Types and Functional Languages. Session types were originally introduced in the context of process calculi [Hon93, THK94, HVK98], however they have been vastly integrated also in functional calculi, a line of work initiated by Gay and collaborators [VRG04, VGR06, GV10]. This family of calculi builds session types directly into a lambda calculus. Toninho *et al.* [TCP13] take an alternative approach, stratifying their system into a session-typed process calculus and a separate functional calculus. There are many pragmatic embeddings of session type systems in existing functional programming languages [NT04, PT08, SE08, IYA10, OY16, KD21a]. A detailed survey is given by Orchard and Yoshida [OY17].

Propositions as Sessions. When Girard introduced linear logic [Gir87] he suggested a connection with concurrency. Abramsky [Abr94] and Bellin and Scott [BS94] give embeddings of linear logic proofs in π-calculus, where cut reduction is simulated by π-calculus reduction. Both embeddings interpret tensor as parallel composition. The correspondence with π-calculus is not tight in that these systems allow independent prefixes to be reordered. Caires and Pfenning [CP10] give a propositions as types correspondence between dual intuitionistic linear logic and a session-typed π-calculus called πDILL. They interpret tensor as output. The correspondence with π-calculus is tight in that independent prefixes may not be reordered. With CP [Wad14], Wadler adapts πDILL to classical linear logic. Aschieri and Genco [AG20] give an interpretation of classical multiplicative linear logic as concurrent functional programs. They interpret $\parr$ as parallel composition, and the connection to session types is less direct.

Priority-based Calculi. Systems such as πDILL, CP, and GV (and indeed HCP and HGV) ensure deadlock freedom by exploiting the type system to statically impose a tree structure on the communication topology — there can be at most one communication channel between any two processes. Another line of work explores a more liberal approach to deadlock freedom enabling some cyclic communication topologies, where deadlock freedom is guaranteed via *priorities*, which impose an order on actions. Priorites were introduced by Kobayashi and Padovani [Kob06, Pad14] and adopted by Dardha and Gay [DG18] in Priority CP (PCP), and Kokke and Dardha in Priority GV (PGV) [KD21b]. Dezani *et al.* [DCdY07] and Vieira and Vasconcelos [VV13] use a partial order on channels to guarantee deadlock freedom, following Kobayashi's work [Kob06]. Later on Dezani *et al.* [DCMYD06] guarantee progress by allowing only one active session at a time. Carbone *et al.* [CDM14] use catalysers to show that progress is a compositional form of lock freedom for standard typed π calculus. The authors describe how this technique can be used for session typed

π-calculus by using the the encoding of session types to linear types [DGS17, Dar14, Dar16]. Dardha and Perez [DP22] compare the different calculi and techniques for deadlock freedom using CP and CLL as a yardstick and showing that the class of processes in CP is strictly included in the class of processes typed by Kobayashi [Kob06].

Graph-theoretic Approaches. Carbone and Debois [CD10] define a graph-theoretic approach for a session typed π-calculus. They define an explicit dependency graph defined inductively on the structure of a process, in contrast to our approach of inducing a graph on type environments given a co-name set. They ensure progress for processes with acyclic graphs using a *catalyser*, which provides a missing counterpart to a process. Jacobs *et al.* [JBK22a] also define a graph-theoretic approach to deadlock freedom, but differently from Carbone and Debois, their work is based on separation logic. A line of work on many-writer, single-reader process calculi [Pad18, dP18] uses explicit dependency graphs to both ensure resource separation and guarantee deadlock freedom, however it is not immediate how to apply this approach to functional calculi.

9. Conclusion and future work

HGV exploits hypersequents to resolve fundamental modularity issues with GV. As a consequence, we have obtained a tight operational correspondence between HGV and HCP. HGV is a modular and extensible core calculus for functional programming with *binary* session types. In future we intend to apply hypersequents to *multiparty* versions of CP [CLM$^+$16] and GV [JBK22b] to exhibit a similarly strong operational correspondence.

Acknowledgements

We are deeply grateful to Marco Peressotti for discovering an error in our presentation of the LTS for HCP in the original conference paper, and for suggesting a fix via the definition of saturated transitions. We thank the CONCUR and LMCS reviewers for their helpful comments and suggestions. This work was supported by EPSRC grants EP/K034413/1 (ABCD), EP/T014628/1 (STARDUST), EP/L01503X/1 (CDT PPar), ERC Consolidator Grant 682315 (Skye), UKRI Future Leaders Fellowship MR/T043830/1 (EHOP), a UK Government ISCF Metrology Fellowship grant, EU HORIZON 2020 MSCA RISE project 778233 (BehAPI), and NSF grant CCF-2044815.

References

[Abr94] Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994. doi:10.1016/0304-3975(94)00103-0.

[AG20] Federico Aschieri and Francesco A. Genco. Par means parallel: multiplicative linear logic proofs as concurrent functional programs. *Proc. ACM Program. Lang.*, 4(POPL):18:1–18:28, 2020. doi:10.1145/3371086.

[ALM16] Robert Atkey, Sam Lindley, and J. Garrett Morris. Conflation confers concurrency. In *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer, 2016. doi:10.1007/978-3-319-30936-1_2.

[Avr91] Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991. doi:10.1007/BF01531058.

[BK01] Nick Benton and Andrew Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001. doi:10.1017/S0956796801004099.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 3 1998. doi:10.1016/s0304-3975(97)00220-x.

[BS94] Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994. doi:10.1016/0304-3975(94)00104-9.

[CD10] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

[CDM14] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Proc. of COORDINATION*, volume 8459 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014. doi:10.1007/978-3-662-43376-8_4.

[CLM+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *Proc. of CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

[CMS18] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Comput.*, 31(1):51–67, 2018. doi:10.1007/s00446-017-0295-1.

[CP10] Luìs Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

[Dar14] Ornela Dardha. Recursive session types revisited. In *Proc. of BEAT*, volume 162 of *Electron. Proc. in Theor. Comput. Sci.*, pages 27–34, 2014. doi:10.4204/EPTCS.162.4.

[Dar16] Ornela Dardha. *Type Systems for Distributed Programs: Components and Sessions*, volume 7 of *Atlantis Studies in Computing*. Springer / Atlantis Press, 2016. doi:10.2991/978-94-6239-204-5.

[DCdY07] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *Proc. of TGC*, volume 4912 of *Lect. Notes Comput. Sci.*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proc. of ECOOP*, volume 4067 of *Lect. Notes Comput. Sci.*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

[DG18] Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *Proc. of FoSSaCS*, volume 10803 of *LNCS*, pages 91–109. Springer, 2018. doi:10.1007/978-3-319-89366-2_5.

[DGS17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

[DMN07] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *J. Funct. Program.*, 17(6):793–812, 2007. doi:10.1017/S0956796807006387.

[dP18] Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *Proc. of ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.15.

[DP22] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022. doi:10.1016/j.jlamp.2021.100717.

[FKD+21] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *Proc. of CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.36.

[FLMD19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

[Fow19] Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, University of Edinburgh, 2019.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[ITT+19] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi:10.1017/S0956796819000169.

[IYA10] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. Session type inference in Haskell. In *Proc. of PLACES*, volume 69 of *EPTCS*, pages 74–91, 2010. doi:10.4204/EPTCS.69.6.

[JBK22a] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022.

[JBK22b] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022.

[KD21a] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proc. of Haskell*, pages 1–13. ACM, 2021. doi:10.1145/3471874.3472979.

[KD21b] Wen Kokke and Ornela Dardha. Prioritise the best variation. In *Proc. of FORTE*, volume 12719 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2021. doi:10.1007/978-3-030-78089-0_6.

[KMP19a] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract semantics for classical processes. *PACMPL*, 3(POPL), 2019. doi:10.1145/3290337.

[KMP19b] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Linearity-TLLA*, volume 292 of *EPTCS*, pages 90–103. Open Publishing Association, 2019. doi:10.4204/EPTCS.292.5.

[Kob03] Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*, pages 439–453. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-40007-3_26.

[Kob06] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

[LM99] Jean-Jacques Lévy and Luc Maranget. Explicit substitutions and programming languages. In *Foundations of Software Technology and Theoretical Computer Science, 1999*, volume 1738 of *LNCS*. Springer, 1999. doi:10.1007/3-540-46691-6_14.

[LM15] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 560–584. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-46669-8_23.

[LM16] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. *SIGPLAN Not.*, 51(9):434–447, 2016. doi:10.1145/3022670.2951921.

[LM17] Sam Lindley and J. Garrett Morris. Lightweight functional session types. In Simon Gay and Antonio Ravara, editors, *Behavioural Types: from Theory to Tools*, chapter 12, pages 265–286. River publishers, 2017.

[LPT03] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[Mon13] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013.

[MP18] Fabrizio Montesi and Marco Peressotti. Classical transitions. *CoRR*, abs/1803.01049, 2018. URL: https://arxiv.org/abs/1803.01049, arXiv:1803.01049.

[MP21] Fabrizio Montesi and Marco Peressotti. Linear logic, the π-calculus, and their metatheory: A recipe for proofs as processes. *CoRR*, abs/2106.11818, 2021. URL: https://arxiv.org/abs/2106.11818, arXiv:2106.11818.

[MV18] Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. *Log. Methods Comput. Sci.*, 14(4), 2018. doi:10.1007/978-3-662-43376-8_8.

[NT04] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Proc. of PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004. doi:10.1007/978-3-540-24836-1_5.

[OY16] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proc. of POPL*, pages 568–581. ACM, 2016. doi:10.1145/2837614.2837634.

[OY17] Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

[Pad14] Luca Padovani. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proc. of CSL-LICS*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

[Pad18] Luca Padovani. Deadlock-free typestate-oriented programming. *Art Sci. Eng. Program.*, 2(3):15, 2018. doi:10.22152/programming-journal.org/2018/2/15.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

[PT08] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. of Haskell*. ACM, 2008. doi:10.1145/1411286.1411290.

[Rey00] John C. Reynolds. The meaning of types—from intrinsic to extrinsic semantics. Technical Report RS-00-32, BRICS, 2000.

[San96] Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1-2):235–274, 1996. doi:10.1016/0304-3975(96)00075-8.

[SE08] Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. Unpublished manuscript, 2008.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proc. of PARLE*, volume 817 of *LNCS*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

[TV20] Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2020. doi:10.1145/3371135.

[Vas12] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.

[VGR06] Vasco Thudichum Vasconcelos, Simon J. Gay, and Antonio Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

[VRG04] Vasco Vasconcelos, Antonio Ravara, and Simon J. Gay. Session types for functional multithreading. In *CONCUR*, volume 3170 of *LNCS*, pages 497–511. Springer, 2004. doi:10.1007/978-3-540-28644-8_32.

[VV13] Hugo Torres Vieira and Vasco Thudichum Vasconcelos. Typing progress in communication-centred systems. In *Proc. of COORDINATION*, volume 7890 of *Lect. Notes Comput. Sci.*, pages 236–250. Springer, 2013. doi:10.1007/978-3-662-43376-8_10.

[Wad14] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014. doi:10.1145/2398856.2364568.

Appendices

Appendix A. Omitted Proofs for Section 3: Hypersequent GV

In this Appendix we give full definitions and proofs for Section 3.

$$\textsc{T-Seq}\ \frac{\Gamma \vdash M : \mathbf{1} \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash M; N : T} \qquad \textsc{T-LamUnit}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda().M : \mathbf{1} \multimap T} \qquad \textsc{T-LamPair}\ \frac{\Gamma, x : T, y : T' \vdash M : U}{\Gamma \vdash \lambda(x, y).M : T \times T' \multimap U}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash M : T \qquad \Delta, x : T \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : U} \qquad \textsc{T-Select-Inl}\ \frac{}{\cdot \vdash \mathbf{select\ inl} : S \oplus S' \multimap S}$$

$$\textsc{T-Select-Inr}\ \frac{}{\cdot \vdash \mathbf{select\ inr} : S \oplus S' \multimap S'} \qquad \textsc{T-Offer}\ \frac{\Gamma \vdash L : S \,\&\, S' \qquad \Delta, x : S \vdash M : T \qquad \Delta, y : S' \vdash N : T}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : T}$$

$$\textsc{T-Offer-Absurd}\ \frac{\Gamma \vdash L : \&\{\}}{\Gamma, \Delta \vdash \mathbf{offer}\ L\ \{\} : T}$$

Figure 9. Derived rules for syntactic sugar

A.1. **Derived typing rules for syntactic sugar.** The main body makes use of syntactic sugar, and encodings of branching and selection. Figure 9 shows the derived typing rules.

A.2. **Preservation Proof.** Next, we detail the proof of preservation. We begin with the usual lemmas to manipulate evaluation contexts, and the usual substitution lemma.

Lemma A.1 (Subterm typeability)**.** *Suppose* $\mathbf{D}$ *is a derivation of* $\Gamma \vdash E[M] : T$*. Then, there exist* Γ_1, Γ_2 *such that* $\Gamma = \Gamma_1, \Gamma_2$*, a type* U*, and some subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : U$*, where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *coincides with the position of the hole in* $\mathbf{D}$*.*

Proof. By induction on the structure of E. □

Lemma A.2 (Replacement, Evaluation Contexts)**.** *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : T$
- $\mathbf{D}'$ *is a subderivation of* $\boldsymbol{D}$ *concluding* $\Gamma_2 \vdash M : U$
- *The position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : U$
- Γ_1, Γ_3 *is defined*

then $\Gamma_1, \Gamma_3 \vdash E[N] : T$.

Proof. By induction on the structure of E. □

Lemma A.3 (Substitution)**.** *If:*

(1) $\Gamma_1, x : U \vdash M : T$
(2) $\Gamma_2 \vdash N : U$
(3) Γ_1, Γ_2 *is defined*

then $\Gamma_1, \Gamma_2 \vdash M\{N/x\} : T$.

Proof. By induction on the derivation of $\Gamma_1, x : U \vdash M : T$. □

Preservation of typing under term reduction is standard.

Lemma A.4 (Preservation, $\longrightarrow_{\mathsf{M}}$)**.** *If* $\Gamma \vdash M : T$ *and* $M \longrightarrow_{\mathsf{M}} N$*, then* $\Gamma \vdash N : T$.

Proof. A standard induction on the derivation of $\longrightarrow_{\mathsf{M}}$. □

Runtime type merging is commutative and associative. We make use of these properties implicitly in the remainder of the proofs.

Lemma A.5.

(1) $R_1 \sqcap R_2 \iff R_2 \sqcap R_1$
(2) $R_1 \sqcap (R_2 \sqcap R_3) \iff (R_1 \sqcap R_2) \sqcap R_3$

Proof. Immediate from the definition of $\sqcap$. □

The first more major result is preservation of configuration typing under structural congruence.

Lemma A.6 (Preservation ($\equiv$))**.** *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$.

Proof. We consider the cases for the equivalence axioms; the congruence cases are straightforward applications of the IH.

Case (SC-ParAssoc).

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$$

$$\dfrac{\mathcal{G}_1 \vdash \mathcal{C} : R_1 \qquad \dfrac{\mathcal{G}_2 \vdash \mathcal{D} : R_2 \qquad \mathcal{G}_3 \vdash \mathcal{E} : R_3}{\mathcal{G}_2 \parallel \mathcal{G}_3 \vdash \mathcal{D} \parallel \mathcal{E} : R_2 \sqcap R_3}}{\mathcal{G}_1 \parallel \mathcal{G}_2 \parallel \mathcal{G}_3 \vdash \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) : R_1 \sqcap R_2 \sqcap R_3} \iff \dfrac{\dfrac{\mathcal{G}_1 \vdash \mathcal{C} : R_1 \qquad \mathcal{G}_2 \vdash \mathcal{D} : R_2}{\mathcal{G}_1 \parallel \mathcal{G}_2 \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \qquad \mathcal{G}_3 \vdash \mathcal{E} : R_3}{\mathcal{G}_1 \parallel \mathcal{G}_2 \parallel \mathcal{G}_3 \vdash (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} : R_1 \sqcap R_2 \sqcap R_3}$$

Case (SC-ParComm).

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C}$$

$$\dfrac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2} \iff \dfrac{\mathcal{H} \vdash \mathcal{D} : U \qquad \mathcal{G} \vdash \mathcal{C} : T}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{D} \parallel \mathcal{C} : R_1 \sqcap R_2}$$

Case (SC-NewComm).

$$(\nu x x')(\nu y y')\mathcal{C} \equiv (\nu y y')(\nu x x')\mathcal{C}$$

Two illustrative subcases:

Subcase (1).

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \parallel \Gamma_3, y : S' \parallel \Gamma_4, y' : \overline{S'} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, x : S \parallel \Gamma_2, x' : \overline{S} \parallel \Gamma_3, \Gamma_4 \vdash (\nu y y')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \parallel \Gamma_3, \Gamma_4 \vdash (\nu x x')(\nu y y')\mathcal{C} : R}$$

$$\Longleftrightarrow$$

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x : S \parallel \Gamma_4, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, \Gamma_4 \vdash (\nu x x')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2 \parallel \Gamma_3, \Gamma_4 \vdash (\nu y y')(\nu x x')\mathcal{C} : R}$$

Subcase (2).

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, x : S \parallel \Gamma_3, x' : \overline{S} \vdash (\nu y y')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu x x')(\nu y y')\mathcal{C} : R}$$

$$\Longleftrightarrow$$

$$\frac{\dfrac{\mathcal{G} \parallel \Gamma_1, x : S, y : S' \parallel \Gamma_2, y' : \overline{S'} \parallel \Gamma_3, x' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G} \parallel \Gamma_1, \Gamma_3, y : S' \parallel \Gamma_2, y' : \overline{S'} \vdash (\nu x x')\mathcal{C} : R}}{\mathcal{G} \parallel \Gamma_1, \Gamma_2, \Gamma_3 \vdash (\nu y y')(\nu x x')\mathcal{C} : R}$$

Case (SC-NewSwap).

$$(\nu x y)\mathcal{C} \equiv (\nu y x)\mathcal{C}$$

Follows immediately since hyper-environments are treated as unordered.

Case (SC-ScopeExt).

$$\mathcal{C} \parallel (\nu x y)\mathcal{D} \equiv (\nu x y)(\mathcal{C} \parallel \mathcal{D})$$

(where $x, y \notin \mathrm{fv}(\mathcal{C})$)

$$\frac{\dfrac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R_2}{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash (\nu x y)\mathcal{D} : R_2}}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash \mathcal{C} \parallel (\nu x y)\mathcal{D} : R_1 \sqcap R_2}$$

$$\Longleftrightarrow$$

$$\frac{\dfrac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}}{\mathcal{G} \parallel \mathcal{H} \parallel \Gamma_1, \Gamma_2 \vdash (\nu x y)(\mathcal{C} \parallel \mathcal{D}) : R_1 \sqcap R_2}$$

Case (SC-LinkComm).

$$x \overset{z}{\leftrightarrow} y \equiv y \overset{z}{\leftrightarrow} x$$

Assumption:

$$\frac{}{x : S, y : \overline{S} \vdash x \overset{z}{\leftrightarrow} y : \circ}$$

By dualising both variables, we have that $x : \overline{S}, y : \overline{\overline{S}}$. Since duality is an involution, we can show $x : S, y : \overline{S} \iff x : \overline{S}, y : S$.

Thus:

$$\frac{}{y : S, x : \overline{S} \vdash y \overset{z}{\leftrightarrow} x : \circ}$$

The reasoning for the symmetric case is identical. □

The next result shows that configuration typeability is preserved under configuration reduction. Note that this lemma makes crucial use of Lemma A.6 due to E-Equiv.

Lemma A.7 (Preservation ($\longrightarrow$)). *If $\mathcal{G} \vdash \mathcal{C} : R$ and $\mathcal{C} \longrightarrow \mathcal{D}$, then $\mathcal{G} \vdash \mathcal{D} : R$.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. Where there is a choice for ϕ, we prove the case for $\phi = \bullet$ and expand $\mathcal{T}[M]$ to $\bullet(E[M])$ for some evaluation context E; the other cases are similar.

Case (E-Reify-Fork).

$$\bullet E[\mathbf{fork}\ V] \longrightarrow (\nu xy)(\bullet E[x] \parallel \circ V\ y)$$

Assumption:

$$\frac{\Gamma \vdash E[\mathbf{fork}\ V] : T}{\Gamma \vdash \bullet E[\mathbf{fork}\ V] : T}$$

By Lemma A.1, there exist Γ_1, Γ_2, S such that $\Gamma = \Gamma_1, \Gamma_2$ and $\Gamma_1, \Gamma_2 \vdash E[\mathbf{fork}\ V] : T$ and:

$$\frac{\Gamma_2 \vdash V : S \multimap \mathbf{end}_!}{\Gamma_2 \vdash \mathbf{fork}\ V : \overline{S}}$$

By Lemma A.2:

$$\frac{\Gamma_1, x : \overline{S} \vdash E[x] : T}{\Gamma_1, x : \overline{S} \vdash \bullet E[x] : T}$$

By TM-App, $\Gamma_2, y : S \vdash V\ y : \mathbf{end}_!$ and so by TC-Child, $\Gamma_2, y : S \vdash V\ y : \circ$
Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma_1, x : \overline{S} \vdash E[x] : T}{\Gamma_1, x : \overline{S} \vdash \bullet E[x] : T} \qquad \dfrac{\Gamma_2, y : S \vdash V\ y : \mathbf{end}_!}{\Gamma_2, y : S \vdash \circ(V\ y) : \circ}}{\Gamma_1, x : \overline{S} \parallel \Gamma_2, y : S \vdash \bullet E[x] \parallel \circ(V\ y) : T}}{\Gamma_1, \Gamma_2 \vdash (\nu xy)(\bullet E[x] \parallel \circ(V\ y) : T}$$

as required.

Case (E-Comm-Send).

$$(\nu xy)(\bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y]) \longrightarrow (\nu xy)(\bullet E[x] \parallel \circ E'[(V, y)])$$

Assumption:

$$\frac{\dfrac{\dfrac{\Gamma, x : S \vdash E[\mathbf{send}\ (V, x)] : U}{\Gamma, x : S \vdash \bullet E[\mathbf{send}\ (V, x)] : U} \qquad \dfrac{\Gamma', y : \overline{S} \vdash E'[\mathbf{recv}\ y] : \mathbf{end}_!}{\Gamma', y : \overline{S} \vdash \circ E'[\mathbf{recv}\ y] : \circ}}{\Gamma, x : S \parallel \Gamma', y : \overline{S} \vdash \bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y] : U}}{\Gamma, \Gamma' \vdash (\nu xy)(\bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y]) : U}$$

By Lemma A.1, there exist Γ_1, Γ_2, S such that $\Gamma = \Gamma_1, \Gamma_2$, and $\Gamma_1, \Gamma_2, x : S \vdash E[\mathbf{send}\ (V, x)] : U$ and:

$$\frac{\Gamma_2 \vdash V : T \qquad x : !T.S' \vdash x : !T.S'}{\Gamma_2, x : !T.S' \vdash \mathbf{send}\ (V, x) : S'}$$

With the knowledge that $S = !T.S'$, we can refine our original derivation:

$$\frac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, x : !T.S' \vdash E[\mathbf{send}\ (V, x)] : U}{\Gamma_1, \Gamma_2, x : !T.S' \vdash \bullet E[\mathbf{send}\ (V, x)] : U} \qquad \dfrac{\Gamma', y : ?T.\overline{S'} \vdash E'[\mathbf{recv}\ y] : \mathbf{end}_!}{\Gamma', y : ?T.\overline{S'} \vdash \circ E'[\mathbf{recv}\ y] : \circ}}{\Gamma_1, \Gamma_2, x : !T.S' \parallel \Gamma', y : ?T.\overline{S'} \vdash \bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y] : U}}{\Gamma_1, \Gamma_2, \Gamma' \vdash (\nu xy)(\bullet E[\mathbf{send}\ (V, x)] \parallel \circ E'[\mathbf{recv}\ y]) : U}$$

Again by Lemma A.1, we have that $\Gamma', y : ?T.\overline{S'} \vdash E'[\mathbf{recv}\ y] : \mathbf{end}_!$ and:

$$\frac{y : ?T.\overline{S'} \vdash y : ?T.\overline{S'}}{y : ?T.\overline{S'} \vdash \mathbf{recv}\ y : T \times \overline{S'}}$$

We can show:

$$\frac{\Gamma_2 \vdash V : T \qquad y : \overline{S'} \vdash y : \overline{S'}}{\Gamma_2, y : \overline{S'} \vdash (V, y) : T \times \overline{S'}}$$

By Lemma A.2, we have that $\Gamma_2, \Gamma', y : \overline{S'} \vdash E'[(V, y)] : \overline{S'}$.
Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma_1, x : S' \vdash E[x] : U}{\Gamma_1, x : S' \vdash \bullet E[x] : U} \qquad \dfrac{\Gamma_2, \Gamma', y : \overline{S'} \vdash E'[(V, y)] : \mathbf{end}_!}{\Gamma_2, \Gamma', y : \overline{S'} \vdash \circ E'[(V, y)] : \circ}}{\Gamma_1, x : S' \parallel \Gamma_2, \Gamma', y : \overline{S'} \vdash \bullet E[x] \parallel \circ E'[(V, y)] : U}}{\Gamma_1, \Gamma_2, \Gamma' \vdash (\nu xy)(\bullet E[x] \parallel \circ E'[(V, y)]) : U}$$

as required.

Case (E-Comm-Close).

$$(\nu xy)(\mathcal{T}[\mathbf{wait}\ x] \parallel \circ y) \longrightarrow \mathcal{T}[()]$$

Taking $\mathcal{T} = \bullet E$, assumption:

$$\frac{\dfrac{\dfrac{\Gamma, x : \mathbf{end}_? \vdash E[\mathbf{wait}\ x] : T}{\Gamma, x : \mathbf{end}_? \vdash \bullet E[\mathbf{wait}\ x] : T} \qquad \dfrac{\overline{y : \mathbf{end}_! \vdash y : \mathbf{end}_!}}{y : \mathbf{end}_! \vdash \circ y : \circ}}{\Gamma, x : \mathbf{end}_? \parallel y : \mathbf{end}_! \vdash \bullet E[\mathbf{wait}\ x] \parallel \circ y : T}}{\Gamma \vdash (\nu xy)(\bullet E[\mathbf{wait}\ x] \parallel \circ y) : T}$$

By Lemma A.1, we have that:

$$\frac{x : \mathbf{end}_? \vdash x : \mathbf{end}_?}{x : \mathbf{end}_? \vdash \mathbf{wait}\ x : \mathbf{1}}$$

By Lemma A.2, $\Gamma \vdash E[()] : T$.
Recomposing:

$$\frac{\Gamma \vdash E[()] : T}{\Gamma \vdash \bullet E[()] : T}$$

as required.

Case (E-Reify-Link).

$$F[\mathbf{link}\ (x, y)] \longrightarrow (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel F[z'])$$

where z, z' fresh.
Taking $F = \bullet E$, we have that:

$$\frac{\Gamma \vdash E[\mathbf{link}\ (x, y)] : T}{\Gamma \vdash \bullet E[\mathbf{link}\ (x, y)] : T}$$

By Lemma A.1, we have that $\Gamma = \Gamma', x : S, y : \overline{S}$ such that:

$$\frac{\dfrac{x : S \vdash x : S \qquad y : \overline{S} \vdash y : \overline{S}}{x : S, y : \overline{S} \vdash (x, y) : S \times \overline{S}}}{x : S, y : \overline{S} \vdash \mathbf{link}\ (x, y) : \circ}$$

By Lemma A.2, we have that $\Gamma', z : \mathbf{end}_! \vdash E[z] : T$.
Reconstructing:

$$\frac{\dfrac{\overline{z : \mathbf{end}_?, x : S, y : \overline{S} \vdash x \overset{z}{\leftrightarrow} y : \circ} \qquad \Gamma', z : \mathbf{end}_! \vdash \bullet E[z] : T}{z : \mathbf{end}_?, x : S, y : \overline{S} \parallel \Gamma', z : \mathbf{end}_! \vdash x \overset{z}{\leftrightarrow} y \parallel \bullet E[z] : T}}{\Gamma', x : S, y : \overline{S} \vdash (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \bullet E[z]) : T}$$

as required.

Case (E-Comm-Link).

$$(\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z \parallel \bullet M) \longrightarrow \bullet(M\{y/x\})$$

Assumption:

$$\frac{\dfrac{\overline{x : S, y : \overline{S}, z : \mathbf{end}_? \vdash x \overset{z}{\leftrightarrow} y : \circ} \qquad \dfrac{\dfrac{z' : \mathbf{end}_! \vdash z : \mathbf{end}_!}{z' : \mathbf{end}_! \vdash \circ z : \circ} \qquad \dfrac{\Gamma, x' : \overline{S} \vdash M : T}{\Gamma, x' : \overline{S} \vdash \bullet M : T}}{z' : \mathbf{end}_! \parallel \Gamma, x' : \overline{S} \vdash \circ z \parallel \bullet M : T}}{x : S, y : \overline{S}, z : \mathbf{end}_? \parallel z' : \mathbf{end}_! \parallel \Gamma, x' : \overline{S} \vdash x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M : T}}{\Gamma, y : \overline{S}, z : \mathbf{end}_? \parallel z' : \mathbf{end}_! \vdash (\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M) : T}}{\Gamma, y : \overline{S} \vdash (\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ z' \parallel \bullet M) : T}$$

By Lemma A.3, $\Gamma, y' : \overline{S} \vdash M\{y/x'\} : T$, thus:

$$\frac{\Gamma, y' : \overline{S} \vdash M\{y/x'\} : T}{\Gamma, y' : \overline{S} \vdash \bullet M\{y/x'\} : T}$$

as required.

Case (E-Res).

$$(\nu xy)\mathcal{C} \longrightarrow (\nu xy)\mathcal{D} \qquad \text{if } \mathcal{C} \longrightarrow \mathcal{D}$$

Immediate by the IH.

Case (E-Par).

$$\mathcal{C} \parallel \mathcal{D} \longrightarrow \mathcal{C}' \parallel \mathcal{D} \qquad \text{if } \mathcal{C} \longrightarrow \mathcal{C}'$$

Immediate by the IH.

Case (E-Equiv).

$$\mathcal{C} \longrightarrow \mathcal{D} \quad \text{if } \mathcal{C} \equiv \mathcal{C}', \mathcal{C}' \longrightarrow \mathcal{D}', \text{ and } \mathcal{D}' \equiv \mathcal{D}$$

Assumption: $\mathcal{G} \vdash \mathcal{C} : R$.
By Lemma A.6, $\mathcal{G} \vdash \mathcal{C}' : R$.
By the IH, $\mathcal{G} \vdash \mathcal{D}' : R$.
By Lemma A.6, $\mathcal{G} \vdash \mathcal{D} : R$, as required.

Case (E-Lift-M).

$$\phi M \longrightarrow \phi N \qquad \text{if } M \longrightarrow_{\mathsf{M}} N$$

Immediate by Lemma A.4. □

Theorem 3.3 (Preservation)**.**
(1) *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$.
(2) *If* $\mathcal{G} \vdash \mathcal{C} : R$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then* $\mathcal{G} \vdash \mathcal{D} : R$.

Proof. A direct corollary of Lemmas A.6 and A.7. □

A.3. **Progress.** Functional reduction satisfies progress: under an environment only containing runtime names, a term will either reduce, be a value, or be ready to perform a communication action.

Lemma A.8 (Progress, Terms)**.** *If* $\Psi \vdash M : T$*, then either* M *is a value, or there exists some* N *such that* $M \longrightarrow_{\mathsf{M}} N$*, or* M *can be written* $E[N]$ *for some* $N \in \{\mathbf{fork}\ V, \mathbf{send}\ (V, W), \mathbf{recv}\ V, \mathbf{wait}\ V, \mathbf{link}\ (V, W)\}$.

Proof. A standard induction on the derivation of $\Psi \vdash M : T$. □

Note that tree canonical forms can be defined inductively:

$$\mathcal{F} ::= \phi M \mid (\nu xy)(\mathcal{A} \parallel \mathcal{F})$$

We assume the same requirement for configurations $\mathcal{F}$ as the non-inductive definition of tree canonical forms: i.e., that for a configuration $(\nu xy)(\mathcal{A} \parallel \mathcal{F})$, that $x \in \mathrm{fv}(\mathcal{A})$.

Lemma 3.17 follows as a direct corollary of a slightly more verbose property, which follows from the inductive definition of TCFs.

Definition A.9 (Open progress)**.** Suppose $\Psi \vdash \mathcal{F} : R$, where $\mathcal{F} \not\longrightarrow$. We say that $\mathcal{F}$ *satisfies open progress* if:

(1) $\mathcal{C} = (\nu x x')(\mathcal{A} \parallel \mathcal{F}')$, where:
  (a) There exist Ψ_1, Ψ_2 such that $\Psi = \Psi_1, \Psi_2$
  (b) $\Psi_1, x{:}S \vdash \mathcal{A} : \circ$ for some session type S, and $\mathsf{blocked}(\mathcal{A}, y)$ for some $y \in \mathrm{fv}(\Psi_1, x{:}S)$
  (c) $\Psi_2, x'{:}\overline{S} \vdash \mathcal{D} : R$, where $\mathcal{F}'$ satisfies open progress
(2) $\mathcal{F} = \phi M$, and either M is a value, or $\mathsf{blocked}(\phi M, x)$ for some $x \in \mathrm{fv}(\Psi)$.

Lemma A.10 (Open progress)**.** *If $\Psi \vdash \mathcal{F} : R$ and $\mathcal{F} \not\longrightarrow$, then $\mathcal{F}$ satisfies open progress.*

Proof. By induction on the derivation of $\mathcal{G} \vdash \mathcal{F} : R$. By the definition of canonical forms, it must be the case that $\mathcal{C}$ is of the form $(\nu x y)(\mathcal{A} \parallel \mathcal{F}')$ or $\bullet M$.

We show the case where $\mathcal{C} = (\nu x y)(\circ M \parallel \mathcal{F}')$; the cases for $\mathcal{A} = x \overset{z}{\leftrightarrow} x'$ and $\mathcal{C} = \bullet M$ follow similar reasoning.

Assumption:

$$\dfrac{\dfrac{\Psi_1, x : S \vdash \mathcal{A} : \circ \qquad \Psi_2, y : \overline{S} \vdash \mathcal{F}' : R}{\Psi_1, x : S \parallel \Psi_2, y : \overline{S} \vdash \mathcal{A} \parallel \mathcal{F}' : R}}{\Psi_1, \Psi_2 \vdash (\nu x y)(\circ M \parallel \mathcal{F}') : R}$$

In both cases, by the induction hypothesis, $\Psi_2, y : \overline{S} \vdash \mathcal{F}' : T$ satisfies open progress.

Subcase ($\mathcal{A} = \circ M$). By Lemma A.8, either M is a value, or M can be written $E[N]$ for some communication and concurrency construct $N \in \{\mathbf{fork}\ V, \mathbf{send}\ (V, W), \mathbf{recv}\ V, \mathbf{wait}\ V, \mathbf{link}\ (V, W)\}$.

Otherwise, M is a communication or concurrency construct. If $N = \mathbf{fork}\ V$, then reduction could occur by E-Reify-Fork. If $N = \mathbf{link}\ (V, W)$, then by the type schema for **link**, we have that $\mathbf{link}\ (V, W)$ must be of the form $\mathbf{link}\ (z, z')$ for $z, z' \in \mathrm{fv}(\Psi, x : S)$ and could reduce by E-Reify-Link.

Otherwise, it must be the case that $\mathsf{blocked}(\circ M, z)$ for some $z \in \mathrm{fv}(\Psi_1, x : S)$.

Thus, $(\nu x y)(\circ M \parallel \mathcal{D})$ satisfies open progress, as required.

Subcase ($\mathcal{A} = z_2 \overset{z_1}{\leftrightarrow} z_3$). We have that $z_1, z_2, z_3 \in \mathrm{fv}(\Psi_1, x : S)$, and the thread must be blocked by definition. □

Appendix B. Omitted Proofs for section 4: Relation between HGV and GV

Theorem 4.3 (Typeability of GV configurations in HGV)**.** *If $\Gamma \vdash_{\mathsf{GV}} \mathcal{C} : R$, then there exists some $\mathcal{G}$ such that $\mathcal{G}$ is a splitting of Γ and $\mathcal{G} \vdash \mathcal{C} : R$.*

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C} : R$.

Case (TG-New). Assumption:

$$\dfrac{\Gamma, \langle y, y' \rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} : R}{\Gamma \vdash_{\mathsf{GV}} (\nu y y')\mathcal{C} : R}$$

Suppose $\Gamma = \langle x_1, x_1' \rangle : S_1^\sharp, \ldots, \langle x_n, x_n' \rangle : S_n^\sharp$ (for clarity, without loss of generality, we assume the absence of non-session variables. As these are simply split between environments, they can be added orthogonally).

By the IH, we have that there exists some hyper-environment $\mathcal{G}$ such that $\mathcal{G} \vdash \mathcal{C} : R$, where $\mathcal{G}$ is a splitting of $\Gamma, \langle y, y' \rangle : S^\sharp$.

Since $\mathcal{G}$ is a splitting of $\mathcal{C}$, we know that $y : S \in \mathcal{G}$ and $y' : \overline{S} \in \mathcal{G}$, and that $\mathcal{G}$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}, \{y, y'\}\}$.

Since $\mathcal{G}$ has a tree structure, by definition we have that $\mathcal{G} = \mathcal{G}' \parallel \Gamma_1, y : S \parallel \Gamma_2, y' : \overline{S}$ for some $\mathcal{G}', \Gamma_1, \Gamma_2$, where $\mathcal{G}'$ has a tree structure.

By Lemma 3.12 (clause 1, left-to-right), $\mathcal{G}' \parallel \Gamma_1, \Gamma_2$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\}$.

Thus, we can show:

$$\frac{\mathcal{G}' \parallel \Gamma_1, y : S \parallel \Gamma_2, y' : \overline{S} \vdash \mathcal{C} : R}{\mathcal{G}' \parallel \Gamma_1, \Gamma_2 \vdash (\nu y y')\mathcal{C} : R}$$

where $\mathcal{G}' \parallel \Gamma_1, \Gamma_2$ has a tree structure with respect to names $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}\}$ and is therefore a splitting of Γ, as required.

Case (TG-Connect$_1$). Assumption:

$$\frac{\Gamma_1, y : S \vdash_{\mathsf{GV}} \mathcal{C} : R_1 \qquad \Gamma_2, y' : \overline{S} \vdash_{\mathsf{GV}} \mathcal{D} : R_2}{\Gamma_1, \Gamma_2, \langle y, y' \rangle : S^\sharp \vdash_{\mathsf{GV}} \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}$$

Suppose $\Gamma_1 = \langle x_1, x_1' \rangle : S_1^\sharp, \ldots, \langle x_m, x_m' \rangle : S_m^\sharp$ and $\Gamma_2 = \langle x_{m+1}, x_{m+1}' \rangle : S_{m+1}^\sharp, \ldots, \langle x_n, x_n' \rangle : S_n^\sharp$.

By the IH, there exist hyper-environments $\mathcal{G}, \mathcal{H}$ such that:

(1) $\mathcal{G}$ is a splitting of $\Gamma_1, y : S$
(2) $\mathcal{H}$ is a splitting of $\Gamma_2, y' : \overline{S}$
(3) $\mathcal{G} \vdash_{\mathsf{GV}} \mathcal{C} : R_1$
(4) $\mathcal{H} \vdash_{\mathsf{GV}} \mathcal{D} : R_2$

By the definition of splittings, $\mathcal{G}$ and $\mathcal{H}$ can be written $\mathcal{G} = \mathcal{G}' \parallel \Gamma_1', y : S$ and $\mathcal{H} = \mathcal{H}' \parallel \Gamma_2', y' : \overline{S}$ for some Γ_1', Γ_2'. Furthermore, $\mathcal{G}$ has a tree structure with respect to $\{\{x_1, x_1'\}, \ldots, \{x_m, x_m'\}\}$ and $\mathcal{H}$ has a tree structure with respect to $\{\{x_{m+1}, x_{m+1}'\}, \ldots, \{x_n, x_n'\}\}$.

By Lemma 3.12 (clause 2, left-to-right), $\mathcal{G}' \parallel \Gamma_1', y : S \parallel \mathcal{H}' \parallel \Gamma_2', y' : \overline{S}$ has a tree structure with respect to $\{\{x_1, x_1'\}, \ldots, \{x_n, x_n'\}, \{y, y'\}\}$ and therefore $\mathcal{G} \parallel \mathcal{H}$ is a splitting of $\Gamma_1, \Gamma_2, \langle y, y' \rangle : S^\sharp$.

Recomposing in HGV:

$$\frac{\mathcal{G} \vdash \mathcal{C} : R_1 \qquad \mathcal{H} \vdash \mathcal{D} : R_2}{\mathcal{G} \parallel \mathcal{H} \vdash \mathcal{C} \parallel \mathcal{D} : R_1 \sqcap R_2}$$

as required.

Case (TG-Connect$_2$). Similar to TG-Connect$_1$.

Case (TG-Child). Assumption:

$$\frac{\Gamma \vdash M : \mathbf{end}_!}{\Gamma \vdash_{\mathsf{GV}} \circ M : \circ}$$

Since we mandated that variables of type $S^\sharp$ cannot appear in terms, there are no names of type $S^\sharp$ in Γ. Therefore, the singleton hyper-environment Γ is a valid splitting, and so we can conclude by TC-Child in HGV.

Case (TG-Main). Similar to TG-Child. □

Proposition 4.5. *Suppose $\Gamma \vdash \mathcal{C} : R$ where $\mathcal{C}$ is in tree canonical form. Then, $\Gamma \vdash_{GV} \mathcal{C} : R$.*

Proof. By induction on the number of ν-bound names.

In the case that $n = 0$, the result follows immediately by TG-Child or TG-Main.

In the case that $n \geq 1$, we have that $\Gamma = \Gamma_1, \Gamma_2$ for some Γ_1, Γ_2 and:

$$\frac{\dfrac{\Gamma_1, x : S \vdash \circ L : \circ \qquad \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R}{\Gamma_1, x : S \parallel \Gamma_2, y : \overline{S} \vdash \circ L \parallel \mathcal{D} : R}}{\Gamma_1, \Gamma_2 \vdash (\nu xy)(\circ L \parallel \mathcal{D}) : R}$$

such that $\mathcal{D}$ is in tree canonical form. That $\Gamma_1, x : S \vdash \circ L : \circ$ follows by the definition of tree canonical forms, since $x \in \mathrm{fv}(L)$.

By the IH, $\Gamma_2, y : \overline{S} \vdash \mathcal{D} : R$ in GV.

Thus, we can write:

$$\frac{\dfrac{\Gamma_1, x : S \vdash \circ L : \circ \qquad \Gamma_2, y : \overline{S} \vdash \mathcal{D} : R}{\Gamma_1, \Gamma_2, \langle x, y \rangle : S^\sharp \vdash \circ L \parallel \mathcal{D} : R}}{\Gamma_1, \Gamma_2 \vdash (\nu xy)(\circ L \parallel \mathcal{D}) : R}$$

as required. □

Appendix C. Omitted Proofs for section 5: Relation between HGV and CP

C.1. Full definition of HGV$*$.

Syntax.

Terms	L, M, N	::=	$V \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \mid V\ W$
		$\mid$	$\mathbf{let}\ () = V\ \mathbf{in}\ M \mid \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M$
		$\mid$	$\mathbf{absurd}\ V \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}$
Values	V, W	::=	$x \mid K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V$
Evaluation contexts	E	::=	$\Box \mid \mathbf{let}\ x = E\ \mathbf{in}\ M$
Thread contexts	F	::=	$\phi\ E$

Typing rules for values $\boxed{\Gamma \vdash V : T}$

$$\text{TV*-Var}\ \frac{}{x : T \vdash x : T} \qquad \text{TV*-Const}\ \frac{}{\cdot \vdash K : T} \qquad \text{TV*-Lam}\ \frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x.M : T \multimap U} \qquad \text{TV*-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}}$$

$$\text{TV*-Pair}\ \frac{\Gamma \vdash V : T \qquad \Delta \vdash W : U}{\Gamma, \Delta \vdash (V, W) : T \times U} \qquad \text{TV*-Absurd}\ \frac{\Gamma \vdash V : \mathbf{0}}{\Gamma \vdash \mathbf{absurd}\ V : T} \qquad \text{TV*-Inl}\ \frac{\Gamma \vdash V : T}{\Gamma \vdash \mathbf{inl}\ V : T + U} \qquad \text{TV*-Inr}\ \frac{\Gamma \vdash V : U}{\Gamma \vdash \mathbf{inr}\ V : T + U}$$

Typing rules for terms $\boxed{\Gamma \vdash M : T}$

$$\text{TM*-App}\ \frac{\Gamma \vdash V : T \multimap U \qquad \Delta \vdash W : T}{\Gamma, \Delta \vdash V\ W : U} \qquad \text{TM*-Let}\ \frac{\Gamma \vdash M : T \qquad \Delta, x : T \vdash N : U}{\Gamma, \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : U}$$

$$\text{TM*-LetUnit}\ \frac{\Gamma \vdash V : \mathbf{1} \qquad \Delta \vdash M : T}{\Gamma, \Delta \vdash \mathbf{let}\ () = V\ \mathbf{in}\ M : T} \qquad \text{TM*-LetPair}\ \frac{\Gamma \vdash V : T \times T' \qquad \Delta, x : T, y : T' \vdash M : U}{\Gamma, \Delta \vdash \mathbf{let}\ (x, y) = V\ \mathbf{in}\ M : U}$$

$$\text{TM*-CaseSum}\ \frac{\Gamma \vdash V : T + T' \qquad \Delta, x : T \vdash M : U \qquad \Delta, y : T' \vdash N : U}{\Gamma, \Delta \vdash \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

The typing of constants is the same as for HGV.

Operational Semantics. The operational semantics for HGV$*$ is the same as for HGV (Figure 4), with the addition of the following explicit rule for **let**:

$$\text{E-Let} \quad \mathbf{let}\ x = V\ \mathbf{in}\ M \quad \longrightarrow \quad M\{V/x\}$$

Similarly, HGV$*$ directly inherits HGV's runtime typing.

C.2. **Translating HGV$*$ to HCP.** The translation is guaranteed to have only internal (i.e., α or β) transitions and transitions on the dedicated output channel. More specifically:

Lemma C.1.

- *If* $[\![C]\!]^{\mathsf{c}}_r \xrightarrow{\ell}$, *then* $\ell \in \{\alpha, \beta\}$ *or* $\ell = \ell_r$.
- *If* $[\![M]\!]^{\mathsf{m}}_r \xrightarrow{\ell}$ *and* M *is a non-value, then* $\ell \in \{\alpha, \beta\}$.
- *If* V *is a value, then* $[\![V]\!]^{\mathsf{m}}_r \xrightarrow{\ell_r}$.
- *If* $[\![V]\!]^{\mathsf{v}}_r \xrightarrow{\ell}$ *then* $\ell \in \{\alpha, \beta\}$.

Proof. By induction on the structure of M. □

We do not use the above lemma directly, but it is a useful sanity check.

Definition C.2 (process contexts)**.** A process context $P[\,]$ is a process with a single hole, denoted $\Box$. We extend the typing rules, LTS and typing rules to process contexts. We write $P[\,] \vdash \mathcal{G}/\mathcal{H}$ to mean that $P[\,]$ is typed under hyper-environment $\mathcal{H}$ expecting a process typed under $\mathcal{G}$, *i.e.*, if $Q \vdash \mathcal{G}$ then $P[Q] \vdash \mathcal{H}$.

Definition C.3. A process P is *blocked on* x if it only has transitions $P \xrightarrow{\ell_x}$.

Lemma C.4. *If* $P[\,]$ *is a process context with* $z, w, w' \notin \mathrm{cn}(P[\,])$, *and* Q *is a process blocked on* w', *then* $(\nu w w')(P[z{\leftrightarrow}w] \parallel Q) \approx_\alpha P[Q\{z/w'\}]$.

Proof. By induction on the process context $P[\,]$.

Case ($\Box$).

$$\begin{array}{rl} & (\nu w w')(z{\leftrightarrow}w \parallel Q) \\ \xrightarrow{\alpha} & Q\{z/w'\} \\ \sim & Q\{z/w'\} \qquad \text{(by reflexivity)} \end{array}$$

Case ($(\nu x y)P[\,]$).

$$\begin{array}{rll} & (\nu w w')((\nu x y)(P[z{\leftrightarrow}w]) \parallel Q) & \\ \sim & (\nu x y)(\nu w w')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.6)} \\ \approx_\alpha & (\nu x y)(P[Q\{z/w'\}]) & \text{(by Lemma 5.6 and IH)} \end{array}$$

Case ($P[\,] \parallel R$).

$$\begin{array}{rll} & (\nu w w')(P[z{\leftrightarrow}w] \parallel R \parallel Q) & \\ \sim & (\nu w w')(P[z{\leftrightarrow}w] \parallel Q) \parallel R & \text{(by Lemma 5.6)} \\ \approx_\alpha & P[Q\{z/w'\}] \parallel R & \text{(by Lemma 5.6 and IH)} \end{array}$$

Case ($R \parallel P[\,]$).

$$\begin{array}{rll} & (\nu w w')(R \parallel P[z{\leftrightarrow}w] \parallel Q) & \\ \sim & R \parallel (\nu w w')(P[z{\leftrightarrow}w] \parallel Q) & \text{(by Lemma 5.6)} \\ \approx_\alpha & R \parallel P[Q\{z/w'\}] & \text{(by Lemma 5.6 and IH)} \end{array}$$

Case ($\pi.P[\,]$). Since Q is blocked on w', the process $(\nu w w')(\pi.P[z{\leftrightarrow}w] \parallel Q)$ has only one transition,

$$(\nu w w')(\pi.P[z{\leftrightarrow}w] \parallel Q) \xrightarrow{\pi} (\nu w w')(P[z{\leftrightarrow}w] \parallel Q).$$

The process $\pi.P[Q\{z/w'\}]$ has only one transition, also with label π,

$$\pi.P[Q\{z/w'\}] \xrightarrow{\pi} P[Q\{z/w'\}].$$

The resulting processes are bisimilar by the induction hypothesis.

Case ($x \triangleright \{\mathrm{inl} : P[\,]; \mathrm{inr} : P'[\,]\}$). Since Q is blocked on w', the process $(\nu w w')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q)$ has only two transitions,

$$(\nu w w')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q) \xrightarrow{x \triangleright \mathrm{inl}} (\nu w w')(P[z{\leftrightarrow}w] \parallel Q)$$

and

$$(\nu w w')(x \triangleright \{\mathrm{inl} : P[z{\leftrightarrow}w]; \mathrm{inr} : P'[z{\leftrightarrow}w]\} \parallel Q) \xrightarrow{x \triangleright \mathrm{inr}} (\nu w w')(P'[z{\leftrightarrow}w] \parallel Q).$$

The process $x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\}$ has only two transitions, also with labels $x \triangleright \mathrm{inl}$ and $x \triangleright \mathrm{inr}$,

$$x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\} \xrightarrow{x \triangleright \mathrm{inl}} P[Q\{z/w'\}]$$

and

$$x \triangleright \{\mathrm{inl} : P[Q\{z/w'\}]; \mathrm{inr} : P'[Q\{z/w'\}]\} \xrightarrow{x \triangleright \mathrm{inr}} P'[Q\{z/w'\}].$$

The resulting processes are bisimilar by the induction hypothesis. $\square$

Lemma 5.10 (Substitution)**.** *If M is a well-typed term with $w \in \mathrm{fv}(M)$, and V is a well-typed value, then $(\nu w w')(\llbracket M \rrbracket^{\mathsf{m}}_r \parallel \llbracket V \rrbracket^{\mathsf{v}}_{w'}) \approx_\alpha \llbracket M\{V/w\} \rrbracket^{\mathsf{m}}_r$.*

Proof. Immediately from Lemma C.4. $\square$

Lemma 5.9 (Type Preservation)**.**

(1) *If* $\Gamma \vdash V : T$*, then* $[\![V]\!]^{\mathsf{v}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}$.
(2) *If* $\Gamma \vdash M : T$*, then* $[\![M]\!]^{\mathsf{m}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \mathbf{1} \otimes \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}$.
(3) *If* $\mathcal{G} \parallel \Gamma \vdash \mathcal{C} : T$*, where* Γ *is the type environment for the main thread in* $\mathcal{C}$*, then* $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \vdash \lfloor\!\lfloor \mathcal{G} \rfloor\!\rfloor \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}$.

Proof. Part 1.

- **Case** (x). We have $\dfrac{}{x : T \vdash x : T}$ and $[\![x]\!]^{\mathsf{v}}_r = r{\leftrightarrow}x$. We can derive:

$$\frac{}{x{\leftrightarrow}r \vdash x : \lfloor\!\lfloor A \rfloor\!\rfloor, r : \lfloor\!\lfloor A \rfloor\!\rfloor^{\perp}}$$

- **Case** (K). We have one case for each communication primitive.
  - Subcase **link**. We have $\mathbf{link} : S \times \overline{S} \multimap \mathbf{end}_!$, where

$$\begin{aligned}\lfloor\!\lfloor S \times S^{\perp} \multimap \mathbf{end}_! \rfloor\!\rfloor^{\perp} &= \lceil\!\lceil S \times S^{\perp} \multimap \mathbf{end}_! \rceil\!\rceil \\ &= \lceil\!\lceil S \times S^{\perp} \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil \mathbf{end}_! \rceil\!\rceil) \\ &= (\lceil\!\lceil S \rceil\!\rceil^{\perp} \parr \lceil\!\lceil S \rceil\!\rceil) \parr (\mathbf{1} \otimes \perp)\end{aligned}$$

    and $[\![\mathbf{link}]\!]^{\mathsf{v}}_r = r(y).y(x).\bar{r}.r().x{\leftrightarrow}y$. We can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{\leftrightarrow}y \vdash x : \lceil\!\lceil S \rceil\!\rceil^{\perp}, y : \lceil\!\lceil S \rceil\!\rceil}}{r().x{\leftrightarrow}y \vdash x : \lceil\!\lceil S \rceil\!\rceil^{\perp}, y : \lceil\!\lceil S \rceil\!\rceil, r : \perp}}{\bar{r}.r().x{\leftrightarrow}y \vdash x : \lceil\!\lceil S \rceil\!\rceil^{\perp}, y : \lceil\!\lceil S \rceil\!\rceil, r : \mathbf{1} \otimes \perp}}{y(x).\bar{r}.r().x{\leftrightarrow}y \vdash y : \lceil\!\lceil S \rceil\!\rceil^{\perp} \parr \lceil\!\lceil S \rceil\!\rceil, r : \mathbf{1} \otimes \perp}}{r(y).y(x).\bar{r}.r().x{\leftrightarrow}y \vdash r : (\lceil\!\lceil S \rceil\!\rceil^{\perp} \parr \lceil\!\lceil S \rceil\!\rceil) \parr (\mathbf{1} \otimes \perp)}$$

  - Subcase **fork**: We have $\mathbf{fork} : (S \multimap \mathbf{end}_!) \multimap \overline{S}$ where

$$\begin{aligned}\lfloor\!\lfloor (S \multimap \mathbf{end}_!) \multimap S^{\perp} \rfloor\!\rfloor^{\perp} &= \lceil\!\lceil (S \multimap \mathbf{end}_!) \multimap S^{\perp} \rceil\!\rceil \\ &= \lceil\!\lceil S \multimap \mathbf{end}_! \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil S^{\perp} \rceil\!\rceil) \\ &= (\lceil\!\lceil S \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil \mathbf{end}_! \rceil\!\rceil))^{\perp} \parr (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor) \\ &= (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor)\end{aligned}$$

    and $[\![\mathbf{fork}]\!]^{\mathsf{v}}_r = (\nu y y')(r(x).y\langle x\rangle.\bar{r}.r{\leftrightarrow}y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0})$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r{\leftrightarrow}y \vdash y : \lfloor\!\lfloor S \rfloor\!\rfloor^{\perp}, r : \lfloor\!\lfloor S \rfloor\!\rfloor}}{\bar{r}.r{\leftrightarrow}y \vdash y : \lfloor\!\lfloor S \rfloor\!\rfloor^{\perp}, r : \mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor}}{y\langle x\rangle.\bar{r}.r{\leftrightarrow}y \vdash y : (\lfloor\!\lfloor S \rfloor\!\rfloor \parr (\mathbf{1} \otimes \perp)) \otimes \lfloor\!\lfloor S \rfloor\!\rfloor^{\perp}, x : (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})), r : (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor)}}{(r(x).y\langle x\rangle.\bar{r}.r{\leftrightarrow}y) \vdash y : (\lfloor\!\lfloor S \rfloor\!\rfloor \parr (\mathbf{1} \otimes \perp)) \otimes \lfloor\!\lfloor S \rfloor\!\rfloor^{\perp}, r : (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor)}\ \mathcal{D}}{(r(x).y\langle x\rangle.\bar{r}.r{\leftrightarrow}y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0}) \vdash y : T, r : (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor) \parallel y' : T^{\perp}}}{(\nu y y')(r(x).y\langle x\rangle.\bar{r}.r{\leftrightarrow}y \parallel y'(x).x\langle y'\rangle.x.x[].\mathbf{0}) \vdash r : (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})) \parr (\mathbf{1} \otimes \lfloor\!\lfloor S \rfloor\!\rfloor)}$$

    where

$$\begin{aligned}T &= (\lfloor\!\lfloor S \rfloor\!\rfloor \parr (\mathbf{1} \otimes \perp)) \otimes \lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \\ T^{\perp} &= (\lfloor\!\lfloor S \rfloor\!\rfloor^{\perp} \otimes (\perp \parr \mathbf{1})) \parr \lfloor\!\lfloor S \rfloor\!\rfloor\end{aligned}$$

and $\mathcal{D}$ is the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}}{x[].\mathbf{0} \vdash x : \mathbf{1}}}{x.x[].\mathbf{0} \vdash x : (\bot \parr \mathbf{1})}}{x\langle y' \rangle.x.x[].\mathbf{0} \vdash x : (\llbracket S \rrbracket^{\bot} \otimes (\bot \parr \mathbf{1})), y' : \llbracket S \rrbracket}}{y'(x).x\langle y' \rangle.x.x[].\mathbf{0} \vdash y' : (\llbracket S \rrbracket^{\bot} \otimes (\bot \parr \mathbf{1})) \parr \llbracket S \rrbracket}$$

– Subcase **send**: We have $\mathbf{send} : T \times !T.S \multimap S$ where

$$\begin{aligned} \llbracket T \times !T.S \multimap S \rrbracket^{\bot} &= \lceil T \times !T.S \multimap S \rceil \\ &= \lceil T \times !T.S \rceil^{\bot} \parr (1 \otimes \lceil S \rceil) \\ &= (\lceil T \rceil^{\bot} \parr \llbracket !T.S \rrbracket) \parr (1 \otimes \lceil S \rceil) \\ &= (\lceil T \rceil^{\bot} \parr (\lceil T \rceil \otimes \llbracket S \rrbracket)) \parr (1 \otimes \llbracket S \rrbracket^{\bot}) \end{aligned}$$

and $\llbracket \mathbf{send} \rrbracket_r^{\mathsf{v}} = r(y).y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \leftrightarrow y \vdash y : \llbracket S \rrbracket, r : \llbracket S \rrbracket^{\bot}}}{\bar{r}.r \leftrightarrow y \vdash y : \llbracket S \rrbracket, r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{y\langle x \rangle.\bar{r}.r \leftrightarrow y \vdash x : \lceil T \rceil^{\bot}, y : (\lceil T \rceil \otimes \llbracket S \rrbracket), r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \vdash y : (\lceil T \rceil^{\bot} \parr (\lceil T \rceil \otimes \llbracket S \rrbracket)), r : (1 \otimes \llbracket S \rrbracket^{\bot})}}{r(y).y(x).y\langle x \rangle.\bar{r}.r \leftrightarrow y \vdash r : (\lceil T \rceil^{\bot} \parr (\lceil T \rceil \otimes \llbracket S \rrbracket)) \parr (1 \otimes \llbracket S \rrbracket^{\bot})}$$

– Subcase **recv**: We have $\mathbf{recv} : ?T.S \multimap T \times S$ where

$$\begin{aligned} \llbracket ?T.S \multimap T \times S \rrbracket^{\bot} &= \lceil ?T.S \multimap T \times S \rceil \\ &= \lceil ?T.S \rceil^{\bot} \parr (1 \otimes \lceil T \times S \rceil) \\ &= (\lceil T \rceil^{\bot} \parr \lceil S \rceil^{\bot}) \parr (1 \otimes (\lceil T \rceil \otimes \lceil S \rceil)) \end{aligned}$$

and $\llbracket \mathbf{recv} \rrbracket_r^{\mathsf{v}} = r(x).x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \leftrightarrow x \vdash x : \lceil S \rceil^{\bot}, r : \lceil S \rceil}}{r\langle y \rangle.r \leftrightarrow x \vdash y : \lceil T \rceil^{\bot}, x : \lceil S \rceil^{\bot}, r : \lceil T \rceil \otimes \lceil S \rceil}}{\bar{r}.r\langle y \rangle.r \leftrightarrow x \vdash y : \lceil T \rceil^{\bot}, x : \lceil S \rceil^{\bot}, r : (1 \otimes (\lceil T \rceil \otimes \lceil S \rceil))}}{x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x \vdash x : (\lceil T \rceil^{\bot} \parr \lceil S \rceil^{\bot}), r : (1 \otimes (\lceil T \rceil \otimes \lceil S \rceil))}}{r(x).x(y).\bar{r}.r\langle y \rangle.r \leftrightarrow x \vdash r : (\lceil T \rceil^{\bot} \parr \lceil S \rceil^{\bot}) \parr (1 \otimes (\lceil T \rceil \otimes \lceil S \rceil))}$$

– Subcase **wait**: We have $\mathbf{wait} : \mathbf{end}_? \multimap \mathbf{1}$ where

$$\begin{aligned} \llbracket \mathbf{end}_? \multimap \mathbf{1} \rrbracket^{\bot} &= \lceil \mathbf{end}_? \multimap \mathbf{1} \rceil \\ &= \lceil \mathbf{end}_? \rceil^{\bot} \parr (\mathbf{1} \otimes \lceil \mathbf{1} \rceil) \\ &= \bot \parr (\mathbf{1} \otimes \mathbf{1}) \end{aligned}$$

and $[\![\mathbf{wait}]\!]^{\mathsf{v}}_r = r(x).x().\bar{r}.r[].\mathbf{0}$. We derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}}{r[].\mathbf{0} \vdash r : \mathbf{1}}}{\bar{r}.r[].\mathbf{0} \vdash r : \mathbf{1} \otimes \mathbf{1}}}{x().\bar{r}.r[].\mathbf{0} \vdash x : \bot, r : \mathbf{1} \otimes \mathbf{1}}}{r(x).x().\bar{r}.r[].\mathbf{0} \vdash r : \bot \parr (\mathbf{1} \otimes \mathbf{1})}$$

- **Case** $(\lambda x.M)$. We assume $[\![M]\!]^{\mathsf{m}}_r : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x : \lfloor\!\lfloor T \rfloor\!\rfloor, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}$ and derive

$$\dfrac{[\![M]\!]^{\mathsf{m}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x : \lfloor\!\lfloor T \rfloor\!\rfloor, r : \mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil}{r(x).[\![M]\!]^{\mathsf{m}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lfloor\!\lfloor T \rfloor\!\rfloor \parr (\mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil)}$$

- **Case** $(())$. We derive:

$$\dfrac{\dfrac{}{\mathbf{0} \vdash \varnothing}}{x[].\mathbf{0} \vdash x : \mathbf{1}}$$

- **Case** $(\mathbf{inl}\ W)$. We assume $[\![W]\!]^{\mathsf{v}}_r : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}$ and derive

$$\dfrac{[\![W]\!]^{\mathsf{v}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}}{r \lhd \mathrm{inl}.[\![W]\!]^{\mathsf{v}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lceil\!\lceil T \rceil\!\rceil \oplus \lceil\!\lceil U \rceil\!\rceil}$$

- **Case** $((V, W))$. We assume $[\![V]\!]^{\mathsf{v}}_x : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x : \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}$, $[\![W]\!]^{\mathsf{v}}_r : \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}$, and derive

$$\dfrac{\dfrac{[\![V]\!]^{\mathsf{v}}_x \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil \qquad [\![W]\!]^{\mathsf{v}}_r \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \lceil\!\lceil U \rceil\!\rceil}{[\![V]\!]^{\mathsf{v}}_x \parallel [\![W]\!]^{\mathsf{v}}_r \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil \parallel \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \lceil\!\lceil U \rceil\!\rceil}}{r[x].([\![V]\!]^{\mathsf{v}}_x \parallel [\![W]\!]^{\mathsf{v}}_r) \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil U \rceil\!\rceil}$$

Part 2.

- **Case** $(V\ W)$. We assume $[\![V]\!]^{\mathsf{v}}_{y'} : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil)$ and $[\![W]\!]^{\mathsf{v}}_{x'} : \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x' : \lceil\!\lceil T \rceil\!\rceil$ and derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \leftrightarrow y \vdash y : \lceil\!\lceil T \rceil\!\rceil \otimes (\bot \parr \lceil\!\lceil U \rceil\!\rceil^{\perp}), r : \mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil}}{y\langle x \rangle.r \leftrightarrow y \vdash x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, y : \lceil\!\lceil T \rceil\!\rceil \otimes (\bot \parr \lceil\!\lceil U \rceil\!\rceil^{\perp}), r : \mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil} \quad \mathcal{D}}{y\langle x \rangle.r \leftrightarrow y \parallel [\![V]\!]^{\mathsf{v}}_{y'} \parallel [\![W]\!]^{\mathsf{v}}_{x'} \vdash x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, y : \lceil\!\lceil T \rceil\!\rceil \otimes (\bot \parr \lceil\!\lceil U \rceil\!\rceil^{\perp}), r : 1 \otimes \lceil\!\lceil U \rceil\!\rceil \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil) \parallel \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x' : \lceil\!\lceil T \rceil\!\rceil}}{(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel [\![V]\!]^{\mathsf{v}}_{y'} \parallel [\![W]\!]^{\mathsf{v}}_{x'}) \vdash x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, r : \mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x' : \lceil\!\lceil T \rceil\!\rceil}}{(\nu x x')(\nu y y')(y\langle x \rangle.r \leftrightarrow y \parallel [\![V]\!]^{\mathsf{v}}_{y'} \parallel [\![W]\!]^{\mathsf{v}}_{x'}) \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil}$$

  where $\mathcal{D}$ is the derivation

$$\dfrac{[\![V]\!]^{\mathsf{v}}_{y'} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil) \qquad [\![W]\!]^{\mathsf{v}}_{x'} \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x' : \lceil\!\lceil T \rceil\!\rceil}{[\![V]\!]^{\mathsf{v}}_{y'} \parallel [\![W]\!]^{\mathsf{v}}_{x'} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr (\mathbf{1} \otimes \lceil\!\lceil U \rceil\!\rceil) \parallel \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x' : \lceil\!\lceil T \rceil\!\rceil}$$

- **Case** $(\mathbf{let}\ (x, y) = V\ \mathbf{in}\ M)$.
  We assume $[\![V]\!]^{\mathsf{v}}_{y'} : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil T' \rceil\!\rceil$ and $[\![M]\!]^{\mathsf{m}}_r : \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, y : \lceil\!\lceil T' \rceil\!\rceil^{\perp}, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}$ and derive

$$\dfrac{\dfrac{\dfrac{[\![M]\!]^{\mathsf{m}}_r \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, y : \lceil\!\lceil T' \rceil\!\rceil^{\perp}, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}}{y(x).[\![M]\!]^{\mathsf{m}}_r \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, y : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr \lceil\!\lceil T' \rceil\!\rceil^{\perp}, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}} \qquad [\![V]\!]^{\mathsf{v}}_{y'} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil T' \rceil\!\rceil}{y(x).[\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{y'} \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, y : \lceil\!\lceil T \rceil\!\rceil^{\perp} \parr \lceil\!\lceil T' \rceil\!\rceil^{\perp}, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp} \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, y' : \lceil\!\lceil T \rceil\!\rceil \otimes \lceil\!\lceil T' \rceil\!\rceil}}{(\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{y'}) \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \mathbf{1} \otimes \lfloor\!\lfloor U \rfloor\!\rfloor^{\perp}}$$

- **Case** (**absurd** V). We assume $[\![V]\!]^{\mathsf{v}}_{x'} : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{0}$, and derive:

$$\dfrac{\dfrac{}{x \triangleright \{\} \vdash r : \mathbf{1} \otimes \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}, x : \top} \qquad [\![V]\!]^{\mathsf{v}}_{x'} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{0}}{\dfrac{x \triangleright \{\} \parallel [\![V]\!]^{\mathsf{v}}_{x'} \vdash r : \mathbf{1} \otimes \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}, x : \top \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{0}}{(\nu x x')(x \triangleright \{\} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \mathbf{1} \otimes \lfloor\!\lfloor T \rfloor\!\rfloor^{\perp}}}$$

- **Case** (**let** $x = M$ **in** N).
  We assume $[\![M]\!]^{\mathsf{m}}_{x'} : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{1} \otimes \lceil\!\lceil T \rceil\!\rceil$ and $[\![N]\!]^{\mathsf{m}}_{r} : \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, r : \lceil\!\lceil U \rceil\!\rceil$ and derive

$$\dfrac{\dfrac{\dfrac{[\![N]\!]^{\mathsf{m}}_{r} \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \lceil\!\lceil T \rceil\!\rceil^{\perp}, r : \lceil\!\lceil U \rceil\!\rceil}{x.[\![N]\!]^{\mathsf{m}}_{r} \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \perp \parr \lceil\!\lceil T \rceil\!\rceil^{\perp}, r : \lceil\!\lceil U \rceil\!\rceil} \qquad [\![M]\!]^{\mathsf{m}}_{x'} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{1} \otimes \lceil\!\lceil T \rceil\!\rceil}{x.[\![N]\!]^{\mathsf{m}}_{r} \parallel [\![M]\!]^{\mathsf{m}}_{x'} \vdash \lfloor\!\lfloor \Delta \rfloor\!\rfloor, x : \perp \parr \lceil\!\lceil T \rceil\!\rceil^{\perp}, r : \lceil\!\lceil U \rceil\!\rceil \parallel \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, x' : \mathbf{1} \otimes \lceil\!\lceil T \rceil\!\rceil}}{(\nu x x')(x.[\![N]\!]^{\mathsf{m}}_{r} \parallel [\![M]\!]^{\mathsf{m}}_{x'}) \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, \lfloor\!\lfloor \Delta \rfloor\!\rfloor, r : \lceil\!\lceil U \rceil\!\rceil}$$

- **Case** (V). We assume $[\![V]\!]^{\mathsf{v}}_{r} : \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lceil\!\lceil T \rceil\!\rceil$ and derive

$$\dfrac{[\![V]\!]^{\mathsf{v}}_{r} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \lceil\!\lceil T \rceil\!\rceil}{\bar{r}.[\![V]\!]^{\mathsf{v}}_{r} \vdash \lfloor\!\lfloor \Gamma \rfloor\!\rfloor, r : \mathbf{1} \otimes \lceil\!\lceil T \rceil\!\rceil}$$

Part 3. The cases are all by immediate induction. □

Lemma C.5. *Let F be an HGV∗ evaluation context and r a result endpoint. Then there exists a process context $[\![F]\!]^{\mathsf{f}}_{r}$ and a result endpoint $v = \mathsf{hr}(F, r)$ for the hole such that for all M we have that $[\![F[M]]\!]^{\mathsf{c}}_{r} = [\![F]\!]^{\mathsf{f}}_{r}[[\![M]\!]^{\mathsf{m}}_{v}]$.*

Proof. By induction on the structure of F. □

In the above lemma, if F is the empty context then $v = r$. Otherwise v is a variable bound by the process context $[\![F]\!]^{\mathsf{f}}_{r}$.

Lemma C.6 (Operational Correspondence, Terms)**.** *If M is a well-typed term:*

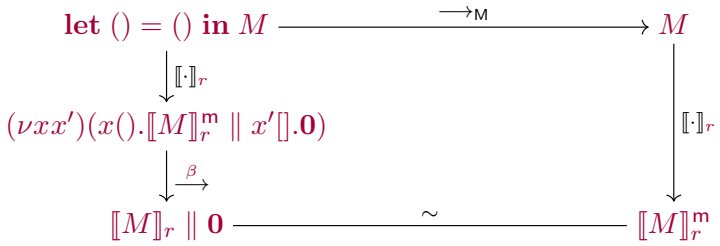
(1) *If $M \longrightarrow_{\mathsf{M}} M'$, then there exists a P such that $[\![M]\!]^{\mathsf{m}}_{r} \overset{\beta^+}{\Longrightarrow}_{\alpha} P$ and $P \approx_{\alpha} [\![M']\!]^{\mathsf{m}}_{r}$; and*
(2) *if $[\![M]\!]^{\mathsf{m}}_{r} \xrightarrow{\beta} P$, then there exists an M' and a P' such that $M \longrightarrow_{\mathsf{M}} M'$ and $P \overset{\beta^*}{\Longrightarrow}_{\alpha} P'$ and $P' \approx_{\alpha} [\![M']\!]^{\mathsf{m}}_{r}$.*

Proof.

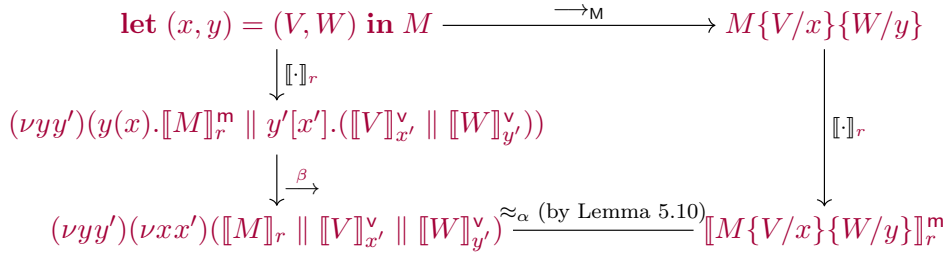
(1) By induction on the reduction $M \longrightarrow_{\mathsf{M}} M'$.
  Case (E-Lam).

$$\begin{array}{ccc}
(\lambda x.M)\ V & \xrightarrow{\quad\longrightarrow_{\mathsf{M}}\quad} & M\{V/x\} \\
\Big\downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_{r}} & & \\
(\nu x x')(\nu y y')(y\langle x\rangle.r \leftrightarrow y \parallel y'(x).[\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \Big\downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_{r}} \\
\Big\downarrow{\scriptstyle \beta \longrightarrow \alpha \longrightarrow} & & \\
(\nu x x')(\nu y y')(r \leftrightarrow y \parallel [\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\
\Big\downarrow{\scriptstyle \alpha \longrightarrow} & & \\
(\nu x x')([\![M]\!]^{\mathsf{m}}_{r} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \xrightarrow{\quad\approx_{\alpha} \text{ (by Lemma 5.10)}\quad} & [\![M\{V/x\}]\!]^{\mathsf{m}}_{r}
\end{array}$$

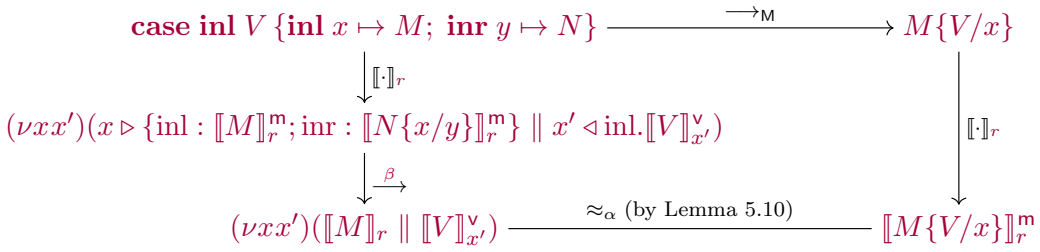
Case (E-Unit).



Case (E-Pair).

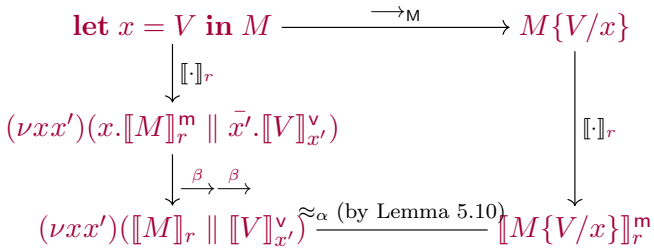


Case (E-Inl).



Case (E-Inr). As E-Inl.

Case (E-Let).



Case (E-Lift). The induction hypothesis yields the reasoning steps depicted by the first diagram, which we use, together with HGV's E-Lift and HCP's Str-Res and Str-Par2,

to justify the second diagram:

$$\begin{array}{ccc} M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M' \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} & & \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} \\ [\![M]\!]^{\mathsf{m}}_r & \xrightarrow{\overset{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha} & [\![M']\!]^{\mathsf{m}}_r \end{array} \qquad \begin{array}{ccc} \mathbf{let}\ x = E[M]\ \mathbf{in}\ N & \xrightarrow{\longrightarrow_{\mathsf{M}}} & \mathbf{let}\ x = E[M']\ \mathbf{in}\ N \\ \downarrow{\scriptstyle [\![\cdot]\!]_r} & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\ (\nu x x')(x.[\![N]\!]^{\mathsf{m}}_r \parallel [\![M]\!]^{\mathsf{m}}_{x'}) & \xrightarrow{\overset{\beta^+}{\Longrightarrow}_\alpha \approx_\alpha} & (\nu x x')(x.[\![N]\!]^{\mathsf{m}}_r \parallel [\![M']\!]^{\mathsf{m}}_{x'}) \end{array}$$

(2) By induction on M.

Case $(U\ V)$. There are two well-typed cases for U: either $U = z$ for some z; or $U = \lambda x.M$ for some x and M. If $U = z$, we have $(\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel z{\leftrightarrow}y' \parallel [\![V]\!]^{\mathsf{v}}_{x'}) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = \lambda x.M$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc} (\lambda x.M)\ V & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\ \downarrow{\scriptstyle [\![\cdot]\!]^{\mathsf{m}}_r} & & \\ (\nu x x')(\nu y y')(y\langle x\rangle.r{\leftrightarrow}y \parallel y'(x).[\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\ \downarrow{\scriptstyle \beta \longrightarrow \alpha \longrightarrow} & & \\ (\nu x x')(\nu y y')(r{\leftrightarrow}y \parallel [\![M]\!]^{\mathsf{m}}_{y'} \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & & \\ \downarrow{\scriptstyle \alpha \longrightarrow} & & \\ (\nu x x')([\![M]\!]^{\mathsf{m}}_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \xrightarrow{\approx_\alpha \text{ (by Lemma 5.10)}} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r \end{array} \qquad \text{with } [\![\cdot]\!]^{\mathsf{m}}_r \text{ from } M\{V/x\} \text{ down}$$

Hence, $M' = M\{V/x\}$.

Case $(\mathbf{let}\ () = U\ \mathbf{in}\ M)$. There are two well-typed cases for U: either $U = z$ for some z; or $U = ()$. If $U = z$, we have $(\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel x'{\leftrightarrow}z) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = ()$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc} \mathbf{let}\ () = ()\ \mathbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M \\ \downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\ (\nu x x')(x().[\![M]\!]^{\mathsf{m}}_r \parallel x'[].\mathbf{0}) & & \\ \downarrow{\scriptstyle \beta \longrightarrow} & & \\ [\![M]\!]_r \parallel \mathbf{0} & \xrightarrow{\sim} & [\![M]\!]^{\mathsf{m}}_r \end{array} \qquad \text{with } [\![\cdot]\!]_r \text{ from } M \text{ down}$$

Hence, $M' = M$.

Case $(\mathbf{let}\ (x, y) = U\ \mathbf{in}\ M)$. There are two well-typed cases for U: either $U = z$ for some z, or $U = (V, W)$. If $U = z$, we have $(\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel y'{\leftrightarrow}z) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = (V, W)$. The only possible β-transition is the one in the

following diagram:

$$\begin{array}{ccc}
\mathbf{let}\ (x,y) = (V,W)\ \mathbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\}\{W/y\} \\
\downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\
(\nu y y')(y(x).[\![M]\!]^{\mathsf{m}}_r \parallel y'[x'].([\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'})) & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\
\downarrow{\scriptstyle \beta\longrightarrow} & & \\
(\nu y y')(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'} \parallel [\![W]\!]^{\mathsf{v}}_{y'}) & \overset{\approx_\alpha \text{ (by Lemma 5.10)}}{\longrightarrow} & [\![M\{V/x\}\{W/y\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case ($\mathbf{case}\ U\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ x \mapsto N\}$). There are two well-typed cases for U: either $U = z$ for some z; or $U = \mathbf{inl}\ V$. If $U = z$, we have $(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x' \leftrightarrow z) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise. Therefore, $U = \mathbf{inl}\ V$. The only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc}
\mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\
(\nu x x')(x \triangleright \{\mathrm{inl} : [\![M]\!]^{\mathsf{m}}_r; \mathrm{inr} : [\![N\{x/y\}]\!]^{\mathsf{m}}_r\} \parallel x' \triangleleft \mathrm{inl}.[\![V]\!]^{\mathsf{v}}_{x'}) & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\
\downarrow{\scriptstyle \beta\longrightarrow} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.10)}}{\longrightarrow} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Case ($\mathbf{absurd}\ U$). There is only one well-typed case for U: $U = z$ for some z. However, $(\nu x x')(x \triangleright \{\} \parallel x' \leftrightarrow z) \overset{\beta}{\not\longrightarrow}$, which contradicts our premise.

Case ($\mathbf{let}\ x = M\ \mathbf{in}\ N$). There are two possible cases: either $M = V$; or $[\![M]\!]^{\mathsf{m}}_{x'} \xrightarrow{\beta} P$ for some P. If M is a value, the only possible β-transition is the one in the following diagram:

$$\begin{array}{ccc}
\mathbf{let}\ x = V\ \mathbf{in}\ M & \xrightarrow{\longrightarrow_{\mathsf{M}}} & M\{V/x\} \\
\downarrow{\scriptstyle [\![\cdot]\!]_r} & & \\
(\nu x x')(x.[\![M]\!]^{\mathsf{m}}_r \parallel \bar{x}'.[\![V]\!]^{\mathsf{v}}_{x'}) & & \downarrow{\scriptstyle [\![\cdot]\!]_r} \\
\downarrow{\scriptstyle \beta\longrightarrow\ \beta\longrightarrow} & & \\
(\nu x x')([\![M]\!]_r \parallel [\![V]\!]^{\mathsf{v}}_{x'}) & \overset{\approx_\alpha \text{ (by Lemma 5.10)}}{\longrightarrow} & [\![M\{V/x\}]\!]^{\mathsf{m}}_r
\end{array}$$

Otherwise, if $[\![M]\!]^{\mathsf{m}}_{x'} \xrightarrow{\beta} P$ for some P, the induction hypothesis gives us an M' such that $M \longrightarrow_{\mathsf{M}} M'$ and $P \approx [\![M']\!]^{\mathsf{m}}_r$. We apply HGV's E-Lift and HCP's Str-Res and Str-Par2.

Case (V). We have $\bar{r}.[\![V]\!]^{\mathsf{v}}_r \overset{\beta}{\not\longrightarrow}$, which contradicts our premise. □

Theorem 5.11 (Operational Correspondence)**.** *Suppose $\mathcal{C}$ is a well-typed configuration.*

(1) *(Preservation of reductions) If $\mathcal{C} \longrightarrow \mathcal{C}'$, then there exists a P such that $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \overset{\beta^+}{\Longrightarrow}_\alpha P$ and $P \approx_\alpha [\![\mathcal{C}']\!]^{\mathsf{c}}_r$; and*
(2) *(Reflection of transitions)*
- *if $[\![\mathcal{C}]\!]^{\mathsf{c}}_r \xrightarrow{\alpha} P$, then $P \approx_\alpha [\![\mathcal{C}]\!]^{\mathsf{c}}_r$; and*

- *if* $[\![\mathcal{C}]\!]_r^{\mathsf{c}} \xrightarrow{\beta} P$, *then there exists a* $\mathcal{C}'$ *and a* P' *such that* $\mathcal{C} \longrightarrow \mathcal{C}'$ *and* $P \overset{\beta^*}{\Longrightarrow}_\alpha P'$ *and* $P' \approx_\alpha [\![\mathcal{C}']\!]_r^{\mathsf{c}}$. *Furthermore,* $\mathcal{C}'$ *is unique up to structural congruence.*

Proof.

(1) By induction on the reduction $\mathcal{C} \longrightarrow \mathcal{C}'$. We implicitly make use of Lemma C.5 throughout the proof in order to recast the translation of a plugged evaluation context $[\![F[M]]\!]_r^{\mathsf{c}}$ into the plugging of the translated evaluation context with the translation of the plugged term $[\![F]\!]_r^{\mathsf{f}}[\![M]\!]_v^{\mathsf{m}}]$ where $v = \mathsf{hr}(F, r)$.

Case (E-Reify-Fork).

$$\begin{array}{ccc}
F[\mathbf{fork}\ V] & \longrightarrow & (\nu x x')(F[x] \parallel \circ\ V\ x') \\
\downarrow [\![\cdot]\!]_r^{\mathsf{c}} & & \\
[\![F]\!]_r^{\mathsf{f}}[(\nu z z')(\nu y y')\begin{pmatrix} y\langle z\rangle.v \leftrightarrow y \parallel \\ (\nu x x')\begin{pmatrix} y'(w).x\langle w\rangle.\bar{y}'.y' \leftrightarrow x \parallel \\ x'(w).w\langle x'\rangle.w.w[].\mathbf{0} \end{pmatrix} \parallel \\ [\![V]\!]_{z'}^{\mathsf{v}} \end{pmatrix}] & & \downarrow [\![\cdot]\!]_r^{\mathsf{c}} \\
\downarrow \xrightarrow{\beta}\xrightarrow{\beta}\xrightarrow{\alpha} & & \\
[\![F]\!]_r^{\mathsf{f}}[(\nu z z')\begin{pmatrix} (\nu x x')\begin{pmatrix} \bar{v}.v \leftrightarrow x \parallel \\ z\langle x'\rangle.z.z[].\mathbf{0} \end{pmatrix} \parallel \\ [\![V]\!]_{z'}^{\mathsf{v}} \end{pmatrix}] & \approx_\alpha & (\nu x x')\begin{pmatrix} [\![F]\!]_r^{\mathsf{f}}[\bar{v}.v \leftrightarrow x] \parallel \\ (\nu y y')\begin{pmatrix} (\nu w w')(\nu z z')\begin{pmatrix} z\langle w\rangle.y \leftrightarrow z \parallel \\ [\![V]\!]_{z'}^{\mathsf{v}} \parallel \\ w' \leftrightarrow x' \end{pmatrix} \parallel \\ y'.y'[].\mathbf{0} \end{pmatrix} \end{pmatrix}
\end{array}$$

The endpoint $v = \mathsf{hr}(F, r)$. The final two terms are bisimilar by Lemma C.4.

Case (E-Reify-Link).

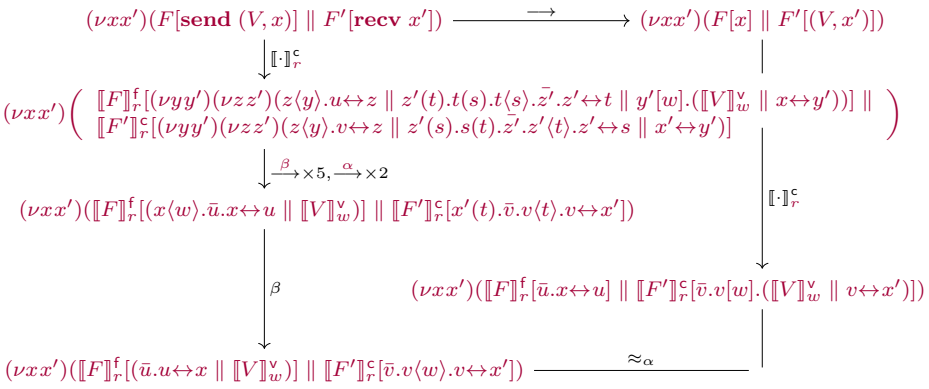
$$\begin{array}{ccc}
\circ\ E[\mathbf{link}\ (x, y)] & \longrightarrow & (\nu z z')(x \overset{z}{\leftrightarrow} y \parallel \circ\ E[z']) \\
\downarrow [\![\cdot]\!]_r^{\mathsf{c}} & & \\
(\nu a a')([\![E]\!]_r^{\mathsf{m}}[(\nu z z')(\nu w w')(w\langle z\rangle.v \leftrightarrow w \parallel w'(t).t(s).\bar{w}'.w'().s \leftrightarrow t \parallel z'\langle x\rangle.y \leftrightarrow z' \parallel \bar{a}'.a'[].\mathbf{0}))] & & \\
\downarrow \xrightarrow{\beta}\xrightarrow{\beta}\xrightarrow{\alpha}\xrightarrow{\alpha} & & \downarrow [\![\cdot]\!]_r^{\mathsf{c}} \\
(\nu a a')([\![E]\!]_r^{\mathsf{m}}[\bar{v}.v().x \leftrightarrow y] \parallel \bar{a}'.a'[].\mathbf{0}) & \approx_\alpha & (\nu z z')(\bar{z}.z().x \leftrightarrow y \parallel (\nu a a')([\![E[v \leftrightarrow z']]\!]_a^{\mathsf{m}} \parallel \bar{a}'.a'[].\mathbf{0}))
\end{array}$$

The endpoint $v = \mathsf{hr}(F, r)$.

Case (E-Comm-Link).

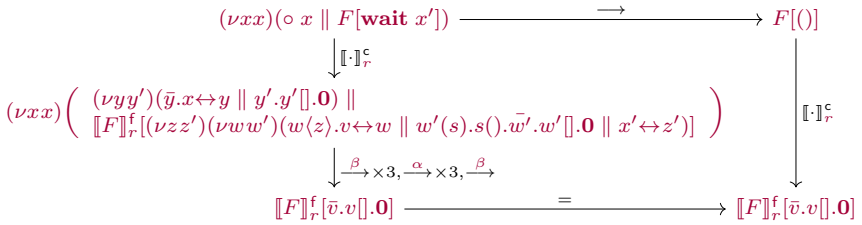
$$\begin{array}{ccc}
(\nu z z')(\nu x x')(x \overset{z}{\leftrightarrow} y \parallel \circ\ z' \parallel \phi\ M) & \longrightarrow & \phi\ (M\{y/x'\}) \\
\downarrow [\![\cdot]\!]_r^{\mathsf{c}} & & \\
(\nu z z')(\nu x x')(\bar{z}.z().x \leftrightarrow y \parallel (\nu w w')(z' \leftrightarrow w \parallel w'.w'[].\mathbf{0}) \parallel [\![\phi\ M]\!]_r^{\mathsf{c}}) & & \downarrow [\![\cdot]\!]_r^{\mathsf{c}} \\
\downarrow \xrightarrow{\alpha}, \xrightarrow{\beta} \times 3 & & \\
[\![\phi\ M]\!]_r^{\mathsf{c}}\{y/x'\} & \approx_\alpha & [\![\phi\ M\{y/x'\}]\!]_r^{\mathsf{c}}
\end{array}$$

Case (E-Comm-Send).



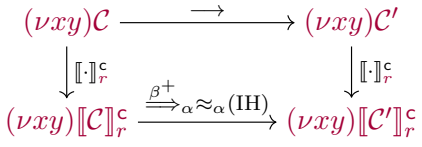
The endpoint $u = \mathsf{hr}(F, r)$ and the endpoint $v = \mathsf{hr}(F', r)$.

Case (E-Comm-Close).

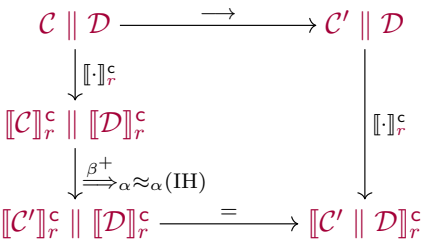


The endpoint $v = \mathsf{hr}(F, r)$.

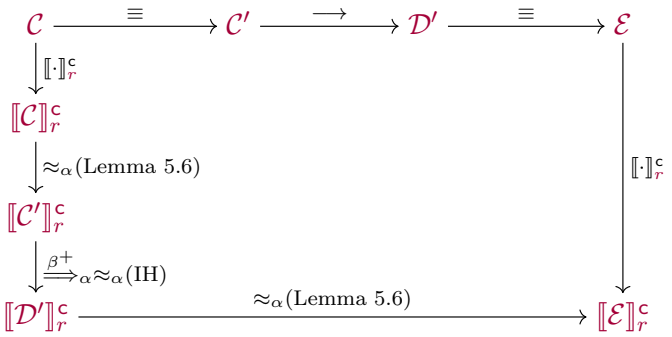
Case (E-Res).



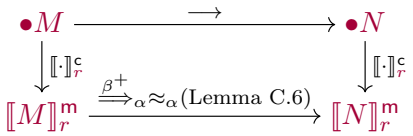
Case (E-Par).



Case (E-Equiv).



Case (E-Lift-M). The cases for $\phi = \bullet$ and $\phi = \circ$ are similar; here we show the case for $\bullet$.



(2) Reflection of α-transitions is trivial as α-transition is included in α-bisimulation. Reflection of β-transitions is by induction on C; as with Lemma C.6, the only well-typed β-transitions that can occur for each case are those specified in the simulation case. □

4.3 Discussion

This section proceeds as follows:

- In § 4.3.1, I present an alternative formulation of GV that uses cuts, rather than lock typing, and argue this significantly simplifies the metatheory of GV.
- In § 4.3.2, I discuss the relation between GV with cuts, lock typing, and hypersequents.

4.3.1 Cooking GV With Cut

In Lindley and Morris' GV, name restriction and parallel composition are separate configuration constructs. However, as discussed, this breaks type preservation for the structural congruence, which significantly complicates the metatheory.

This section presents an alternative formulation of GV, which we refer to as "GV with Cut" or GV_{Cut}, which more closely corresponds to CP. While this version is not preferable to Hypersequent GV, presented in this chapter, it represents my view of how GV should have been formulated prior to the introduction of hypersequents, and is helpful when examining Lindley and Morris' lock typing.

In this section, the terms and types of GV_{Cut} are printed in *pink* and *green*, respectively, and both are rendered in a sans-serif font, and any relations, such as typing and reduction, are marked by a subscript "GVC".

To construct GV_{Cut}, we replace name restriction and parallel composition with a cut. For configuration typing, we replace the rules TG-New, TG-Connect$_1$, and TG-Connect$_2$ with the rule TG-Cut.

$$\begin{array}{rcl} C, D & ::= & \cancel{(\nu x \bar{x}) C} \\ & \mid & \cancel{C \parallel D} \\ & \mid & (\nu x \bar{x})(C \parallel D) \end{array}$$

$$\text{TG-Cut} \quad \frac{\Gamma, x : S \vdash_{GVC} C : R \qquad \Delta, \bar{x} : \overline{S} \vdash_{GVC} D : R'}{\Gamma, \Delta \vdash_{GVC} (\nu x \bar{x})(C \parallel D) : R \sqcap R'}$$

For the structural congruence, we replace the rules SC-ParAssoc, SC-ParComm, SC-NewAssoc, SC-NewSwap, and SC-ScopeExt with the rules SC-CutComm and SC-CutAssoc.

$$\begin{array}{lll} (\nu x \bar{x})(C_1 \parallel C_2) & \equiv_{GVC} (\nu \bar{x} x)(C_2 \parallel C_1) & \text{SC-CutComm} \\ (\nu x \bar{x})((\nu y \bar{y})(C_1 \parallel C_2) \parallel C_3) & \equiv_{GVC} (\nu y \bar{y})((\nu x \bar{x})(C_1 \parallel C_3) \parallel C_2) & \text{SC-CutAssoc} \\ & \text{where } x \notin C_2 \text{ and } y \notin C_3 & \end{array}$$

In GV_{Cut}, the structural congruence preserves types. Much like the rules themselves, the proof is nearly identical to the proof of preservation for CP's structural congruence, and is left as an exercise.

Lemma 4.1. *If* $\Gamma \vdash_{GVC} C : R$ *and* $C \equiv_{GVC} D$, *then* $\Gamma \vdash_{GVC} D : R$.

Since $\mathrm{GV}_{\mathrm{Cut}}$'s structural congruence preserves types, it can be harmlessly embedded into the reduction relation, as per EG-Equiv.

$$\textsc{EG-Equiv}\quad \frac{C \equiv_{\mathsf{GVC}} C' \qquad C' \longrightarrow_{\mathsf{GVC}} D' \qquad D' \equiv_{\mathsf{GVC}} D}{C \longrightarrow_{\mathsf{GVC}} D}$$

This small change tightens $\mathrm{GV}_{\mathrm{Cut}}$'s correspondence with CP and considerably simplifies its metatheory, as there is no longer any need to prove preservation up to equivalence [e.g. Fowler, 2019, Theorem 2, pp. 40–43].

4.3.2 Lock Types, Cuts, and Hypersequents

This section discusses the differences between GV with lock types, cuts, and hypersequents, and argues that, while lock types and hypersequents both permit the syntax for configurations to separate name restriction from parallel composition, lock types are, in effect, more complicated cuts, whereas hypersequents capture parallelism.

We compare the configuration typing rules for GV (Figure I.5), $\mathrm{GV}_{\mathrm{Cut}}$ (§ 4.3.1), and HGV (Figure I.3). In this section, the terms and types of both GV and $\mathrm{GV}_{\mathrm{Cut}}$ are printed in *pink* and *green*, respectively, both are rendered in a sans-serif font, and any relations, such as typing and reduction, are marked by subscript "GV" and "GVC", respectively.

Let us begin with an observation. Both GV and HGV use the exact same syntax for configurations, but the translation from HGV to GV (in § I.4, under "Translating HGV to GV") shows a gap between the two systems. Every GV configuration is typeable in HGV, but there are HGV configurations that are not typeable in GV.

What is the relation between GV's configurations and $\mathrm{GV}_{\mathrm{Cut}}$'s configurations? As we will see, GV's parallel composition is, in essence, a cut, and therefore the two are identical.

Let us begin by discussing GV's lock types in slightly more detail. GV splits cut into name restriction and parallel composition:

$$\begin{array}{lll} C, D ::= \ldots & & \\ \quad \mid \cancel{(\nu x \bar{x})(C \parallel D)} & \text{cut} \\ \quad \mid (\nu x \bar{x}) C & \text{name restriction} \\ \quad \mid C \parallel D & \text{parallel composition} \end{array}$$

As naively decomposing cuts would break GV's tree connection structure and, more importantly, deadlock freedom, GV ensures these properties by lock typing. It extends runtime typing environments with locked channel

type assignments.[3]

$$\Gamma ::= \cdots \mid \Gamma, \langle x, \bar{x} \rangle : S^{\#}$$

The special type assignment "$\langle x, \bar{x} \rangle : S^{\#}$" represents a channel with endpoints x and $\bar{x}$ which has been created, but has not yet been split across a parallel composition. Hence, it is *locked*.

When a locked channel is split across a parallel composition, it becomes unlocked, and the endpoints become available for use. Each parallel composition must split exactly one locked channel. This is guaranteed by the typing rules for name restriction and parallel composition, reproduced below from Figure I.5 (eliding the superfluous TG-Connect$_2$).

$$\text{TG-New}\ \frac{\Gamma, \langle x, \bar{x} \rangle : S^{\#} \vdash_{GV} C : R}{\Gamma \vdash_{GV} (\nu x \bar{x}) C : R} \qquad \text{TG-Connect}_1\ \frac{\Gamma, x : S \vdash_{GV} C : R \qquad \Delta, \bar{x} : \overline{S} \vdash_{GV} D : R'}{\Gamma, \Delta, \langle x, \bar{x} \rangle : S^{\#} \vdash_{GV} C \parallel D : R \sqcap R'}$$

The rule TG-New creates a locked channel. The rule TG-Connect$_1$ splits a locked channel across a parallel composition. As the typing environments for term typing cannot contain locked channels, every locked channel must be split across some parallel composition.

There is a tension between the structural congruence and the typing rules. The typing rules require that each parallel composition splits exactly one locked channel, but the structural congruence—specifically, SC-ParAssoc—does not respect this invariant. Hence, GV's structural congruence does not preserve types. For instance, the following configuration is well-typed:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x : S \vdash_{GV} C_1 : R_1 \qquad \dfrac{\Gamma_2, y : S' \vdash_{GV} C_2 : R_2 \qquad \Gamma_3, \bar{x} : \overline{S}, \bar{y} : \overline{S'} \vdash_{GV} C_3 : R_3}{\Gamma_2, \Gamma_3, \bar{x} : \overline{S}, \langle y, \bar{y} \rangle : S' \vdash_{GV} C_2 \parallel C_3 : R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, \bar{x} \rangle : S, \langle y, \bar{y} \rangle : S' \vdash_{GV} C_1 \parallel (C_2 \parallel C_3) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3, \langle x, \bar{x} \rangle : S \vdash_{GV} (\nu y \bar{y})(C_1 \parallel (C_2 \parallel C_3)) : R_1 \sqcap R_2 \sqcap R_3}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash_{GV} (\nu x \bar{x})(\nu y \bar{y})(C_1 \parallel (C_2 \parallel C_3)) : R_1 \sqcap R_2 \sqcap R_3}$$

However, by SC-ParAssoc

$$(\nu x \bar{x})(\nu y \bar{y})(C_1 \parallel (C_2 \parallel C_3)) \equiv_{GV} (\nu x \bar{x})(\nu y \bar{y})((C_1 \parallel C_2) \parallel C_3)$$

The configuration on the right-hand side is not typeable, as the left-most parallel composition splits *no* locked channels, and the right-most parallel composition splits *two* locked channels.

Consequently, GV, as well as a significant portion of work based on Lindley and Morris' GV, proves some variation of the following

[3] Technically, the presentation of GV in § 4.2 adds $S^{\#}$ as a runtime session type, rather than as part of a special type assignment. Nonetheless, the two presentations are equivalent. Lock types cannot occur in user programs, and so cannot occur as part of any other type. Hence, they already occur only as the top-most connective in a type assignment.

proposition, which states that if we break typing before some reduction, we can restore it afterwards.

Proposition 4.2. *If* $\Gamma \vdash_{GV} C : R$ *and* $C \equiv_{GV} C'$ *and* $C' \longrightarrow_{GV} D'$*, then there exists some* D *such that* $D' \equiv_{GV} D$ *and* $\Gamma \vdash_{GV} D : R$*.*

Proof. See Fowler [2019, pp. 41-43, Theorem 2]. Fowler's proof is for a variant of GV *without* link or link threads, but the only rule of structural congruence that breaks typing is SC-ParAssoc. Hence, the proof of Theorem 2 trivially extends to the system *with* link and link threads. Fowler's proof is stronger, as it restores typing before and after the reduction.

□

GV's parallel composition is *covertly* a cut, albeit one that leaves the channel *implicit*.[4] Like cut, parallel composition must split exactly one channel, and, while cut restricts a channel by removing its endpoints from the typing environment, parallel composition restricts a channel by locking its endpoints. This is just as effective, since there is no way to use the endpoints of a locked channel. The locking mechanism keeps the endpoints around for the sole purpose of being removed by name restriction, which only adds unnecessary complexity.

To formalise this fact, we define a translation on configuration typing derivations from GV to GV_{Cut}, written $[\![\cdot]\!]_{GVC}$, which removes all lock types from typing environments, removes all name restrictions, and replaces parallel compositions with cuts. All uses of the rule TG-New are removed:

$$\left[\!\!\left[\dfrac{\begin{array}{c}\delta\\ \Gamma, \langle x, \bar{x}\rangle : S^{\#} \vdash_{GV} C : R\end{array}}{\Gamma \vdash_{GV} (\nu x\bar{x})C : R}\ \text{TG-New} \right]\!\!\right]_{GVC} \triangleq \begin{array}{c}[\![\delta]\!]_{GVC}\\ \Gamma \vdash_{GVC} C : R\end{array}$$

Each use of the rule TG-Connect_1 is replaced with a use of the rule TG-Cut:

$$\left[\!\!\left[\dfrac{\begin{array}{cc}\delta_1 & \delta_2\\ \Gamma, x : S \vdash_{GV} C : R & \Delta, \bar{x} : \overline{S} \vdash_{GV} D : R'\end{array}}{\Gamma, \Delta, \langle x, \bar{x}\rangle : S^{\#} \vdash_{GV} C \parallel D : R \sqcap R'}\ \text{TG-Connect}_1 \right]\!\!\right]_{GVC}$$

$$\shortmid\!\triangleright$$

$$\dfrac{\begin{array}{cc}[\![\delta_1]\!]_{GVC} & [\![\delta_2]\!]_{GVC}\\ \Gamma, x : S \vdash_{GVC} C : R & \Delta, \bar{x} : \overline{S} \vdash_{GVC} D : R'\end{array}}{\Gamma, \Delta \vdash_{GVC} (\nu x\bar{x})(C \parallel D) : R \sqcap R'}\ \text{TG-Cut}$$

The translation acts as the identity on threads, terms, and types. We believe that the translation preserves reduction, but proving this matter

[4] Tellingly, Lindley and Morris [2015, § 3.3] write "$\parallel_x$" when they need the channel name.

would be tedious and not worthwhile. The intuition is that, under the translation $[\![\cdot]\!]_{GVC}$, any use of SC-LinkComm is preserved, any use of SC-ParComm becomes SC-CutComm, and all uses of SC-NewAssoc, SC-NewSwap, or SC-ScopeExt become reflexivity. The tedious part is the proof that all type breaking uses of SC-ParAssoc are repaired in such a way that the source and target can be proven equivalent using SC-CutAssoc. The structure of this proof depends heavily on the computational content of the proof that repairs ill-typed configurations.

4.4 Conclusion

In this chapter, we introduced Hypersequent GV, as well as GV, with their typing rules, reduction semantics, and their metatheory. Hypersequent GV rectifies a design flaw that complicates all previous metatheory of GV by dropping lock typing in favour of hypersequents. For HGV, we proved *preservation* (Theorem I.3.3), the *tree-structure* of connections in configurations (Theorem I.3.14), *global progress* (Theorem I.3.20), the *diamond property* (Theorem I.3.21), and *termination* (Theorem I.3.22). We related HGV to GV by means of a translation from GV to HGV (Theorem I.4.3) and a translation from HGV to GV (Corollary I.4.7). We related HGV to HCP by means of two translations, which translate HGV to HCP via fine-grain call-by-value HGV, and proved that the latter translation preserves types (Lemma I.5.9), and that it gives rise to a sound and complete operational correspondence (Theorem I.5.11). Finally, I introduced a variant of GV which uses cuts, instead of lock typing, and compared Hypersequent GV to GV with lock typing and GV with cuts.

In the future, it would be interesting to extend Hypersequent GV with the extensions for GV described in previous work, such as unlimited types [Lindley and Morris, 2015], fixed points [Lindley and Morris, 2016b], polymorphism [Lindley and Morris, 2017], and exceptions [Fowler et al., 2019]. Furthermore, it would be interesting to describe the construction of GV compositionally, as the combination of a process calculus and a λ-calculus.

Part III

Prioritise The Best Variation

Chapter 5

Priority Classical Processes & Priority Good Variation

This chapter presents both Priority CP (PCP) and Priority GV (PGV), variants of CP and GV that use priorities [Kobayashi, 2006, Padovani, 2014]. PCP is a session-typed process calculus, based on CP, which was introduced by Dardha and Gay [2018]. PGV is a session-typed functional language, based on GV, which was introduced by Kokke and Dardha [2021a].

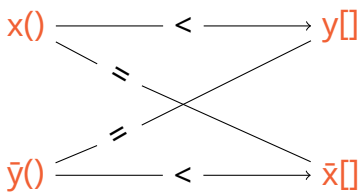
Priorities are an extra-logical mechanism for guaranteeing deadlock freedom, which arose from research on typed π-calculus rather than logic. In a system using priorities, each session type constructor—or, equivalently each communication action—is annotated with some priority, which the type system uses to ensure deadlock freedom. Let us consider two intuitions for the mechanism by which priorities work.

Concretely, *priorities* are natural numbers that represent the time at which some communication happens, i.e. for the process $x[y].\, z(w).\, P$, we might assign the action $x[y]$ priority 1, because it happens at time 1, and $z(w)$ priority 2, since it necessarily happens at some later time. Type checking requires that the priority assignment reflects this dependency, and that dual actions take place at the same time. In a session-typed system, this suffices to rule out deadlocks, as there are no valid priority assignments for deadlocking processes. For instance, for the process

$$(\nu x\bar{x})(\nu y\bar{y})(x().\, y[].\, 0 \parallel \bar{y}().\, \bar{x}[].\, 0)$$

there exists no valid priority assignment, since, by duality, $x()$ must happen at the same time as $\bar{x}[]$, and $y()$ must happen at the same time as $\bar{y}[]$, but, by dependency, $x()$ must happen before $y[]$, and $\bar{y}()$ must happen before $\bar{x}[]$. Represented visually, we can see that the these requirements

are essentially cyclic:



Programs are annotated with a lower and and upper *priority bound*, which represent the time at which the program begins to communicate, and the time it finishes. Each priority bound is either a concrete priority, or $\top$, or $\bot$, where the latter two are the supremum and infimum, respectively, i.e. $\top$ is greater than any priority, and $\bot$ is smaller than any priority. The supremum and infimum are useful in a variety of circumstances. For instance, a program that does not communicate is assigned the lower bound $\top$ and the upper bound $\bot$, i.e. it never begins communicating, and has already finished. Alternatively, $\bot$ and $\top$ may be used as lower and upper bounds, respectively, to represent unknowns. For instance, a program that loops might not have a concrete upper bound, and can be assigned upper bound $\top$, to mean that we do not know if or when it finishes communicating.

Abstractly, we can view priority metavariables as names for actions. (Here, we say priority metavariables, rather than priorities, to emphasize that we are referring to the names, rather than the concrete numbers with which they will be instantiated.) Type checking imposes equality and inequality constraints on priority metavariables, which define the deep dependency graph by analogy to the shallow dependency graph as defined for, e.g. HCP in § 3.2.4. For instance, the visual representation of constraints, shown above, is isomorphic to the dependency graph for the same process. If the deep dependency graph is essentially acyclic, there exists a topological sort of the priority metavariables, which we can use to instantiate them with concrete priorities. This is the mechanism behind priority inference, which we discuss in § 5.3.2.

Generally, priority type systems place the priority annotations on session type connectives, rather than actions, since it makes it easier to integrate priorities into type checking. (This is what the type systems for PCP and PGV, presented in § 5.2, do as well.) Explicitly tracking both the priority lower and upper bounds can be quite syntactically burdensome, but there are several tricks to ease this:

- In general, the lower bound can be under-approximated by taking the smallest priority of all endpoints in the typing environment, since a program cannot start communicating earlier than its endpoints permit.
- In systems such as the π-calculus, we do not need to track the upper bound. When composing two programs in sequence, we must check

that the first program finishes communicating before the second program starts. However, the only kind of sequential composition in the π-calculus is prefixing a process with an action. In this case, we must check that the action finishes before the process starts. Hence, we only need to check that the priority of the action is smaller than the lower bound of the process.
- Finally, if we must track both the lower and the upper bound, we can slightly ease the burden of syntax by tracking pairs of lower and upper bounds instead. Since pairs of priority bounds form a lattice, this lets us track both bounds while only manipulating a single value.

One of the benefits of GV is that it closely resembles the manner in which concurrency is exposed to the user in programming languages, which makes it easy to implement the deadlock-free session-typed communication offered by GV as a library, especially in a host language that supports linear types. Less so for Priority GV, as presented in § 5.2, since priority typing requires more of the type system. Fortunately, session-typed communication can be reflected into a monad, in the sense of the *monadic reflection* of an effect, following Filinski [1994], and priority typing can be reflected into a parametrised monad, parametrised by pairs of lower and upper priority bounds, following Atkey [2009]. We explore the monadic reflection of priorities in § 6.2.

The version of PCP introduced in this chapter is different from the version introduced by Dardha and Gay [2018].

- We drop the commuting conversions. Hence, it bears the same relation to Dardha and Gay's PCP as the version of CP in Chapter 2 does to Wadler's CP. Notably, Dardha and Gay's PCP is non-confluent for the same reason Wadler's CP is non-confluent (see § 2.3).
- We simplify variant select and offer to binary select and offer, and reintroduce a separate construct for the absurd offer.
- We allow arbitrary process continuations in the close construct, rather than forcing the process to terminate.
- We restrict ourselves to studying the multiplicative additive core, and omit the exponentials, as is done in most of this thesis. (Neither publication includes the second order quantifiers.)
- We add the additive units, which are omitted in Dardha and Gay [2018], and use the 'inert' semantics for the absurd offer, as discussed in § 2.1.6. (Since the priority constraints encode the blocking behaviour of actions, using the exceptional semantics, as discussed in § 3.3.1, requires a different typing rule for the absurd offer.)

While Dardha and Gay [2018] introduce PCP as an *extension* of CP, it does not formally extend CP, as there are CP processes that are not typeable in PCP. We discuss the relation between CP and PCP in detail in § 5.3.1.

The bulk of the chapter consists of the paper *Prioritise the Best Variation* by Kokke and Dardha [2023], hereafter referred to as Paper II. References made from the main body of this thesis into Paper II will be prefixed by an "II", e.g. "Theorem II.3.13". This chapter proceeds as follows:

- In § 5.1, we provide a legend and an errata for Paper II.
- In § 5.2, we present Priority GV and Priority CP, their metatheories, and the correspondence between Priority GV and Priority CP. This section consists entirely of Paper II, and proceeds as follows:
  - In § II.2, we introduce PGV.
  - In § II.3, we present the metatheory for PGV.

    Notably, we prove *preservation* (referred to as *subject reduction*, § II.3.1, Theorem II.3.5) and *global progress* (§ II.3.2, Theorem II.3.14).
  - In § II.4, we revisit PCP and its metatheory, and prove a sound and complete operational correspondence between PCP and PGV.

    The version of PCP presented in this section differs from Dardha and Gay's PCP, as discussed previously, as it drops commuting conversions, allows arbitrary process continuations after a close action, and does not include the exponentials. Notably, we prove *progress* (§ II.4.5, Theorem II.4.4), define a translation from PCP into PGV (§ II.4.6), and prove that the translation preserves types (§ II.4.6, Theorem II.4.6), and gives rise to a complete (§ II.4.6, Theorem II.4.7) and sound (§ II.4.6, Theorem II.4.10) operational correspondence.
  - In § II.5, we discuss an example program.
  - In § II.6, we discuss related work.
- In § 5.3, we discuss PCP in further detail:
  - In § 5.3.1, we discuss the relation between PCP and CP.
  - In § 5.3.2, we introduce priority inference for PCP.

5.1 Legend and Errata

The conventions and terminology in Paper II are different from those used in the rest of this thesis.

- The terms, types, and priorities of *both* Priority CP and Priority GV are printed in red, blue, and green, respectively, and are rendered in an italicised or bolded font with serif.

- The names for the rules of structural congruence, reduction, and the type system differ slightly from those used for HGV in Chapter 4 and for HCP in Chapter 3.
- The following changes relate to the presentation of PCP:
  - Duality is written as $A^{\perp}$ instead of $\overline{A}$.
  - The absurd offer is written as $x \triangleright \{\}$ instead of $x \not\downarrow N$.
  - Reduction is not definitionally closed under evaluation contexts, as by E-Cong. Instead, there are separate congruence rules for name restriction and parallel composition (E-LiftRes and E-LiftPar).
  - A *ready* process is called an "action" and a process *ready to act on* some endpoint is said to "act on" that endpoint (§ II.4.5, Definition II.4.1).
  - There is no proof of Progress analogous to Proposition 3.35. Instead, the paper proves Closed Progress (§ II.4.5, Theorem II.4.4).
  - There is no definition of *canonical form* analogous to Definition 3.33, which includes both normal and neutral forms. (The latter is not needed for Closed Progress. Hence, 0 suffices.)
    Unfortunately, the paper uses the name "canonical form" to refer to a form that serves the same purpose as the *maximum configuration context* or *right-branching form*, i.e. the form
    $$(\nu x_1 \bar{x}_1) \cdots (\nu x_n \bar{x}_n)(P_1 \parallel \cdots \parallel P_m)$$
    where each process P_i (for $1 \leq i \leq m$) is ready (§ II.4.5, Definition II.4.2).
- The following changes relate to the presentation of PGV:
  - Configuration typing does not use configuration types, as in Chapter 4, but omits the type of the value returned by the configuration, as, e.g. in Lindley and Morris [2015] (see Figure II.2).

There are minor errors in Paper II:

- The text states that "Priority GV is more flexible than GV" and "more expressive than GV" (§ II.2), which is not true, as we discuss in § 5.3.1.
- The text states that "PLL [Priority Linear Logic] is an extension of CLL" (§ II.4.4), which—when taken as a statement about proofs—is not true, as we discuss in § 5.3.1. The next sentence, which states that "PCP is not in correspondence with CLL" implies the previous statement concerns provability, rather than proofs, and while this is not known to be false, it remains unproven.
- The defintion of *normal form* (§ II.3.2, Definition I.3.11) reads "a configuration $\mathscr{C}$ is in normal form if it is of the form
  $$(\nu x_1 x_1') \ldots (\nu x_n x_n')(\circ M_1 \parallel \ldots \parallel \circ M_m \parallel \bullet V)$$

where each M_i is ready to act on x_i." Instead, the final part of the phrase should read "where each M_i (for $1 \leq i \leq m$) is ready to act on some x_j (for $1 \leq j \leq n$) or some free channel endpoint."

- *Closed Progress* (Theorem II.4.4) is stated using equality rather than structural congruence. The correct statement is:

  If $P \vdash \varnothing$, then either $P \equiv Q$ or there exists a Q such that $P \Longrightarrow Q$

- In the proof for Lemma II.4.8, the cases for $x \triangleleft \mathsf{inl}.\, P$ and $x \triangleleft \mathsf{inr}.\, P$ erroneously list the premise and conclusion of the reduction as $(\!|x[y].\, P|\!)_M$ and $(\!|x[y].\, P|\!)_C$, rather than $(\!|x \triangleleft \mathsf{inl}.\, P|\!)_M$ and $(\!|x \triangleleft \mathsf{inl}.\, P|\!)_C$, and $(\!|x \triangleleft \mathsf{inl}.\, P|\!)_M$ and $(\!|x \triangleleft \mathsf{inr}.\, P|\!)_C$, respectively. The intermediate steps are correct.

5.2 Paper II: Prioritise the Best Variation

This section contains the paper with the same title, written in collaboration with Ornela Dardha, which was originally published in the journal Logical Methods in Computer Science, Volume 19, Issue 4, 2023, which is an extended journal version of the paper with the same title and authors originally published in the proceedings for the 41st International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2021) as part of the Lecture Notes in Computer Science (LNCS) series.

The work presented in the paper was conceived of by all the authors. I co-developed Priority GV and was primarily responsible for the initial draft of its theory and metatheory. I adapted the changes to the reduction semantics of Priority CP from my previous work on CP.

Logical Methods in Computer Science
Volume 19, Issue 4, 2023, pp. 28:1–28:47
https://lmcs.episciences.org/

Submitted Dec. 18, 2021
Published Dec. 18, 2023

PRIORITISE THE BEST VARIATION

WEN KOKKE AND ORNELA DARDHA

[a] University of Strathclyde, Glasgow, UK
e-mail address: wen.kokke@strath.ac.uk

[b] School of Computing Science, University of Glasgow, UK
e-mail address: ornela.dardha@glasgow.ac.uk

Abstract. Binary session types guarantee communication safety and session fidelity, but *alone* they cannot rule out deadlocks arising from the interleaving of different sessions.

In Classical Processes (CP) [Wad14]—a process calculus based on classical linear logic—deadlock freedom is guaranteed by combining channel creation and parallel composition under the same logical cut rule. Similarly, in Good Variation (GV) [Wad15, LM15]—a linear concurrent λ-calculus—deadlock freedom is guaranteed by combining channel creation and thread spawning under the same operation, called fork.

In both CP and GV, deadlock freedom is achieved at the expense of expressivity, as the only processes allowed are tree-structured. This is true more broadly than for CP and GV, and it holds for all works in the research line based on Curry-Howard correspondences between linear logic and session types, starting with Caires and Pfenning [CP10]. To overcome the limitations of tree-structured processes, Dardha and Gay [DG18a] define Priority CP (PCP), which allows cyclic-structured processes and restores deadlock freedom by using *priorities*, in line with Kobayashi and Padovani [Kob06, Pad14].

Following PCP, we present Priority GV (PGV), a variant of GV which decouples channel creation from thread spawning. Consequently, we type cyclic-structured processes and restore deadlock freedom by using priorities. We show that our type system is sound by proving subject reduction and progress. We define an encoding from PCP to PGV and prove that the encoding preserves typing and is sound and complete with respect to the operational semantics.

1. Introduction

Session types [Hon93, THK94, HVK98] are a type formalism that ensures *communication channels* are used according to their protocols, much like, *e.g.*, data types ensure that functions are used according to their signature. Session types have been studied in many settings. Most notably, they have been defined for the π-calculus [Hon93, THK94, HVK98], a foundational calculus for communication and concurrency, and the concurrent λ-calculi [GV10], including the main focus of our paper: Good Variation [Wad15, LM15, GV].

Key words and phrases: session types, π-calculus, functional programming, deadlock freedom, GV, CP.

Supported by the EU HORIZON 2020 MSCA RISE project 778233 "Behavioural Application Program Interfaces" (BehAPI).

DOI:10.46298/LMCS-19(4:28)2023

GV is a concurrent λ-calculus with *binary* session types, where each channel is shared between exactly two processes. Binary session types guarantee two crucial properties: *communication safety*—*e.g.*, if the protocol says to transmit an integer, you transmit an integer—and *session fidelity*—*e.g.*, if the protocol says send, you send. A third crucial property is *deadlock freedom*, which ensures that processes do not have cyclic dependencies—*e.g.*, when two processes wait for each other to send a value. Binary session types alone are insufficient to rule out deadlocks arising from interleaved sessions, but several additional techniques have been developed to guarantee deadlock freedom in session-typed π-calculus and concurrent λ-calculus.

In the π-calculus literature, there have been several developments of Curry-Howard correspondences between session-typed π-calculus and linear logic [Gir87]: Caires and Pfenning's πDILL [CP10] corresponds to dual intuitionistic linear logic [Bar96], and Wadler's Classical Processes [Wad14, CP] corresponds to classical linear logic [Gir87, CLL]. Both calculi guarantee deadlock freedom, which they achieve by the combination of binary session types with a restriction on the way processes can be connected by channels: two processes can share at most one channel, and, more generally, information transmitted between any two processes must pass through one unique series of channels and intermediate processes. We refer to such processes as *tree-structured*, because the *communication graph*—where the vertices are ready processes and two vertices are connected by an edge if and only if the corresponding processes share a channel—is a tree. For πDILL, CP, and GV, tree-structure follows from a syntactic restriction: the combination of name restriction and parallel composition into a single syntactic construct, corresponding to the logical cut.

There are many downsides to combining name restriction and parallel composition, such as lack of modularity, difficulty typing structural congruence and formulating label-transition semantics. GV, specifically, struggles with a complicated metatheory due to the mismatch between its term language—where restriction and parallel composition are combined—and its configuration language—where they are not. There have been various approaches to decoupling restriction and parallel composition. Hypersequent CP [MP18, KMP19a, KMP19b, HCP], Hypersequent GV [FKD$^+$21], and Linear Compositional Choreographies [CMS18] decouple them, but maintain the tree-structure of processes and a correspondence to linear logic, *e.g.*, while the typing rules for HCP are no longer exactly the proof rules for CLL, every typing derivations in HCP is isomorphic a proof in CLL, and vice versa. Priority CP [DG18b, PCP] weakens the correspondence to CLL. PCP is a non-conservative extension of CLL: every proof in CLL can be translated to a typing derivation in PCP [DP22], but PCP can prove strictly more theorems, including (partial) bijections between several CLL connectives. In exchange, PCP has a much more expressive language which allows cyclic-structured processes. PCP decouples CP's cut rule into two separate constructs: one for parallel composition via a mix rule, and one for name restriction via a cycle rule. To restore deadlock freedom, PCP uses *priorities* [Kob06, Pad14]. Priorities encode the *order of actions* and rule out bad cyclic interleavings. Dardha and Gay [DG18b] prove cycle-elimination for PCP, adapting the cut-elimination proof for classical linear logic, and deadlock freedom follows as a corollary.

CP and GV are related via a pair of translations which satisfy simulation [LM16], and which can be tweaked to satisfy operational correspondence. The two calculi share the same strong guarantees. GV achieves deadlock freedom via a similar syntactic restriction: it combines channel creation and thread spawning into a single operation, called "fork", which

is related to the cut construct in CP. Unfortunately, as with CP, this syntactic restriction has its downsides.

Our aim is to develop a more expressive version of GV while maintaining deadlock freedom. While process calculi have their advantages, *e.g.*, their succinctness compared to concurrent λ-calculi, we chose to work with GV for several reasons. In general, concurrent λ-calculi support higher-order functions, and have a capability for abstraction not usually present in process calculi. Within a concurrent λ-calculus, one can derive extensions of the communication capabilities of the language via well-understood extensions of the functional fragment, *e.g.*, we can derive internal/external choice from sum types. Concurrent λ-calculi, among other languages, maintain a clear separation between the program which the user writes and the configurations which represent the state of the system as it evaluates the program. However, our main motivation is that results obtained for λ-calculi transfer more easily to real-world functional programming languages. Case in point: we easily adapted the type system of PGV to Linear Haskell [BBN+18], which gives us a library for deadlock-free session-typed programming [KD21a]. The benefit of working specifically with GV, as opposed to other concurrent λ-calculi, is its relation to CP [Wad14], and its formal properties, including deadlock freedom.

We thus pose our research question for GV:

RQ: *Can we design a more expressive GV which guarantees deadlock freedom for cyclic-structured processes?*

We follow the line of work from CP to Priority CP, and present Priority GV (PGV), a variant of GV which decouples channel creation from thread spawning, thus allowing cyclic-structured processes, but which nonetheless guarantees deadlock freedom via priorities. This closes the circle of the connection between CP and GV [Wad14], and their priority-based versions, PCP [DG18b] and PGV. We cannot straightforwardly adapt the priority typing from PCP to PGV, as PGV adds higher-order functions. Instead, the priority typing for PGV follow the work by Padovani and Novara [PN15].

We make the following main contributions:

(1) **Priority GV**. We present Priority GV (§. 2, PGV), a session-typed functional language with priorities, and prove *subject reduction* (Theorem 3.5) and *progress* (Theorem 3.13).
   We addresses several problems in the original GV language, most notably:
   (a) PGV does not require the pseudo-type $S^\sharp$;
   (b) Structural congruence is type preserving.
   PGV answers our research question positively as it allows cyclic-structured binary session-typed processes that are deadlock free.

(2) **Translation from PCP to PGV**. We present a *sound and complete encoding* of PCP [DG18b] in PGV (§. 4). We prove the encoding preserves typing (Theorem 4.6) and satisfies operational correspondence (Theorem 4.7 and Theorem 4.10).
   To obtain a tight correspondence, we update PCP, moving away from commuting conversions and reduction as cut elimination towards reduction based on structural congruence, as it is standard in process calculi.

This paper is an improved and extended version of a paper published at FORTE 2021 international conference [KD21b]. We present detailed examples and complete proofs of our technical results.

Example 2.1.

$$\begin{array}{l}\mathbf{let}\ (x, x') = \mathbf{new}\ ()\ \mathbf{in}\\ \mathbf{let}\ (y, y') = \mathbf{new}\ ()\ \mathbf{in}\\ \mathbf{spawn}\left(\lambda().\ \begin{array}{l}\mathbf{let}\ ((), x') = \mathbf{recv}\ x'\ \mathbf{in}\\ \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\\ \mathbf{wait}\ x'; \mathbf{close}\ y\end{array}\right);\\ \mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}\\ \underline{\mathbf{let}\ ((), y') = \mathbf{recv}\ y'\ \mathbf{in}}\\ \mathbf{close}\ x; \mathbf{wait}\ y'\end{array}$$

Example 2.2.

$$\begin{array}{l}\mathbf{let}\ (x, x') = \mathbf{new}\ ()\ \mathbf{in}\\ \mathbf{let}\ (y, y') = \mathbf{new}\ ()\ \mathbf{in}\\ \mathbf{spawn}\left(\lambda().\ \begin{array}{l}\mathbf{let}\ ((), x') = \mathbf{recv}\ x'\ \mathbf{in}\\ \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\\ \mathbf{wait}\ x'; \mathbf{close}\ y\end{array}\right);\\ \mathbf{let}\ ((), y') = \mathbf{recv}\ y'\ \mathbf{in}\\ \underline{\mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}}\\ \mathbf{close}\ x; \mathbf{wait}\ y'\end{array}$$

2. Priority GV

We present Priority GV (PGV), a session-typed functional language based on GV [Wad15, LM15] which uses priorities à la Kobayashi and Padovani [Kob06, PN15] to enforce deadlock freedom. Priority GV is more flexible than GV because it allows processes to share more than one communication channel.

We illustrate this with two programs in PGV, Example 2.1 and Example 2.2. Each program contains two processes—the main process, and the child process created by **spawn**—which communicate using *two* channels. The child process receives a unit over the channel x/x', and then sends a unit over the channel y/y'. The main process does one of two things:

(a) in Example 2.1, it sends a unit over the channel x/x', and then waits to receive a unit over the channel y/y';
(b) in Example 2.2, it does these in the opposite order, which results in a deadlock.

PGV is more expressive than GV: Example 2.1 is typeable and guaranteed to be deadlock-free in PGV, but is not typeable in GV [Wad14] and not guaranteed deadlock-free in GV's predecessor [GV10]. We believe PGV is a non-conservative *extension* of GV, as CP can be embedded in a Kobayashi-style system [DP18].

2.1. Syntax of Types and Terms.

Session types. Session types (S) are defined by the following grammar:

$$S \quad ::= \quad !^{o}T.S \ \mid\ ?^{o}T.S \ \mid\ \mathbf{end}_{!}^{o} \ \mid\ \mathbf{end}_{?}^{o}$$

Session types $!^{o}T.S$ and $?^{o}T.S$ describe the endpoints of a channel over which we send or receive a value of type T, and then proceed as S. Types $\mathbf{end}_{!}^{o}$ and $\mathbf{end}_{?}^{o}$ describe endpoints of a channel whose communication has finished, and over which we must synchronise before closing the channel. Each connective in a session type is annotated with a *priority* $o \in \mathbb{N}$.

Types. Types (T, U) are defined by the following grammar:

$$T, U \quad ::= \quad T \times U \ \mid\ \mathbf{1} \ \mid\ T + U \ \mid\ \mathbf{0} \ \mid\ T \multimap^{p,q} U \ \mid\ S$$

Types $T \times U$, $\mathbf{1}$, $T + U$, and $\mathbf{0}$ are the standard linear λ-calculus product type, unit type, sum type, and empty type. Type $T \multimap^{p,q} U$ is the standard linear function type, annotated with *priority bounds* $p, q \in \mathbb{N} \cup \{\bot, \top\}$. Every session type is also a type. Given a function with type $T \multimap^{p,q} U$, p is a *lower bound* on the priorities of the endpoints captured by the body of the function, and q is an *upper bound* on the priority of the communications

that take place as a result of applying the function. The type of pure functions $T \multimap U$, *i.e.*, those which perform no communications, is syntactic sugar for $T \multimap^{\top,\bot} U$. The lower bound for a pure function is $\top$ as pure functions never start communicating. For similar reasons, the upper bound for a pure function is $\bot$.

We postulate that the only function types—and, consequently, sequents—that are inhabited in PGV are pure functions and functions $T \multimap^{p,q} U$ for which $p < q$.

Typing Environments. Typing environments Γ, Δ, Θ associate types to names. Environments are linear, so two environments can only be combined as Γ, Δ if their names are distinct, *i.e.*, $\mathrm{fv}(\Gamma) \cap \mathrm{fv}(\Delta) = \varnothing$.

$$\Gamma, \Delta \quad ::= \quad \varnothing \mid \Gamma, x : T$$

Type Duality. Duality plays a crucial role in session types. The two endpoints of a channel are assigned dual types, ensuring that, for instance, whenever one program sends a value on a channel, the program on the other end is waiting to receive. Each session type S has a dual, written $\overline{S}$. Duality is an involutive function which preserves priorities:

$$\overline{!^o T.S} = ?^o T.\overline{S} \qquad \overline{?^o T.S} = !^o T.\overline{S} \qquad \overline{\mathbf{end}^o_!} = \mathbf{end}^o_? \qquad \overline{\mathbf{end}^o_?} = \mathbf{end}^o_!$$

Priorities. Function $\mathrm{pr}(\cdot)$ returns the smallest priority of a session type. The type system guarantees that the top-most connective always holds the smallest priority, so we simply return the priority of the top-most connective:

$$\mathrm{pr}(!^o T.S) = o \qquad \mathrm{pr}(?^o T.S) = o \qquad \mathrm{pr}(\mathbf{end}^o_!) = o \qquad \mathrm{pr}(\mathbf{end}^o_?) = o$$

We extend the function $\mathrm{pr}(\cdot)$ to types and typing contexts by returning the smallest priority in the type or context, or $\top$ if there is no priority. We use $\sqcap$ and $\sqcup$ to denote the minimum and maximum, respectively:

$$\begin{array}{lcllcl} \mathrm{pr}(T \times U) & = & \mathrm{pr}(T) \sqcap \mathrm{pr}(U) & \mathrm{pr}(\mathbf{1}) & = & \top \\ \mathrm{pr}(T + U) & = & \mathrm{pr}(T) \sqcap \mathrm{pr}(U) & \mathrm{pr}(\mathbf{0}) & = & \top \\ \mathrm{pr}(T \multimap^{p,q} U) & = & p & \mathrm{pr}(S) & = & \mathrm{pr}(S) \\ \mathrm{pr}(\Gamma, x : A) & = & \mathrm{pr}(\Gamma) \sqcap \mathrm{pr}(A) & \mathrm{pr}(\varnothing) & = & \top \end{array}$$

Terms. Terms (L, M, N) are defined by the following grammar:

$$\begin{array}{lcl} L, M, N & ::= & x \mid K \mid \lambda x.M \mid M\ N \\ & \mid & () \mid M; N \mid (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \\ & \mid & \mathbf{inl}\ M \mid \mathbf{inr}\ M \mid \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \mid \mathbf{absurd}\ M \\ K & ::= & \mathbf{link} \mid \mathbf{new} \mid \mathbf{spawn} \mid \mathbf{send} \mid \mathbf{recv} \mid \mathbf{close} \mid \mathbf{wait} \end{array}$$

Let x, y, z, and w range over variable names. Occasionally, we use a, b, c, and d. The term language is the standard linear λ-calculus with products, sums, and their units, extended with constants K for the communication primitives.

Constants are best understood in conjunction with their typing and reduction rules in Figs. 1 and 2.

Briefly, **link** links two endpoints together, forwarding messages from one to the other, **new** creates a new channel and returns a pair of its endpoints, and **spawn** spawns off its argument as a new thread.

The **send** and **recv** functions send and receive values on a channel. However, since the typing rules for PGV ensure the linear usage of endpoints, they also return a new copy of the endpoint to continue the session.

The **close** and **wait** functions close a channel.

We use syntactic sugar to make terms more readable: we write $\mathbf{let}\ x = M\ \mathbf{in}\ N$ in place of $(\lambda x.N)\ M$, $\lambda().M$ in place of $\lambda z.z; M$, and $\lambda(x, y).M$ in place of $\lambda z.\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M$. We can recover GV's **fork** as $\lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{spawn}\ (\lambda().x\ y); z$.

Internal and External Choice. Typically, session-typed languages feature constructs for internal and external choice. In GV, these can be defined in terms of the core language, by sending or receiving a value of a sum type [LM15]. We use the following syntactic sugar for internal ($S \oplus^o S'$) and external ($S \,\&^o\, S'$) choice and their units:

$$\begin{array}{lcllcl} S \oplus^o S' & \triangleq & !^o(\overline{S} + \overline{S'}).\mathbf{end}_!^{o+1} & \qquad \oplus^o\{\} & \triangleq & !^o\mathbf{0}.\mathbf{end}_!^{o+1} \\ S \,\&^o\, S' & \triangleq & ?^o(S + S').\mathbf{end}_?^{o+1} & \qquad \&^o\{\} & \triangleq & ?^o\mathbf{0}.\mathbf{end}_?^{o+1} \end{array}$$

As the syntax for units suggests, these are the binary and nullary forms of the more common n-ary choice constructs $\oplus^o\{l_i : S_i\}_{i \in I}$ and $\&^o\{l_i : S_i\}_{i \in I}$, which one may obtain generalising the sum types to variant types. For simplicity, we present only the binary and nullary forms.

Similarly, we use syntactic sugar for the term forms of choice, which combine sending and receiving with the introduction and elimination forms for the sum and empty types. There are two constructs for binary internal choice, expressed using the meta-variable ℓ which ranges over $\{\mathbf{inl}, \mathbf{inr}\}$. As there is no introduction for the empty type, there is no construct for nullary internal choice:

$$\begin{array}{l} \mathbf{select}\ \ell \quad \triangleq \quad \lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\ell\ y, x)); z \\ \mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \triangleq \\ \quad \mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \\ \mathbf{offer}\ L\ \{\} \quad \triangleq \quad \mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{absurd}\ z \end{array}$$

2.2. Operational Semantics.

Configurations. Priority GV terms are evaluated as part of a configuration of processes. Configurations are defined by the following grammar:

$$\phi ::= \bullet \mid \circ \qquad \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \phi\ M \mid \mathcal{C} \parallel \mathcal{D} \mid (\nu x x')\mathcal{C}$$

Configurations ($\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$) consist of threads $\phi\ M$, parallel compositions $\mathcal{C} \parallel \mathcal{D}$, and name restrictions $(\nu x x')\mathcal{C}$. To preserve the functional nature of PGV, where programs return a single value, we use flags (ϕ) to differentiate between the main thread, marked $\bullet$, and child threads created by **spawn**, marked $\circ$. Only the main thread returns a value. We determine the flag of a configuration by combining the flags of all threads in that configuration:

$$\bullet + \circ = \bullet \qquad \circ + \bullet = \bullet \qquad \circ + \circ = \circ \qquad (\bullet + \bullet \text{ is undefined})$$

To distinguish between the use of $\circ$ to mark child threads [LM15] and the use of the meta-variable o for priorities [DG18b], they are typeset in a different font and colour.

Values. Values (V, W), evaluation contexts (E), thread evaluation contexts ($\mathcal{F}$), and configuration contexts ($\mathcal{G}$) are defined by the following grammars:

$$\begin{array}{lcl} V, W & ::= & x \mid K \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\ V \mid \mathbf{inr}\ V \\ E & ::= & \Box \mid E\ M \mid V\ E \\ & \mid & E; N \mid (E, M) \mid (V, E) \mid \mathbf{let}\ (x, y) = E\ \mathbf{in}\ M \\ & \mid & \mathbf{inl}\ E \mid \mathbf{inr}\ E \mid \mathbf{case}\ E\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \mid \mathbf{absurd}\ E \\ \mathcal{F} & ::= & \phi\ E \\ \mathcal{G} & ::= & \Box \mid \mathcal{G} \parallel \mathcal{C} \mid (\nu xy)\mathcal{G} \end{array}$$

Values are the subset of terms which cannot reduce further. Evaluation contexts are one-hole term contexts, *i.e.*, terms with exactly one hole, written $\Box$. We write $E[M]$ for the evaluation context E with its hole replaced by the term M. Evaluation contexts are specifically those one-hole term contexts under which term reduction can take place. Thread contexts are a convenient way to lift the notion of evaluation contexts to threads. We write $\mathcal{F}[M]$ for the thread context $\mathcal{F}$ with its hole replaced by the term M. Configuration contexts are one-hole configuration contexts, *i.e.*, configurations with exactly one hole, written $\Box$. Specifically, configuration contexts are those one-hole term contexts under which configuration reduction can take place. The definition for $\mathcal{G}$ only gives the case in which the hole is in the left-most parallel process, *i.e.*, it only defines $\mathcal{G} \parallel \mathcal{C}$ and not $\mathcal{C} \parallel \mathcal{G}$. The latter is not needed, as $\parallel$ is symmetric under structural congruence, though it would be harmless to add. We write $\mathcal{G}[\mathcal{C}]$ for the evaluation context $\mathcal{G}$ with its hole replaced by the term $\mathcal{C}$.

Reduction Relation. We factor the reduction relation of PGV into a deterministic reduction on terms ($\longrightarrow_M$) and a non-deterministic reduction on configurations ($\longrightarrow_\mathcal{C}$), see Fig. 1. We write $\longrightarrow_M^+$ and $\longrightarrow_\mathcal{C}^+$ for the transitive closures, and $\longrightarrow_M^\star$ and $\longrightarrow_\mathcal{C}^\star$ for the reflexive-transitive closures.

Term reduction is the standard call-by-value, left-to-right evaluation for GV, and only deviates from reduction for the linear λ-calculus in that it reduces terms to values or ready terms waiting to perform a communication action.

Configuration reduction resembles evaluation for a process calculus: E-Link, E-Send, and E-Close perform communications, E-LiftC allows reduction under configuration contexts, and E-LiftSC embeds a structural congruence $\equiv$. The remaining rules mediate between the process calculus and the functional language: E-New and E-Spawn evaluate the **new** and **spawn** constructs, creating the equivalent configuration constructs, and E-LiftM embeds term reduction.

Structural congruence satisfies the following axioms: SC-LinkSwap allows swapping channels in the link process. SC-ResLink allows restriction to be applied to link which is structurally equivalent to the terminated process, thus allowing elimination of unnecessary restrictions. SC-ResSwap allows swapping channels and SC-ResComm states that restriction is commutative. SC-ResExt is the standard scope extrusion rule. Rules SC-ParNil, SC-ParComm and SC-ParAssoc state that parallel composition uses the terminated process as the neutral element; it is commutative and associative.

While our configuration reduction is based on the standard evaluation for GV, the increased expressiveness of PGV allows us to simplify the relation on two counts.

(i) *We decompose the* **fork** *construct.* In GV, **fork** creates a new channel, spawns a child thread, and, when the child thread finishes, it closes the channel to its parent. In PGV, these are three separate operations: **new**, **spawn**, and **close**. We no longer require

Term reduction.

$$
\begin{array}{lllll}
\text{E-Lam} & (\lambda x.M)\ V & & \longrightarrow_M & M\{V/x\} \\
\text{E-Unit} & (); M & & \longrightarrow_M & M \\
\text{E-Pair} & \mathbf{let}\ (x,y) = (V,W)\ \mathbf{in}\ M & & \longrightarrow_M & M\{V/x\}\{W/y\} \\
\text{E-Inl} & \mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} & & \longrightarrow_M & M\{V/x\} \\
\text{E-Inr} & \mathbf{case}\ \mathbf{inr}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} & & \longrightarrow_M & N\{V/y\}
\end{array}
$$

$$
\text{E-Lift}\ \frac{M \longrightarrow_M M'}{E[M] \longrightarrow_M E[M']}
$$

Structural congruence.

$$
\begin{array}{lllll}
\text{SC-LinkSwap} & \mathcal{F}[\mathbf{link}\ (x,y)] & \equiv & \mathcal{F}[\mathbf{link}\ (y,x)] \\
\text{SC-ResLink} & (\nu xy)(\phi\ \mathbf{link}\ (x,y)) & \equiv & \phi\ () \\
\text{SC-ResSwap} & (\nu xy)\mathcal{C} & \equiv & (\nu yx)\mathcal{C} \\
\text{SC-ResComm} & (\nu xy)(\nu zw)\mathcal{C} & \equiv & (\nu zw)(\nu xy)\mathcal{C},\ \text{if } \{x,y\} \cap \{z,w\} = \varnothing \\
\text{SC-ResExt} & (\nu xy)(\mathcal{C} \parallel \mathcal{D}) & \equiv & \mathcal{C} \parallel (\nu xy)\mathcal{D},\ \text{if } x,y \notin \mathrm{fv}(\mathcal{C}) \\
\text{SC-ParNil} & \mathcal{C} \parallel \circ() & \equiv & \mathcal{C} \\
\text{SC-ParComm} & \mathcal{C} \parallel \mathcal{D} & \equiv & \mathcal{D} \parallel \mathcal{C} \\
\text{SC-ParAssoc} & \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) & \equiv & (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}
\end{array}
$$

Configuration reduction.

$$
\begin{array}{lllll}
\text{E-Link} & (\nu xy)(\mathcal{F}[\mathbf{link}\ (w,x)] \parallel \mathcal{C}) & \longrightarrow_{\mathcal{C}} & \mathcal{F}[()] \parallel \mathcal{C}\{w/y\} \\
\text{E-New} & \mathcal{F}[\mathbf{new}\ ()] & \longrightarrow_{\mathcal{C}} & (\nu xy)(\mathcal{F}[(x,y)]),\ \text{if } x,y \notin \mathrm{fv}(\mathcal{F}) \\
\text{E-Spawn} & \mathcal{F}[(\mathbf{spawn}\ V)] & \longrightarrow_{\mathcal{C}} & \mathcal{F}[()] \parallel \circ\ V\ () \\
\text{E-Send} & (\nu xy)(\mathcal{F}[\mathbf{send}\ (V,x)] \parallel \mathcal{F}'[\mathbf{recv}\ y]) & \longrightarrow_{\mathcal{C}} & (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V,y)]) \\
\text{E-Close} & (\nu xy)(\mathcal{F}[\mathbf{wait}\ x] \parallel \mathcal{F}'[\mathbf{close}\ y]) & \longrightarrow_{\mathcal{C}} & \mathcal{F}[()] \parallel \mathcal{F}'[()]
\end{array}
$$

$$
\text{E-LiftC}\ \frac{\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{C}'}{\mathcal{G}[\mathcal{C}] \longrightarrow_{\mathcal{C}} \mathcal{G}[\mathcal{C}']}
\qquad
\text{E-LiftM}\ \frac{M \longrightarrow_M M'}{\mathcal{F}[M] \longrightarrow_M \mathcal{F}[M']}
\qquad
\text{E-LiftSC}\ \frac{\mathcal{C} \equiv \mathcal{C}' \qquad \mathcal{C}' \longrightarrow_{\mathcal{C}} \mathcal{D}' \qquad \mathcal{D}' \equiv \mathcal{D}}{\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{D}}
$$

Figure 1: Operational Semantics for PGV.

that every child thread finishes by returning a terminated channel. Consequently, we also simplify the evaluation of the **link** construct.

Intuitively, evaluating **link** causes a substitution: if we have a channel bound as (νxy), then **link** (w,x) replaces all occurrences of y by w. However, in GV, **link** is required to return a terminated channel, which means that the semantics for *link* must create a fresh channel of type $\mathbf{end}_!/\mathbf{end}_?$. The endpoint of type $\mathbf{end}_!$ is returned by the *link* construct, and a **wait** on the other endpoint guards the actual substitution. In PGV, evaluating **link** simply causes a substitution.

(ii) *Our structural congruence is type preserving*. Consequently, we can embed it directly into the reduction relation. In GV, this is not the case, and subject reduction relies on proving that if the result of rewriting via $\equiv$ followed by reducing via $\longrightarrow_{\mathcal{C}}$ is an ill-typed configuration, we can rewrite it to a well-typed configuration via $\equiv$.

Static Typing Rules.

$$\text{T-Var}\ \frac{}{x : T \vdash^{\bot} x : T} \qquad \text{T-Const}\ \frac{}{\varnothing \vdash^{\bot} K : T} \qquad \text{T-Lam}\ \frac{\Gamma, x : T \vdash^{q} M : U}{\Gamma \vdash^{\bot} \lambda x.M : T \multimap^{\mathrm{pr}(\Gamma),q} U}$$

$$\text{T-App}\ \frac{\Gamma \vdash^{p} M : T \multimap^{p',q'} U \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta) \qquad q < p'}{\Gamma, \Delta \vdash^{p \sqcup q \sqcup q'} M\ N : U}$$

$$\text{T-Unit}\ \frac{}{\varnothing \vdash^{\bot} () : \mathbf{1}} \qquad \text{T-LetUnit}\ \frac{\Gamma \vdash^{p} M : \mathbf{1} \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} M; N : T}$$

$$\text{T-Pair}\ \frac{\Gamma \vdash^{p} M : T \qquad \Delta \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} (M, N) : T \times U}$$

$$\text{T-LetPair}\ \frac{\Gamma \vdash^{p} M : T \times T' \qquad \Delta, x : T, y : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, T, T')}{\Gamma, \Delta \vdash^{p \sqcup q} \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}$$

$$\text{T-Inl}\ \frac{\Gamma \vdash^{p} M : T \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma \vdash^{p} \mathbf{inl}\ M : T + U} \qquad \text{T-Inr}\ \frac{\Gamma \vdash^{p} M : U \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma \vdash^{p} \mathbf{inr}\ M : T + U}$$

$$\text{T-CaseSum}\ \frac{\Gamma \vdash^{p} L : T + T' \qquad \Delta, x : T \vdash^{q} M : U \qquad \Delta, y : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

$$\text{T-Absurd}\ \frac{\Gamma \vdash^{p} M : \mathbf{0}}{\Gamma, \Delta \vdash^{p} \mathbf{absurd}\ M : T}$$

Type Schemas for Constants.

$$\mathbf{link} : S \times \overline{S} \multimap \mathbf{1} \qquad \mathbf{new} : \mathbf{1} \multimap S \times \overline{S} \qquad \mathbf{spawn} : (\mathbf{1} \multimap^{p,q} \mathbf{1}) \multimap \mathbf{1}$$

$$\mathbf{send} : T \times\ !^{o}T.S \multimap^{\top,o} S \qquad \mathbf{recv} : ?^{o}T.S \multimap^{\top,o} T \times S$$

$$\mathbf{close} : \mathbf{end}_{!}^{o} \multimap^{\top,o} \mathbf{1} \qquad \mathbf{wait} : \mathbf{end}_{?}^{o} \multimap^{\top,o} \mathbf{1}$$

Runtime Typing Rules.

$$\text{T-Main}\ \frac{\Gamma \vdash^{p} M : T}{\Gamma \vdash^{\bullet} \bullet\ M} \qquad \text{T-Child}\ \frac{\Gamma \vdash^{p} M : \mathbf{1}}{\Gamma \vdash^{\circ} \circ\ M} \qquad \text{T-Res}\ \frac{\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu xy)\mathcal{C}} \qquad \text{T-Par}\ \frac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \Delta \vdash^{\phi'} \mathcal{D}}{\Gamma, \Delta \vdash^{\phi + \phi'} \mathcal{C} \parallel \mathcal{D}}$$

Figure 2: Typing Rules for PGV.

2.3. **Typing Rules.**

Terms Typing. Typing rules for terms are at the top of Fig. 2. Terms are typed by a judgement $\Gamma \vdash^{p} M : T$ stating that "a term M has type T and an upper bound on its priority p under the typing environment Γ". Typing for the linear λ-calculus is standard. Linearity is ensured by splitting environments on branching rules, requiring that the environment in the variable rule consists of just the variable, and the environment in the constant and unit rules are empty. Constants K are typed using type schemas, which hold for any concrete assignment of types and priorities to their meta-variables. Instantiated type schemas are embedded into typing derivations using T-Const in Fig. 2, *e.g.*, the type schema for **send** can be instantiated with $o = 2$, $T = \mathbf{1}$, and $S = \mathbf{0}$, and embedded using T-Const to give the following typing derivation:

$$\frac{}{\mathbf{send} : \mathbf{1} \times !^{2}\mathbf{1}.\mathbf{0} \multimap^{\top,2} \mathbf{0}}$$

The typing rules treat all variables as linear resources, even those of non-linear types such as $\mathbf{1}$, though they can easily be extended to allow values with unrestricted usage [Wad14].

The only non-standard feature of the typing rules is the priority annotations. Priorities are based on obligations/capabilities used by Kobayashi [Kob06], and simplified to single priorities following Padovani [Pad14]. The integration of priorities into GV is adapted from Padovani and Novara [PN15]. Paraphrasing Dardha and Gay [DG18b], priorities obey the following two laws:

(i) an action with lower priority happens before an action with higher priority; and
(ii) communication requires *equal* priorities for dual actions.

In PGV, we keep track of a lower and upper bound on the priorities of a term, *i.e.*, while evaluating the term, when it starts communicating, and when it finishes, respectively. The upper bound is written on the sequent and the lower bound is approximated from the typing environment, *e.g.*, for $\Gamma \vdash^{p} M : T$ the upper bound is p and the lower bound is at least $\mathrm{pr}(\Gamma)$. The latter is correct because a term cannot communicate at a priority earlier than the earliest priority amongst the channels it has access to. It is an *approximation* on function terms, as these can "skip" communication by returning the corresponding channel unused. However, linearity prevents such functions from being used in well typed configurations: once the unused channel's priority has passed, it can no longer be used.

Typing rules for sequential constructs enforce sequentiality, *e.g.*, the typing for $M; N$ has a side condition which requires that the upper bound of M is smaller than the lower bound of N, *i.e.*, M finishes before N starts. The typing rule for **new** ensures that both endpoints of a channel share the same priorities. Together, these two constraints guarantee deadlock freedom.

To illustrate this, let's go back to the deadlocked program in Example 2.2. Crucially, it composes the terms below in parallel. While each of these terms itself is well typed, they impose opposite conditions on the priorities, so connecting x to x' and y to y' using T-Res is ill-typed, as there is no assignment to o and o' that can satsify both $o < o'$ and $o' < o$.

(We omit the priorities on $\mathbf{end}_!$ and $\mathbf{end}_?$.)

$$\frac{\begin{array}{c} y' : ?^{o'}\mathbf{1}.\mathbf{end}_? \vdash^{o'} \mathbf{recv}\ y' : \mathbf{1} \times \mathbf{end}_? \\ x : !^{o}\mathbf{1}.\mathbf{end}_!, y' : \mathbf{end}_? \vdash^{p} \mathbf{let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}\ \ldots : \mathbf{1} \qquad o' < o \end{array}}{x : !^{o}\mathbf{1}.\mathbf{end}_!, y' : ?^{o'}\mathbf{1}.\mathbf{end}_? \vdash^{p} \mathbf{let}\ ((), y') = \mathbf{recv}\ y'\ \mathbf{in\ let}\ x = \mathbf{send}\ ((), x)\ \mathbf{in}\ \ldots : \mathbf{1}}$$

$$\frac{\begin{array}{c} x' : ?^{o}\mathbf{1}.\mathbf{end}_? \vdash^{o} \mathbf{recv}\ x' : \mathbf{1} \times \mathbf{end}_? \\ y : !^{o'}\mathbf{1}.\mathbf{end}_!, x' : \mathbf{end}_? \vdash^{q} \mathbf{let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\ \ldots : \mathbf{1} \qquad o < o' \end{array}}{y : !^{o'}\mathbf{1}.\mathbf{end}_!, x' : ?^{o}\mathbf{1}.\mathbf{end}_? \vdash^{q} \mathbf{let}\ ((), x') = \mathbf{recv}\ x'\ \mathbf{in\ let}\ y = \mathbf{send}\ ((), y)\ \mathbf{in}\ \ldots : \mathbf{1}}$$

Closures suspend communication, so T-Lam stores the priority bounds of the function body on the function type, and T-App restores them. For instance, $\lambda x.\mathbf{send}\ (x, y)$ is assigned the type $A \multimap^{o,o} S$, *i.e.*, a function which, when applied, starts and finishes communicating at priority o.

$$\frac{\dfrac{}{\mathbf{send} : T \times\, !^{o}T.S \multimap^{\top,o} S} \qquad \dfrac{\dfrac{}{x : T \vdash^{\bot} x : T} \qquad \dfrac{}{x : T, y : !^{o}T.S \vdash^{\bot} y : !^{o}T.S}}{x : T, y : !^{o}T.S \vdash^{\bot} (x, y) : T \times\, !^{o}T.S}}{x : T, y : !^{o}T.S \vdash^{o} \mathbf{send}\ (x, y) : S}}{y : !^{o}T.S \vdash^{\bot} \lambda x.\mathbf{send}\ (x, y) : T \multimap^{o,o} S}$$

For simplicity, we assume priority annotations are not inferred, but provided as an input to type checking. However, for any term, priorities can be inferred, *e.g.*, by using the topological ordering of the directed graph where the vertices are the priority meta-variables and the edges are the inequality constraints between the priority meta-variables in the typing derivation.

Configurations Typing. Typing rules for configurations are at the bottom of Fig. 2. Configurations are typed by a judgement $\Gamma \vdash^{\phi} \mathcal{C}$ stating that "a configuration $\mathcal{C}$ with flag ϕ is well typed under typing environment Γ". Configuration typing is based on the standard typing for GV. Terms are embedded either as main or as child threads. The priority bound from the term typing is discarded, as configurations contain no further blocking actions. Main threads are allowed to return a value, whereas child threads are required to return the unit value. Sequents are annotated with a flag ϕ, which ensures that there is at most one main thread.

While our configuration typing is based on the standard typing for GV, it differs on two counts:

(i) *we require that child threads return the unit value*, as opposed to a terminated channel; and
(ii) *we simplify typing for parallel composition*.

In order to guarantee deadlock freedom, in GV each parallel composition must split *exactly one* channel of the channel pseudo-type $S^{\sharp}$ into two endpoints of type S and $\overline{S}$. Consequently, associativity of parallel composition does not preserve typing. In PGV, we guarantee deadlock freedom using priorities, which removes the need for the channel pseudo-type $S^{\sharp}$, and simplifies typing for parallel composition, while restoring type preservation for the structural congruence.

$$\textsc{T-LamUnit}\ \frac{\Gamma \vdash^{q} M : T}{\Gamma \vdash^{\bot} \lambda().M : \mathbf{1} \multimap^{\mathrm{pr}(\Gamma),q} T} \triangleq \frac{\dfrac{}{z : \mathbf{1} \vdash^{\bot} z : \mathbf{1}} \qquad \Gamma \vdash^{q} M : T}{\Gamma, z : \mathbf{1} \vdash^{q} z; M : T}}{\Gamma \vdash^{\bot} \lambda z.z; M : \mathbf{1} \multimap^{\mathrm{pr}(\Gamma),q} T}$$

$$\textsc{T-LamPair}\ \frac{\Gamma, x : T, y : T' \vdash^{q} M : U}{\Gamma \vdash^{\bot} \lambda(x, y).M : T \times T' \multimap^{\mathrm{pr}(\Gamma),q} U} \triangleq$$

$$\frac{\dfrac{}{z : T \times T' \vdash^{\bot} z : T \times T'} \qquad \Gamma, x : T, y : T' \vdash^{q} M : U}{\Gamma, z : T \times T' \vdash^{q} \mathbf{let}\ (x, y) = z\ \mathbf{in}\ M : T}}{\Gamma \vdash^{\bot} \lambda z.\mathbf{let}\ (x, y) = z\ \mathbf{in}\ M : T \times T' \multimap^{\mathrm{pr}(\Gamma),q} U}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash^{p} M : T \qquad \Delta, x : T \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} \mathbf{let}\ x = M\ \mathbf{in}\ N : U} \triangleq$$

$$\frac{\dfrac{\Delta, x : T \vdash^{q} N : U}{\Delta \vdash^{\bot} \lambda x.N : T \multimap^{\mathrm{pr}(\Delta),q} U} \qquad \Gamma \vdash^{p} M : T \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{q \sqcup p} (\lambda x.N)\ M : U}$$

$$\textsc{T-Fork}\ \frac{}{\varnothing \vdash^{\bot} \mathbf{fork} : (S \multimap^{p,q} \mathbf{1}) \multimap \overline{S}} \triangleq$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\text{(A)}\quad \dfrac{}{\varnothing \vdash^{\bot} () : \mathbf{1}}}{\varnothing \vdash^{\bot} \mathbf{new}\ () : S \times \overline{S}} \qquad \dfrac{}{x : S \multimap^{p,q} \mathbf{1} \vdash^{\bot} x : S \multimap^{p,q} \mathbf{1}} \qquad \dfrac{}{y : S \vdash^{\bot} y : S}}{x : S \multimap^{p,q} \mathbf{1}, y : S \vdash^{q} x\ y : \mathbf{1}}}{\text{(B)}\quad x : S \multimap^{p,q} \mathbf{1}, y : S \vdash^{\bot} \lambda().x\ y : \mathbf{1} \multimap^{p,q} \mathbf{1}}}{x : S \multimap^{p,q} \mathbf{1}, y : S \vdash^{\bot} \mathbf{spawn}\ (\lambda().x\ y) : \mathbf{1}} \qquad \dfrac{}{z : \overline{S} \vdash^{\bot} z : \overline{S}}}{x : S \multimap^{p,q} \mathbf{1}, y : S, z : \overline{S} \vdash^{\bot} \mathbf{spawn}\ (\lambda().x\ y); z : \overline{S}}}{x : S \multimap^{p,q} \mathbf{1} \vdash^{\bot} \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{spawn}\ (\lambda().x\ y); z : \overline{S}}}{\varnothing \vdash^{\bot} \lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{spawn}\ (\lambda().x\ y); z : (S \multimap^{p,q} \mathbf{1}) \multimap \overline{S}}$$

$$\text{(A)} = \mathbf{new} : \mathbf{1} \multimap S \times \overline{S} \qquad \text{(B)} = \mathbf{spawn} : (\mathbf{1} \multimap^{p,q} \mathbf{1}) \multimap \mathbf{1}$$

Figure 3: Typing Rules for Syntactic Sugar for PGV (T-LamUnit, T-LamPair, T-Let, and T-Fork).

$$\textsc{T-Select-Inl}\quad \frac{\mathrm{pr}(S) = \mathrm{pr}(S')}{\varnothing \vdash^{\perp} \mathbf{select\ inl} : S \oplus^{o} S' \multimap^{\top,o} S} \triangleq$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash () : \mathbf{1}}\,(\textsc{a})}{\varnothing \vdash \mathbf{new}\ () : \overline{S} \times S} \quad \dfrac{\dfrac{\dfrac{}{y : \overline{S} \vdash y : \overline{S}}}{y : \overline{S} \vdash \mathbf{inl}\ y : \overline{S} + \overline{S'}} \quad \dfrac{}{x : S \oplus^{o} S' \vdash x : S \oplus^{o} S'}}{x : S \oplus^{o} S', y : \overline{S} \vdash (\mathbf{inl}\ y, x) : (\overline{S} + \overline{S'}) \times (S \oplus^{o} S')}\,(\textsc{c})}{x : S \oplus^{o} S', y : \overline{S} \vdash \mathbf{send}\ (\mathbf{inl}\ y, x) : \mathbf{end}_{!}^{o+1}}\,(\textsc{b})}{x : S \oplus^{o} S', y : \overline{S} \vdash \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)) : \mathbf{1}} \quad \dfrac{}{z : S \vdash z : S}}{x : S \oplus^{o} S', y : \overline{S}, z : S \vdash \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z : S}}{x : S \oplus^{o} S' \vdash \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z : S}}{\varnothing \vdash \lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z : S \oplus^{o} S' \multimap^{\top,o} S}$$

$$(\textsc{a}) = \mathbf{new} : \mathbf{1} \multimap^{\top,o} \overline{S} \times S \qquad (\textsc{b}) = \mathbf{close} : \mathbf{end}_{!}^{o+1} \multimap^{\top,o+1} \mathbf{1}$$

$$(\textsc{c}) = \mathbf{send} : (\overline{S} + \overline{S'}) \times (S \oplus^{o} S') \multimap^{\top,o} \mathbf{end}_{!}^{o+1}$$

$$\textsc{T-Select-Inr}\quad \frac{\mathrm{pr}(S) = \mathrm{pr}(S')}{\varnothing \vdash^{\perp} \mathbf{select\ inr} : S \oplus^{o} S' \multimap^{\top,o} S'} \triangleq$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash () : \mathbf{1}}\,(\textsc{a})}{\varnothing \vdash \mathbf{new}\ () : \overline{S'} \times S'} \quad \dfrac{\dfrac{\dfrac{}{y : \overline{S'} \vdash y : \overline{S'}}}{y : \overline{S'} \vdash \mathbf{inr}\ y : \overline{S} + \overline{S'}} \quad \dfrac{}{x : S \oplus^{o} S' \vdash x : S \oplus^{o} S'}}{x : S \oplus^{o} S', y : \overline{S'} \vdash (\mathbf{inr}\ y, x) : (\overline{S} + \overline{S'}) \times (S \oplus^{o} S')}\,(\textsc{c})}{x : S \oplus^{o} S', y : \overline{S'} \vdash \mathbf{send}\ (\mathbf{inr}\ y, x) : \mathbf{end}_{!}^{o+1}}\,(\textsc{b})}{x : S \oplus^{o} S', y : \overline{S'} \vdash \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)) : \mathbf{1}} \quad \dfrac{}{z : S' \vdash z : S'}}{x : S \oplus^{o} S', y : \overline{S'}, z : S' \vdash \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z : S'}}{x : S \oplus^{o} S' \vdash \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z : S'}}{\varnothing \vdash \lambda x.\mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z : S \oplus^{o} S' \multimap^{\top,o} S'}$$

$$(\textsc{a}) = \mathbf{new} : \mathbf{1} \multimap^{\top,o} \overline{S'} \times S' \qquad (\textsc{b}) = \mathbf{close} : \mathbf{end}_{!}^{o+1} \multimap^{\top,o+1} \mathbf{1}$$

$$(\textsc{c}) = \mathbf{send} : (\overline{S} + \overline{S'}) \times (S \oplus^{o} S') \multimap^{\top,o} \mathbf{end}_{!}^{o+1}$$

Figure 4: Typing Rules for Syntactic Sugar for PGV (T-Select-Inl and T-Select-Inr).

$$\text{T-Offer}\quad \frac{\Gamma \vdash^{p} L : S \,\&^{o}\, S' \quad \Delta, x : S \vdash^{q} M : T \quad \Delta, y : S' \vdash^{q} N : T \quad o \sqcup p < \mathrm{pr}(\Delta, S, S')}{\Gamma, \Delta \vdash^{o \sqcup p \sqcup q} \mathbf{offer}\ L\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : T} \quad \triangleq$$

(C)

$$\frac{\dfrac{\dfrac{}{z : S + S' \vdash^{\perp} z : S + S'} \quad \Delta, x : S \vdash^{q} M : T \quad \Delta, y : S' \vdash^{q} N : T}{\Delta, z : S + S' \vdash^{q} \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : T} \quad o < \mathrm{pr}(\Delta, S + S')}{\Delta, z : S + S', w : \mathbf{end}_{?}^{o+1} \vdash^{o \sqcup q} \mathbf{wait}\ w; \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : T}$$

(B) $\dfrac{}{w : \mathbf{end}_{?}^{o+1} \vdash^{\perp} w : \mathbf{end}_{?}^{o+1}}$

$$\frac{\dfrac{(\text{A}) \quad \Gamma \vdash^{p} L : ?^{o}(S + S').\mathbf{end}_{?}^{o+1}}{\Gamma \vdash^{o \sqcup p} \mathbf{recv}\ L : (S + S') \times \mathbf{end}_{?}^{o+1}} \quad (\text{C}) \quad o \sqcup p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{o \sqcup p \sqcup q} \mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{case}\ z\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : T}$$

(A) $= \mathbf{recv} : ?^{o}(S + S').\mathbf{end}_{?}^{o+1} \multimap^{\top, o} (S + S') \times \mathbf{end}_{?}^{o+1}$ (B) $= \mathbf{wait} : \mathbf{end}_{?}^{o+1} \multimap^{\top, o} \mathbf{1}$

$$\text{T-Offer-Absurd}\quad \frac{\Gamma \vdash^{p} L : \&^{o}\{\} \quad o \sqcup p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{o \sqcup p} \mathbf{offer}\ L\ \{\} : T} \quad \triangleq$$

$$\frac{\dfrac{\dfrac{(\text{A}) \quad \Gamma \vdash^{p} L : ?^{o}\mathbf{0}.\mathbf{end}_{?}^{o+1}}{\Gamma \vdash^{o \sqcup p} \mathbf{recv}\ L : \mathbf{0} \times \mathbf{end}_{?}^{o+1}} \quad (\text{B}) \quad \dfrac{}{w : \mathbf{end}_{?}^{o+1} \vdash^{\perp} w : \mathbf{end}_{?}^{o+1}}}{w : \mathbf{end}_{?}^{o+1} \vdash^{o} \mathbf{wait}\ w : \mathbf{1}} \quad \dfrac{\dfrac{}{z : \mathbf{0} \vdash^{\perp} z : \mathbf{0}}}{\Delta, z : \mathbf{0} \vdash^{\perp} \mathbf{absurd}\ z : T} \quad o < \mathrm{pr}(\Delta)}{\Delta, z : \mathbf{0}, w : \mathbf{end}_{?}^{o+1} \vdash^{o} \mathbf{wait}\ w; \mathbf{absurd}\ z : T} \quad o \sqcup p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{o \sqcup p} \mathbf{let}\ (z, w) = \mathbf{recv}\ L\ \mathbf{in}\ \mathbf{wait}\ w; \mathbf{absurd}\ z : T}$$

(A) $= \mathbf{recv} : ?^{o}\mathbf{0}.\mathbf{end}_{?}^{o+1} \multimap^{\top, o} \mathbf{0} \times \mathbf{end}_{?}^{o+1}$ (B) $= \mathbf{wait} : \mathbf{end}_{?}^{o+1} \multimap^{\top, o} \mathbf{1}$

Figure 5: Typing Rules for Syntactic Sugar for PGV (T-Offer and T-Offer-Absurd).

Syntactic Sugar Typing. The following typing rules given in Figs. 3 to 5, cover syntactic sugar typing for PGV.

3. Technical Developments

3.1. **Subject Reduction.** Unlike with previous versions of GV, structural congruence, term reduction, and configuration reduction are all type preserving.

We must show that substitution preserves priority constraints. For this, we prove Lemma 3.1, which shows that values have finished all their communication, and that any priorities in the type of the value come from the typing environment.

Lemma 3.1. *If* $\Gamma \vdash^{p} V : T$*, then* $p = \bot$*, and* $\mathrm{pr}(\Gamma) = \mathrm{pr}(T)$.

Proof. By induction on the derivation of $\Gamma \vdash^{o} V : T$.

Case T-Var. *Immediately.*

$$\frac{}{x : T \vdash^{\bot} x : T}$$

Case T-Const. *Immediately.*

$$\frac{}{\varnothing \vdash^{\bot} K : T}$$

Case T-Lam. *Immediately.*

$$\frac{\Gamma, x : T \vdash^{q} M : U}{\Gamma \vdash^{\bot} \lambda x.M : T \multimap^{\mathrm{pr}(\Gamma), q} U}$$

Case T-Unit. *Immediately.*

$$\frac{}{\varnothing \vdash^{\bot} () : \mathbf{1}}$$

Case T-Pair. *The induction hypotheses give us* $p = q = \bot$*, hence* $p \sqcup q = \bot$*, and* $\mathrm{pr}(\Gamma) = \mathrm{pr}(T)$ *and* $\mathrm{pr}(\Delta) = \mathrm{pr}(U)$*, hence* $\mathrm{pr}(\Gamma, \Delta) = \mathrm{pr}(\Gamma) \sqcap \mathrm{pr}(\Delta) = \mathrm{pr}(T) \sqcap \mathrm{pr}(U) = \mathrm{pr}(T \times U)$.

$$\frac{\Gamma \vdash^{p} V : T \qquad \Delta \vdash^{q} W : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta \vdash^{p \sqcup q} (V, W) : T \times U}$$

Case T-Inl. *The induction hypothesis gives us* $p = \bot$*, and* $\mathrm{pr}(\Gamma) = \mathrm{pr}\,(T)$. *We know* $\mathrm{pr}(T) = \mathrm{pr}(U)$*, hence* $\mathrm{pr}(\Gamma) = \mathrm{pr}(T + U)$.

$$\frac{\Gamma \vdash^{p} V : T \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma \vdash^{p} \mathbf{inl}\ V : T + U}$$

Case T-Inr. *The induction hypothesis gives us* $p = \bot$*, and* $\mathrm{pr}(\Gamma) = \mathrm{pr}\,(U)$. *We know* $\mathrm{pr}(T) = \mathrm{pr}(U)$*, hence* $\mathrm{pr}(\Gamma) = \mathrm{pr}(T + U)$.

$$\frac{\Gamma \vdash^{p} V : U \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma \vdash^{p} \mathbf{inr}\ V : T + U}$$

□

Lemma 3.2. *If* $\Gamma, x : U' \vdash^{p} M : T$ *and* $\Theta \vdash^{q} V : U'$*, then* $\Gamma, \Theta \vdash^{p} M\{V/x\} : T$.

Proof. By induction on the derivation of $\Gamma, x : U' \vdash^{p} M : T$.

Case T-Var. *By Lemma 3.1,* $q = \bot$.

$$\frac{}{x : U' \vdash^{\bot} x : U'} \xrightarrow{\{V/x\}} \Theta \vdash^{\bot} V : U'$$

Case T-Lam. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$*, hence* $\mathrm{pr}(\Gamma, \Theta) = \mathrm{pr}(\Gamma, U')$.

$$\frac{\Gamma, x : U', y : T \vdash^{q} M : U}{\Gamma, x : U' \vdash^{\bot} \lambda y.M : T \multimap^{\mathrm{pr}(\Gamma, U'), q} U} \xrightarrow{\{V/x\}} \frac{\Gamma, \Theta, y : T \vdash^{q} M\{V/x\} : U}{\Gamma, \Theta \vdash^{\bot} \lambda y.M\{V/x\} : T \multimap^{\mathrm{pr}(\Gamma, \Theta), q} U}$$

Case T-App. *There are two subcases:*

Subcase $x \in M$. *Immediately, from the induction hypothesis.*

$$\frac{\Gamma, x : U' \vdash^{p} M : T \multimap^{p', q'} U \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta) \qquad q < p'}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q \sqcup q'} M\ N : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : T \multimap^{p', q'} U \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta) \qquad q < p'}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q \sqcup q'} (M\{V/x\})\ N : U}$$

Subcase $x \in N$. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$*, hence* $\mathrm{pr}(\Delta, \Theta) = \mathrm{pr}(\Delta, U')$.

$$\frac{\Gamma \vdash^{p} M : T \multimap^{p', q'} U \qquad \Delta, x : U' \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta, U') \qquad q < p'}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q \sqcup q'} M\ N : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma \vdash^{p} M : T \multimap^{p', q'} U \qquad \Delta, \Theta \vdash^{q} N\{V/x\} : T \qquad p < \mathrm{pr}(\Delta, \Theta) \qquad q < p'}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q \sqcup q'} M\ (N\{V/x\}) : U}$$

Case T-LetUnit. *There are two subcases:*

Subcase $x \in M$. *Immediately, from the induction hypothesis.*

$$\frac{\Gamma, x : U' \vdash^{p} M : \mathbf{1} \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} M; N : T} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : \mathbf{1} \qquad \Delta \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} M\{V/x\}; N : T}$$

Subcase $x \in N$. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$*, hence* $\mathrm{pr}(\Delta, \Theta) = \mathrm{pr}(\Delta, U')$.

$$\frac{\Gamma \vdash^{p} M : \mathbf{1} \qquad \Delta, x : U' \vdash^{q} N : T \qquad p < \mathrm{pr}(\Delta, U')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} M; N : T} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma \vdash^{p} M : \mathbf{1} \qquad \Delta, \Theta \vdash^{q} N\{V/x\} : T \qquad p < \mathrm{pr}(\Delta, \Theta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} M; N\{V/x\} : T}$$

Case T-Pair. *There are two subcases:*

Subcase $x \in M$. *Immediately, from the induction hypothesis.*

$$\frac{\Gamma, x : U' \vdash^{p} M : T \qquad \Delta \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, U')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} (M, N) : T \times U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : T \qquad \Delta \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, \Theta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} (M\{V/x\}, N) : T \times U}$$

Subcase $x \in N$. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$, *hence* $\mathrm{pr}(\Delta, \Theta) = \mathrm{pr}(\Delta, U')$.

$$\frac{\Gamma \vdash^{p} M : T \qquad \Delta, x : U' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, U')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} (M, N) : T \times U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma \vdash^{p} M : T \qquad \Delta, \Theta \vdash^{q} N\{V/x\} : U \qquad p < \mathrm{pr}(\Delta, \Theta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} (M, N\{V/x\}) : T \times U}$$

Case T-LetPair. *There are two subcases:*

Subcase $x \in M$. *Immediately, from the induction hypothesis.*

$$\frac{\Gamma, x : U' \vdash^{p} M : T \times T' \qquad \Delta, y : T, z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, T, T')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} \mathbf{let}\ (y, z) = M\ \mathbf{in}\ N : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : T \times T' \qquad \Delta, y : T, z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, T, T')}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} \mathbf{let}\ (y, z) = M\{V/x\}\ \mathbf{in}\ N : U}$$

Subcase $x \in N$. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$, *hence* $\mathrm{pr}(\Delta, \Theta, T, T') = \mathrm{pr}(\Delta, U', T, T')$.

$$\frac{\Gamma \vdash^{p} M : T \times T' \qquad \Delta, x : U', y : T, z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, U', T, T')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} \mathbf{let}\ (y, z) = M\ \mathbf{in}\ N : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma \vdash^{p} M : T \times T' \qquad \Delta, \Theta, y : T, z : T' \vdash^{q} N\{V/x\} : U \qquad p < \mathrm{pr}(\Delta, \Theta, T, T')}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} \mathbf{let}\ (y, z) = M\ \mathbf{in}\ N\{V/x\} : U}$$

Case T-Absurd.

$$\frac{\Gamma, x : U' \vdash^{p} M : \mathbf{0}}{\Gamma, \Delta, x : U' \vdash^{p} \mathbf{absurd}\ M : T} \xrightarrow{\{V/x\}} \frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : \mathbf{0}}{\Gamma, \Delta, \Theta \vdash^{p} \mathbf{absurd}\ M\{V/x\} : T}$$

Case T-Inl.

$$\frac{\Gamma, x : U' \vdash^{p} M : T \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma, x : U' \vdash^{p} \mathbf{inl}\ M : T + U} \xrightarrow{\{V/x\}} \frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : T \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma, \Theta \vdash^{p} \mathbf{inl}\ M\{V/x\} : T + U}$$

Case T-Inr.

$$\frac{\Gamma, x : U' \vdash^{p} M : U \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma, x : U' \vdash^{p} \mathbf{inr}\ M : T + U} \xrightarrow{\{V/x\}} \frac{\Gamma, \Theta \vdash^{p} M\{V/x\} : U \qquad \mathrm{pr}(T) = \mathrm{pr}(U)}{\Gamma, \Theta \vdash^{p} \mathbf{inr}\ M\{V/x\} : T + U}$$

Case T-CaseSum. *There are two subcases:*

Subcase $x \in L$. *Immediately, from the induction hypothesis.*

$$\frac{\Gamma, x : U' \vdash^{p} L : T + T' \qquad \Delta, y : T \vdash^{q} M : U \qquad \Delta, z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} \mathbf{case}\ L\ \{\mathbf{inl}\ y \mapsto M;\ \mathbf{inr}\ z \mapsto N\} : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma, \Theta \vdash^{p} L\{V/x\} : T + T' \qquad \Delta, y : T \vdash^{q} M : U \qquad \Delta, z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} \mathbf{case}\ L\{V/x\}\ \{\mathbf{inl}\ y \mapsto M;\ \mathbf{inr}\ z \mapsto N\} : U}$$

Subcase $x \in M$ *and* $x \in N$. *By Lemma 3.1,* $\mathrm{pr}(\Theta) = \mathrm{pr}(U')$, *hence* $\mathrm{pr}(\Delta, \Theta, T) = \mathrm{pr}(\Delta, U', T)$ *and* $\mathrm{pr}(\Delta, \Theta, T') = \mathrm{pr}(\Delta, U', T')$.

$$\frac{\Gamma \vdash^{p} L : T + T' \qquad \Delta, x : U', y : T \vdash^{q} M : U \qquad \Delta, x : U', z : T' \vdash^{q} N : U \qquad p < \mathrm{pr}(\Delta, U')}{\Gamma, \Delta, x : U' \vdash^{p \sqcup q} \mathbf{case}\ L\ \{\mathbf{inl}\ y \mapsto M;\ \mathbf{inr}\ z \mapsto N\} : U} \xrightarrow{\{V/x\}}$$

$$\frac{\Gamma \vdash^{p} L : T + T' \qquad \Delta, \Theta, y : T \vdash^{q} M\{V/x\} : U \qquad \Delta, \Theta, z : T' \vdash^{q} N\{V/x\} : U \qquad p < \mathrm{pr}(\Delta, \Theta)}{\Gamma, \Delta, \Theta \vdash^{p \sqcup q} \mathbf{case}\ L\ \{\mathbf{inl}\ y \mapsto M\{V/x\};\ \mathbf{inr}\ z \mapsto N\{V/x\}\} : U}$$

We omit the cases where $x \notin M$, as they are straightforward. □

Lemma 3.3. *If* $\Gamma \vdash^{p} M : T$ *and* $M \longrightarrow_{M} M'$, *then* $\Gamma \vdash^{p} M' : T$.

Proof. The proof closely follows the standard proof of subject reduction for the simply-typed linear λ-calculus, as the constants are uninterpreted by the term reduction ($\longrightarrow_{M}$) and priority constraints are maintained consequence of § 3.1.
By induction on the derivation of $M \longrightarrow_{M} M'$.

Case E-Lam. *By Lemma 3.2.*

$$\frac{\dfrac{\Gamma, x : T \vdash^{p} M : U}{\Gamma \vdash^{\bot} \lambda x.M : T \multimap^{\mathrm{pr}(\Gamma), p} U} \qquad \Delta \vdash^{\bot} V : T}{\Gamma, \Delta \vdash^{p} (\lambda x.M)\ V : U} \longrightarrow_{M} \Gamma, \Delta \vdash^{p} M\{V/x\} : U$$

Case E-Unit. *By Lemma 3.2.*

$$\frac{\dfrac{}{\varnothing \vdash^{\bot} () : \mathbf{1}} \qquad \Gamma \vdash^{p} M : T}{\Gamma \vdash^{p} (); M : T} \longrightarrow_{M} \Gamma \vdash^{p} M : T$$

Case E-Pair. *By Lemma 3.2.*

$$\frac{\dfrac{\Gamma \vdash^{\bot} V : T \qquad \Delta \vdash^{\bot} W : T'}{\Gamma, \Delta \vdash^{\bot} (V, W) : T \times T'} \qquad \Theta, x : T, y : T' \vdash^{p} M : U}{\Gamma, \Delta, \Theta \vdash \mathbf{let}\ (x, y) = (V, W)\ \mathbf{in}\ M : U}$$

$$\Big\downarrow_{M}$$

$$\Gamma, \Delta, \Theta \vdash^{p} M\{V/x\}\{W/y\} : U$$

Case E-Inl. *By Lemma 3.2.*

$$\dfrac{\dfrac{\Gamma \vdash^{\perp} V : T}{\Gamma \vdash^{\perp} \mathbf{inl}\ V : T + T'} \qquad \Delta, x : T \vdash^{p} M : U \qquad \Delta, y : T' \vdash^{p} N : U}{\Gamma, \Delta \vdash^{p} \mathbf{case}\ \mathbf{inl}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

$$\Longrightarrow_{M}$$

$$\Gamma, \Delta \vdash^{p} M\{V/x\} : U$$

Case E-Inr. *By Lemma 3.2.*

$$\dfrac{\dfrac{\Gamma \vdash^{\perp} V : T'}{\Gamma \vdash^{\perp} \mathbf{inr}\ V : T + T'} \qquad \Delta, x : T \vdash^{p} M : U \qquad \Delta, y : T' \vdash^{p} N : U}{\Gamma, \Delta \vdash^{p} \mathbf{case}\ \mathbf{inr}\ V\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U}$$

$$\Longrightarrow_{N}$$

$$\Gamma, \Delta \vdash^{p} N\{V/y\} : U$$

Case E-Lift. *Immediately by induction on the evaluation context* E.

□

Lemma 3.4. *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{C}'$*, then* $\Gamma \vdash^{\phi} \mathcal{C}'$.

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{C}'$.

Case SC-LinkSwap.

$$\dfrac{\dfrac{}{\mathbf{link} : S \times \overline{S} \multimap \mathbf{1}} \qquad \dfrac{\dfrac{}{x : S \vdash^{\perp} x : S} \qquad \dfrac{}{y : \overline{S} \vdash^{\perp} y : \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} (x, y) : S \times \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} \mathbf{link}\ (x, y) : \mathbf{1}}$$
$$\vdots$$
$$\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{F}[\mathbf{link}\ (x, y)]$$

$$\Vert$$

$$\dfrac{\dfrac{}{\mathbf{link} : \overline{S} \times S \multimap \mathbf{1}} \qquad \dfrac{\dfrac{}{y : \overline{S} \vdash^{\perp} y : \overline{S}} \qquad \dfrac{}{x : S \vdash^{\perp} x : S}}{x : S, y : \overline{S} \vdash^{\perp} (y, x) : S \times \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} \mathbf{link}\ (y, x) : \mathbf{1}}$$
$$\vdots$$
$$\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{F}[\mathbf{link}\ (y, x)]$$

Case SC-ResLink.

$$\dfrac{\dfrac{\dfrac{}{\mathbf{link} : S \times \overline{S} \multimap \mathbf{1}} \qquad \dfrac{\dfrac{}{x : S \vdash^{\perp} x : S} \qquad \dfrac{}{y : \overline{S} \vdash^{\perp} y : \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} (x, y) : S \times \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} \mathbf{link}\ (x, y) : \mathbf{1}}}{\dfrac{x : S, y : \overline{S} \vdash^{\phi} \phi\ \mathbf{link}\ (x, y)}{\varnothing \vdash^{\phi} (\nu xy)(\phi\ \mathbf{link}\ (x, y))}} \quad \equiv \quad \dfrac{\dfrac{}{\varnothing \vdash^{\phi} () : \mathbf{1}}}{\varnothing \vdash^{\phi} \phi\ ()}$$

Case SC-ResSwap.

$$\dfrac{\dfrac{\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu xy)\mathcal{C}}}{} \quad \equiv \quad \dfrac{\dfrac{\Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu yx)\mathcal{C}}}{}$$

Case SC-ResComm.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : S, y : \overline{S}, z : S', w : \overline{S'} \vdash^{\phi} \mathcal{C}}{\Gamma, x : S, y : \overline{S} \vdash^{\phi} (\nu zw)\mathcal{C}}}{\Gamma \vdash^{\phi} (\nu xy)(\nu zw)\mathcal{C}}}{} \quad \equiv \quad \dfrac{\dfrac{\dfrac{\Gamma, x : S, y : \overline{S}, z : S', w : \overline{S'} \vdash^{\phi} \mathcal{C}}{\Gamma, z : S', w : \overline{S'} \vdash^{\phi} (\nu xy)\mathcal{C}}}{\Gamma \vdash^{\phi} \vdash (\nu zw)(\nu xy)\mathcal{C}}}{}$$

Case SC-ResExt.

$$\dfrac{\dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \Delta, x : S, y : \overline{S} \vdash^{\phi} \mathcal{D}}{\Gamma, \Delta, x : S, y : \overline{S} \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D})}}{\Gamma, \Delta \vdash^{\phi} (\nu xy)(\mathcal{C} \parallel \mathcal{D})} \quad \equiv \quad \dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \dfrac{\Delta, x : S, y : \overline{S} \vdash^{\phi} \mathcal{D}}{\Delta \vdash^{\phi} (\nu xy)\mathcal{D}}}{\Gamma, \Delta \vdash^{\phi} \mathcal{C} \parallel (\nu xy)\mathcal{D}}$$

Case SC-ParNil.

$$\dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \dfrac{\dfrac{}{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\varnothing \vdash^{\circ} \circ()}}{\Gamma \vdash^{\phi} \mathcal{C} \parallel \circ()} \quad \equiv \Gamma \vdash^{\phi} \mathcal{C}$$

Case SC-ParComm.

$$\dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \Delta \vdash^{\phi'} \mathcal{D}}{\Gamma, \Delta \vdash^{\phi+\phi'} (\mathcal{C} \parallel \mathcal{D})} \quad \equiv \quad \dfrac{\Delta \vdash^{\phi'} \mathcal{D} \qquad \Gamma \vdash^{\phi} \mathcal{C}}{\Gamma, \Delta \vdash^{\phi'+\phi} (\mathcal{D} \parallel \mathcal{C})}$$

Case SC-ParAssoc.

$$\dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \dfrac{\Delta \vdash^{\phi'} \mathcal{D} \qquad \Theta \vdash^{\phi''} \mathcal{E}}{\Delta, \Theta \vdash^{\phi'+\phi''} (\mathcal{D} \parallel \mathcal{E})}}{\Gamma, \Delta, \Theta \vdash^{\phi+\phi'+\phi''} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})} \quad \equiv \quad \dfrac{\dfrac{\Gamma \vdash^{\phi} \mathcal{C} \qquad \Delta \vdash^{\phi'} \mathcal{D}}{\Gamma, \Delta \vdash^{\phi+\phi'} (\mathcal{C} \parallel \mathcal{D})} \qquad \Theta \vdash^{\phi''} \mathcal{E}}{\Gamma, \Delta, \Theta \vdash^{\phi+\phi'+\phi''} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

□

Theorem 3.5. *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{C}'$*, then* $\Gamma \vdash^{\phi} \mathcal{C}'$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{C}'$.

Case E-New.

$$\dfrac{\dfrac{\overline{\mathbf{new} : \mathbf{1} \multimap S \times \overline{S}} \qquad \overline{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\varnothing \vdash^{\perp} \mathbf{new}\ () : S \times \overline{S}} \\ \vdots \\ \Gamma \vdash^{\phi} \mathcal{F}[\mathbf{new}\ ()]}$$

$$\longrightarrow_{\mathcal{C}}$$

$$\dfrac{\dfrac{\overline{x : S \vdash^{\perp} x : S} \qquad \overline{y : \overline{S} \vdash^{\perp} y : \overline{S}}}{x : S, y : \overline{S} \vdash^{\perp} (x, y) : S \times \overline{S}} \\ \vdots \\ \Gamma, x : S, y : \overline{S} \vdash^{\phi} \mathcal{F}[(x, y)]}{\Gamma \vdash^{\phi} (\nu xy)\mathcal{F}[(x, y)]}$$

Case E-Spawn.

$$\dfrac{\dfrac{\overline{\mathbf{spawn} : (\mathbf{1} \multimap^{p,q} \mathbf{1}) \multimap \mathbf{1}} \quad \Delta \vdash^{\perp} V : \mathbf{1} \multimap^{p,q} \mathbf{1}}{\Delta \vdash^{\perp} \mathbf{spawn}\ V : \mathbf{1}} \\ \vdots \\ \Gamma, \Delta \vdash^{\phi} \mathcal{F}[\mathbf{spawn}\ V]}$$

$$\downarrow_{\mathcal{C}}$$

$$\dfrac{\dfrac{\overline{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\vdots} \qquad \dfrac{\Delta \vdash^{\perp} V : \mathbf{1} \multimap^{p,q} \mathbf{1} \qquad \overline{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\Delta \vdash^{q} V\ () : \mathbf{1}}}{\dfrac{\Gamma \vdash^{\phi} \mathcal{F}[()]}{\Gamma, \Delta \vdash^{\phi} \mathcal{F}[()] \parallel \circ\ (V\ ())} \qquad \Delta \vdash^{\circ} \circ\ (V\ ())}$$

Case E-Send. *See Fig. 6.*

Case E-Close.

$$\dfrac{\dfrac{\dfrac{\overline{\mathbf{close} : \mathbf{end}^{o}_{!} \multimap^{\top,o} \mathbf{1}} \qquad \overline{x : \mathbf{end}^{o}_{!} \vdash^{\perp} x : \mathbf{end}^{o}_{!}}}{x : \mathbf{end}^{o}_{!} \vdash^{o} \mathbf{close}\ x : \mathbf{1}} \\ \vdots \\ \Gamma, x : \mathbf{end}^{o}_{!} \vdash^{\phi} \mathcal{F}[\mathbf{close}\ x]}{\dfrac{\Gamma, \Delta, x : \mathbf{end}^{o}_{!}, y : \mathbf{end}^{o}_{?} \vdash^{\phi+\phi'} \mathcal{F}[\mathbf{close}\ x] \parallel \mathcal{F}'[\mathbf{wait}\ y]}{\Gamma, \Delta \vdash^{\phi+\phi'} (\nu xy)(\mathcal{F}[\mathbf{close}\ x] \parallel \mathcal{F}'[\mathbf{wait}\ y])}} \qquad \dfrac{\dfrac{\dfrac{\overline{\mathbf{wait} : \mathbf{end}^{o}_{?} \multimap^{\top,o} \mathbf{1}} \qquad \overline{y : \mathbf{end}^{o}_{?} \vdash^{\perp} y : \mathbf{end}^{o}_{?}}}{y : \mathbf{end}^{o}_{?} \vdash^{o} \mathbf{wait}\ y : \mathbf{1}} \\ \vdots \\ \Delta, y : \mathbf{end}^{o}_{?} \vdash^{\phi'} \mathcal{F}'[\mathbf{wait}\ y]}$$

$$\downarrow_{\mathcal{C}}$$

$$\dfrac{\dfrac{\overline{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\vdots} \qquad \dfrac{\overline{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\vdots}}{\dfrac{\Gamma \vdash^{\phi} \mathcal{F}[()]}{\Gamma, \Delta \vdash^{\phi+\phi'} \mathcal{F}[()] \parallel \mathcal{F}'[()]} \qquad \Delta \vdash^{\phi'} \mathcal{F}'[()]}$$

Case E-LiftC. *By induction on the evaluation context* $\mathcal{G}$.

Case E-LiftM. *By Lemma 3.3.*

Case E-LiftSC. *By Lemma 3.4.*

□

$$\dfrac{\dfrac{\dfrac{}{\textbf{send} : T \times {!^o}T.S \multimap^{\top,o} S} \quad \dfrac{\Delta \vdash^p V : T \qquad \overline{x : {!^o}T.S \vdash^{\bot} x : {!^o}T.S}}{\Delta, x : {!^o}T.S \vdash^p (V, x) : T \times {!^o}T.S}}{\Delta, x : {!^o}T.S \vdash^{p \sqcup o} \textbf{send}\ (V, x) : S}}{\vdots}}{\Gamma, \Delta, x : {!^o}T.S \vdash^{\phi} \mathcal{F}[\textbf{send}\ (V, x)]} \quad (\text{A}) \triangleq$$

$$\dfrac{\dfrac{\dfrac{}{\textbf{recv} : {?^o}T.\overline{S} \multimap^{\top,o} T \times \overline{S}} \quad \overline{y : {?^o}T.\overline{S} \vdash^{\bot} y : {?^o}T.\overline{S}}}{y : {?^o}T.\overline{S} \vdash^o \textbf{recv}\ y : T \times \overline{S}}}{\vdots}}{\Theta, y : {?^o}T.\overline{S} \vdash^{\phi'} \mathcal{F}'[\textbf{recv}\ y]} \quad (\text{B}) \triangleq$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{}{x : S \vdash^{\bot} x : S}}{\vdots}}{\Gamma, x : S \vdash^{\phi} \mathcal{F}[x]} \quad \dfrac{\dfrac{\Delta \vdash^p V : T \qquad \overline{\Delta, y : \overline{S} \vdash^{\bot} y : \overline{S}}}{\Delta, y : \overline{S} \vdash^p (V, y) : T \times \overline{S}}}{\vdots}}{\Delta, \Theta, y : \overline{S} \vdash^{\phi'} \mathcal{F}'[(V, y)]}$$

$$\dfrac{\Gamma, \Delta, \Theta, x : S, y : \overline{S} \vdash^{\phi+\phi'} \mathcal{F}[x] \parallel \mathcal{F}'[(V, y)]}{\Gamma, \Delta, \Theta \vdash^{\phi+\phi'} (\nu xy)(\mathcal{F}[x] \parallel \mathcal{F}'[(V, y)])}$$

Figure 6: Subject Reduction (E-Send)

3.2. **Progress and Deadlock Freedom.** PGV satisfies progress, as PGV configurations either reduce or are in normal form. However, the normal forms may seem surprising at first, as evaluating a well-typed PGV term does not necessarily produce just a value. If a term returns an endpoint, then its normal form contains a thread which is ready to communicate on the dual of that endpoint. This behaviour is not new to PGV.

Let us consider an example, adapted from Lindley and Morris [LM15], in which a term returns an endpoint linked to an echo server. The echo server receives a value and sends it back unchanged. Consider the program which creates a new channel, with endpoints x and

x', spawns off an echo server listening on x, and then returns x':

- $\textbf{let}\ (x, x') = \textbf{new}\ ()\ \textbf{in}$ $\quad \textbf{spawn}\ (\lambda().\textbf{echo}_x); x'$ $\qquad \textbf{echo}_x \triangleq \textbf{let}\ (y, x) = \textbf{recv}\ x\ \textbf{in}$ $\quad \textbf{let}\ x = \textbf{send}\ (y, x)\ \textbf{in close}\ x$

If we reduce the above program, we get $(\nu x x')(\circ\ \textbf{echo}_x \parallel \bullet\ x')$. Clearly, no more evaluation is possible, even though the configuration contains the thread $\circ\ \textbf{echo}_x$, which is blocked on x. In Corollary 3.14 we will show that if a term does not return an endpoint, it must produce only a value.

Actions are terms which perform communication actions and which synchronise between two threads.

Definition 3.6. A term *acts on* an endpoint x if it is $\textbf{send}\ (V, x)$, $\textbf{recv}\ x$, $\textbf{close}\ x$, or $\textbf{wait}\ x$. A term is an *action* if it acts on some endpoint x.

Ready terms are terms which perform communication actions, either by themselves, *e.g.*, creating a new channel or thread, or with another thread, *e.g.*, sending or receiving. It is worth mentioning that the notion of readiness presented here is akin to live processes introduced by Caires and Pfenning [CP10, DP22], and poised processes introduced by Pfenning and Griffith [PG15] and later used by Balzer *et al.* [BP17, BTP19]. Ready processes like live/poised processes denote processes that are ready to communicate on their providing channel.

Definition 3.7. A term L is *ready* if it is of the form $E[M]$, where M is of the form $\textbf{new}$, $\textbf{spawn}\ N$, $\textbf{link}\ (x, y)$, or M acts on x. In the latter case, we say that L is *ready to act on* x or is *blocked on*.

Progress for the term language is standard for GV, and deviates from progress for linear λ-calculus only in that terms may reduce to values or *ready terms*, where the definition of ready terms encompasses all terms whose reduction is struck on some constant K.

Lemma 3.8. *If* $\Gamma \vdash^p M : T$ *and* Γ *contains only session types, then:* (i) M *is a value;* (ii) $M \longrightarrow_M N$ *for some* N*; or* (iii) M *is ready.*

With "Γ contains only session types" we mean that for every $x : T \in \Gamma$, T is a session type, *i.e.*, is of the form S.

Canonical forms deviate from those for GV, in that we opt to move all ν-binders to the top. The standard GV canonical form, alternating ν-binders and their corresponding parallel compositions, does not work for PGV, since multiple channels may be split across a single parallel composition.

A configuration either reduces, or it is equivalent to configuration in normal form. Crucial to the normal form is that each term M_i is blocked on the corresponding channel x_i, and hence no two terms act on dual endpoints. Furthermore, no term M_i can perform a communication action by itself, since those are excluded by the definition of actions. Finally, as a corollary, we get that well-typed terms which do not return endpoints return just a value:

Definition 3.9. A configuration $\mathcal{C}$ is in canonical form if it is of the form $(\nu x_1 x_1') \ldots (\nu x_n x_n')(\circ\ M_1 \parallel \cdots \parallel \circ\ M_m \parallel \bullet\ N)$ where no term M_i is a value.

Lemma 3.10. *If* $\Gamma \vdash^\bullet \mathcal{C}$*, there exists some* $\mathcal{D}$ *such that* $\mathcal{C} \equiv \mathcal{D}$ *and* $\mathcal{D}$ *is in canonical form.*

Proof. We move any ν-binders to the top using SC-ResExt, discard any superfluous occurrences of $\circ\ ()$ using SC-ParNil, and move the main thread to the rightmost position using SC-ParComm and SC-ParAssoc. □

Definition 3.11. A configuration $\mathcal{C}$ is in normal form if it is of the form $(\nu x_1 x_1') \dots (\nu x_n x_n')(\circ\, M_1 \parallel \cdots \parallel \circ\, M_m \parallel \bullet\, V)$ where each M_i is ready to act on x_i.

Lemma 3.12. *If $\Gamma \vdash^p L : T$ is ready to act on $x : S \in \Gamma$, then the priority bound p is some priority o,* i.e., *not $\bot$ or $\top$.*

Proof. Let $L = E[M]$. By induction on the structure of E. M has priority $\mathrm{pr}(S)$, and each constructor of the evaluation context E passes on the *maximum* of the priorities of its premises. No rule introduces the priority bound $\top$ on the sequent. □

Theorem 3.13. *If $\varnothing \vdash^\bullet \mathcal{C}$ and $\mathcal{C}$ is in canonical form, then either $\mathcal{C} \longrightarrow_{\mathcal{C}} \mathcal{D}$ for some $\mathcal{D}$; or $\mathcal{C} \equiv \mathcal{D}$ for some $\mathcal{D}$ in normal form.*

Proof. Let $\mathcal{C} = (\nu x_1 x_1') \dots (\nu x_n x_n')(\circ\, M_1 \parallel \cdots \parallel \circ\, M_m \parallel \bullet\, N)$. We apply Lemma 3.8 to each M_i and N. If for any M_i or N we obtain a reduction $M_i \longrightarrow_M M_i'$ or $N \longrightarrow_M N'$, we apply E-LiftM and E-LiftC to obtain a reduction on $\mathcal{C}$. Otherwise, each term M_i is ready, and N is either ready or a value. Pick the *ready* term $L \in \{M_1, \dots, M_m, N\}$ with the smallest priority bound.

(1) If L is a new $E[\mathbf{new}\ ()]$, we apply E-New.
(2) If L is a spawn $E[\mathbf{spawn}\ M]$, we apply E-Spawn.
(3) If L is a link $E[\mathbf{link}\ (y, z)]$ or $E[\mathbf{link}\ (z, y)]$, we apply E-Link.
(4) Otherwise, L is ready to act on some endpoint $y : S$. Let $y' : \overline{S}$ be the dual endpoint of y. The typing rules enforce the linear use of endpoints, so there must be a term $L' \in \{M_1, \dots, M_m, N\}$ which uses y'. L' must be either a ready term or a value:
   (a) L' is ready. By Lemma 3.12, the priority of L is $\mathrm{pr}(S)$. By duality, $\mathrm{pr}(\overline{S}) = \mathrm{pr}(S)$. We cannot have $L = L'$, otherwise the action on y' would be guarded by the action on y, requiring $\mathrm{pr}(\overline{S}) < \mathrm{pr}(S)$.
   The term L' must be ready to act on y', otherwise the action y' would be guarded by another action with priority smaller than $\mathrm{pr}\,(S)$, which contradicts our choice of L as having the smallest priority.
   Therefore, we have two terms ready to act on dual endpoints. We apply the appropriate reduction rule, *i.e.*, E-Send or E-Close.
   (b) $L' = N$ and is a value. We rewrite $\mathcal{C}$ to put L in the position corresponding to the endpoint it is blocked on, using SC-ParComm, SC-ParAssoc, and optionally SC-ResSwap. We then repeat the steps above with the term with the next smallest priority, until either we find a reduction, or the configuration has reached the desired normal form.
   The argument based on the priority being the smallest continues to hold, since we know that neither L nor L' will be picked, and no other term uses y or y'.

□

Corollary 3.14. *If $\varnothing \vdash^\phi \mathcal{C}$, $\mathcal{C} \not\longrightarrow_{\mathcal{C}}$, and $\mathcal{C}$ contains no endpoints, then $\mathcal{C} \equiv \phi\, V$ for some value V.*

An immediate consequence of Theorem 3.13 and Corollary 3.14 is that *a term which does not return an endpoint will complete all its communication actions, thus satisfying deadlock freedom.*

4. Relation to Priority CP

Thus far we have presented Priority GV (PGV) together with the relevant technical results. We remind the reader that this line of work of adding priorities, started with Priority CP (PCP) [DG18a] where priorities are integrated in Wadler's Classical Processes (CP), which is a π-calculus leveraging the correspondence of session types as linear logic propositions [Wad12]. In his work, Wadler presents a connection (via encoding) of CP and GV. Following that work, we sat out to understand the connection between the priority versions of CP and GV, thus comparing PGV and PCP. Before presenting our formal results, we will revisit PCP in the following section.

4.1. **Revisiting Priority CP.**

Types. Types (A, B) in PCP are based on classical linear logic propositions, and are defined by the following grammar:

$$A, B \quad ::= \quad A \otimes^o B \mid A \parr^o B \mid \mathbf{1}^o \mid \bot^o \mid A \oplus^o B \mid A \,\&^o\, B \mid \mathbf{0}^o \mid \top^o$$

Each connective is annotated with a priority $o \in \mathbb{N}$.

Types $A \otimes^o B$ and $A \parr^o B$ type the endpoints of a channel over which we send or receive a channel of type A, and then proceed as type B. Types $\mathbf{1}^o$ and $\bot[o]$ type the endpoints of a channel whose session has terminated, and over which we send or receive a *ping* before closing the channel. These two types act as units for $A \otimes^o B$ and $A \parr^o B$, respectively.

Types $A \oplus^o B$ and $A \,\&^o\, B$ type the endpoints of a channel over which we can receive or send a choice between two branches A or B. We have opted for a simplified version of choice and followed the original Wadler's CP [Wad14], however types $\oplus$ and $\&$ can be trivially generalised to $\oplus^o\{l_i : A_i\}_{i\in I}$ and $\&^o\{l_i : A_i\}_{i\in I}$, respectively, as in the original PCP [DG18b].

Types $\mathbf{0}^o$ and $\top^o$ type the endpoints of a channel over which we can send or receive a choice between no options. These two types act as units for $A \oplus^o B$ and $A \,\&^o\, B$, respectively.

Typing Environments. Typing environments Γ, Δ associate names to types. Environments are linear, so two environments can only be combined as Γ, Δ if their names are distinct, *i.e.*, $\mathrm{fv}(\Gamma) \cap \mathrm{fv}(\Delta) = \varnothing$.

$$\Gamma, \Delta \quad ::= \quad \varnothing \mid \Gamma, x : A$$

Type Duality. Duality is an involutive function on types which preserves priorities:

$$\begin{array}{llll} (\mathbf{1}^o)^\perp = \bot^o & (A \otimes^o B)^\perp = A^\perp \parr^o B^\perp & (\mathbf{0}^o)^\perp = \top^o & (A \oplus^o B)^\perp = A^\perp \,\&^o\, B^\perp \\ (\bot^o)^\perp = \mathbf{1}^o & (A \parr^o B)^\perp = A^\perp \otimes^o B^\perp & (\top^o)^\perp = \mathbf{0}^o & (A \,\&^o\, B)^\perp = A^\perp \oplus^o B^\perp \end{array}$$

Priorities. The function $\mathrm{pr}(\cdot)$ returns smallest priority of a type. As with PGV, the type system guarantees that the top-most connective always holds the smallest priority. The function $\mathrm{pr}(\cdot)$ returns the *minimum* priority of all types a typing context, or $\top$ if the context is empty:

$$\begin{array}{llll} \mathrm{pr}(\mathbf{1}^o) = o & \mathrm{pr}(A \otimes^o B) = o & \mathrm{pr}(\mathbf{0}^o) = o & \mathrm{pr}(A \oplus^o B) = o \\ \mathrm{pr}(\bot^o) = o & \mathrm{pr}(A \parr^o B) = o & \mathrm{pr}(\top^o) = o & \mathrm{pr}(A \,\&^o\, B) = o \end{array}$$

$$\mathrm{pr}(\varnothing) = \top \quad \mathrm{pr}(\Gamma, x : T) = \mathrm{pr}(\Gamma) \sqcap \mathrm{pr}(T)$$

Terms. Processes (P, Q) in PCP are defined by the following grammar.

$$\begin{array}{rcl} P, Q & ::= & x \leftrightarrow y \mid (\nu xy)P \mid (P \parallel Q) \mid \mathbf{0} \\ & \mid & x[y].P \mid x[].P \mid x(y).P \mid x().P \\ & \mid & x \triangleleft \mathrm{inl}.P \mid x \triangleleft \mathrm{inr}.P \mid x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \mid x \triangleright \{\} \end{array}$$

Process $x \leftrightarrow y$ links endpoints x and y and forwards communication from one to the other. $(\nu xy)P$, $(P \parallel Q)$ and $\mathbf{0}$ denote respectively the restriction processes where channel endpoints x and y are bound together and with scope P, the parallel composition of processes P and Q and the terminated process.

Processes $x[y].P$ and $x(y).P$ send or receive over channel x a value y and proceed as process P. Processes $x[].P$ and $x().P$ send and receive an empty value—denoting the closure of channel x, and continue as P.

Processes $x \triangleleft \mathrm{inl}.P$ and $x \triangleleft \mathrm{inr}.P$ make a left and right choice, respectively and proceed as process P. Dually, $x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}$ offers both left and right branches, with continuations P and Q, and $x \triangleright \{\}$ is the empty offer.

We write *unbound* send as $x\langle y\rangle.P$, which is syntactic sugar for $x[z].(y \leftrightarrow z \parallel P)$. Alternatively, we could take $x\langle y\rangle.P$ as primitive, and let $x[y].P$ be syntactic sugar for $(\nu yz)(x\langle z\rangle.P)$. CP takes *bound* sending as primitive, as it is impossible to eliminate the top-level cut in terms such as $(\nu yz)(x\langle z\rangle.P)$, even with commuting conversions. In our setting without commuting conversions and with more permissive normal forms, this is no longer an issue, but, for simplicity, we keep bound sending as primitive.

On Commuting Conversions. The main change we make to PCP is removing commuting conversions. Commuting conversions are necessary if we want our reduction strategy to correspond exactly to cut (or cycle in [DG18b]) elimination. However, as Lindley and Morris [LM15] show, all communications that can be performed *with* the use of commuting conversions, can also be performed *without* them, but using structural congruence.

From the perspective of process calculi, commuting conversions behave strangely. Consider the commuting conversion ($\kappa_{\parr}$) for $x(y).P$:

$$(\kappa_{\parr}) \quad (\nu zz')(x(y).P \parallel Q) \Longrightarrow x(y).(\nu zz')(P \parallel Q)$$

As a result of ($\kappa_{\parr}$), Q becomes blocked on $x(y)$, and any actions Q was able to perform become unavailable. Consequently, CP is non-confluent:

$$\begin{array}{ccc} (\nu xx')(a(y).P \parallel (\nu zz')(z[].\mathbf{0} \parallel z'().Q)) & & \\ \Downarrow & & \Downarrow{\scriptstyle +} \\ a(y).(\nu xx')(P \parallel (\nu zz')(z[].\mathbf{0} \parallel z'().Q)) & & a(y).(\nu xx')(P \parallel Q) \end{array}$$

Structural congruence.

$$
\begin{array}{lllll}
\text{SC-LinkSwap} & x{\leftrightarrow}y & \equiv & y{\leftrightarrow}x \\
\text{SC-ResLink} & (\nu xy)x{\leftrightarrow}y & \equiv & \mathbf{0} \\
\text{SC-ResSwap} & (\nu xy)P & \equiv & (\nu yx)P \\
\text{SC-ResComm} & (\nu xy)(\nu zw)P & \equiv & (\nu zw)(\nu xy)P \\
\text{SC-ResExt} & (\nu xy)(P \parallel Q) & \equiv & P \parallel (\nu xy)Q, \text{ if } x, y \notin \mathrm{fv}(P) \\
\text{SC-ParNil} & P \parallel \mathbf{0} & \equiv & P \\
\text{SC-ParComm} & P \parallel Q & \equiv & Q \parallel P \\
\text{SC-ParAssoc} & P \parallel (Q \parallel R) & \equiv & (P \parallel Q) \parallel R
\end{array}
$$

Reduction.

$$
\begin{array}{lllll}
\text{E-Link} & (\nu xy)(w{\leftrightarrow}x \parallel P) & \Longrightarrow & P\{w/x\} \\
\text{E-Send} & (\nu xy)(x[z].P \parallel x(w).Q) & \Longrightarrow & (\nu xy)(\nu zw)(P \parallel Q) \\
\text{E-Close} & (\nu xy)(x[].P \parallel y().Q) & \Longrightarrow & P \parallel Q \\
\text{E-Select-Inl} & (\nu xy)(x \triangleleft \mathrm{inl}.P \parallel x \triangleright \{\mathrm{inl} : Q; \mathrm{inr} : R\}) & \Longrightarrow & (\nu xy)(P \parallel Q) \\
\text{E-Select-Inr} & (\nu xy)(x \triangleleft \mathrm{inr}.P \parallel x \triangleright \{\mathrm{inl} : Q; \mathrm{inr} : R\}) & \Longrightarrow & (\nu xy)(P \parallel R)
\end{array}
$$

$$
\text{E-LiftRes}\ \frac{P \Longrightarrow P'}{(\nu xy)P \Longrightarrow (\nu xy)P'} \qquad \text{E-LiftPar}\ \frac{P \Longrightarrow P'}{P \parallel Q \Longrightarrow P' \parallel Q} \qquad \text{E-LiftSC}\ \frac{P \equiv P' \quad P' \Longrightarrow Q' \quad Q' \equiv Q}{P \Longrightarrow Q}
$$

Figure 7: Operational Semantic for PCP.

In PCP, commuting conversions break our intuition that an action with lower priority occurs before an action with higher priority. To cite Dardha and Gay [DG18b] "*if a prefix on a channel endpoint x with priority o is pulled out at top level, then to preserve priority constraints in the typing rules [..], it is necessary to increase priorities of all actions after the prefix on x*" by $o + 1$.

4.2. **Operational Semantics.** The operational semantics for PCP, given in Fig. 7, is defined as a reduction relation $\Longrightarrow$ on processes (bottom) and uses structural congruence (top). Each of the axioms of structural congruence corresponds to the axiom of the same name for PGV. We write $\Longrightarrow^+$ for the transitive closures, and $\Longrightarrow^\star$ for the reflexive-transitive closures.

The reduction relation is given by a set of axioms and inference rules for context closure. Reduction occurs under restriction. E-Link reduces a parallel composition with a link into a substitution. E-Send is the main communication rule, where send and receive processes sychronise and reduce to the corresponding continuations. E-Close follows the previous rule and it closes the channel identified by endpoints x and y. E-Select-Inl and E-Select-Inr are generalised versions of E-Send. They state respectively that a left and right selection synchronises with a choice offering and reduces to the corresponding continuations. The last three rules state that reduction is closed under restriction, parallel composition and structural congruence, respectively.

4.3. **Typing Rules.** Figure 8 gives the typing rules for our version of PCP. A typing judgement $P \vdash \Gamma$ states that "process P is well typed under the typing context Γ".

$$\textsc{T-Link}\ \frac{}{x{\leftrightarrow}^{A}y \vdash x : A, y : A^{\perp}} \qquad \textsc{T-Res}\ \frac{P \vdash \Gamma, x : A, y : A^{\perp}}{(\nu xy)P \vdash \Gamma} \qquad \textsc{T-Par}\ \frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \parallel Q \vdash \Gamma, \Delta} \qquad \textsc{T-Halt}\ \frac{}{\mathbf{0} \vdash \varnothing}$$

$$\textsc{T-Send}\ \frac{P \vdash \Gamma, y : A, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x[y].P \vdash \Gamma, x : A \otimes^{o} B} \qquad \textsc{T-Close}\ \frac{P \vdash \Gamma \qquad o < \mathrm{pr}(\Gamma)}{x[].P \vdash \Gamma, x : \mathbf{1}^{o}}$$

$$\textsc{T-Recv}\ \frac{P \vdash \Gamma, y : A, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x(y).P \vdash \Gamma, x : A \parr^{o} B} \qquad \textsc{T-Wait}\ \frac{P \vdash \Gamma \qquad o < \mathrm{pr}(\Gamma)}{x().P \vdash \Gamma, x : \bot^{o}}$$

$$\textsc{T-Select-Inl}\ \frac{P \vdash \Gamma, x : A \qquad o < \mathrm{pr}(\Gamma, A, B) \qquad \mathrm{pr}(A) = \mathrm{pr}(B)}{x \triangleleft \mathrm{inl}.P \vdash \Gamma, x : A \oplus^{o} B}$$

$$\textsc{T-Select-Inr}\ \frac{P \vdash \Gamma, x : B \qquad o < \mathrm{pr}(\Gamma, A, B) \qquad \mathrm{pr}(A) = \mathrm{pr}(B)}{x \triangleleft \mathrm{inr}.P \vdash \Gamma, x : A \oplus^{o} B}$$

$$\textsc{T-Offer}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \vdash \Gamma, x : A \,\&^{o}\, B} \qquad \textsc{T-Offer-Absurd}\ \frac{o < \mathrm{pr}(\Gamma)}{x \triangleright \{\} \vdash \Gamma, x : \top^{o}}$$

Figure 8: Typing Rules for PCP.

T-Link states that the link process $x{\leftrightarrow}y$ is well typed under channels x and y having dual types, respectively A and $A^{\perp}$. T-Res states that the restriction process $(\nu xy)P$ is well typed under typing context Γ if process P is well typed in Γ augmented with channel endpoints x and y having dual types, respectively A and $A^{\perp}$. T-Par states that the parallel composition of processes P and Q is well typed under the disjoint union of their respective typing contexts. T-Halt states that the terminated process $\mathbf{0}$ is well typed in the empty context.

T-Send and T-Recv state that the sending and receiving of a bound name y over a channel x is well typed under Γ and x of type $A \otimes^{o} B$, respectively $A \parr^{o} B$. Priority o is the smallest among all priorities of the types used by the output or input process, captured by the side condition $o < \mathrm{pr}(\Gamma, A, B)$.

Rules T-Close and T-Wait type the closure of channel x and are in the same lines as the previous two rules, requiring that the priority of channel x is the smallest among all priorities in Γ.

T-Select-Inl and T-Select-Inr type respectively the left $x \triangleleft \mathrm{inl}.P$ and right $x \triangleleft \mathrm{inr}.P$ choice performed on channel x. T-Offer and T-Offer-Absurd type the offering of a choice, or empty choice, on channel x. In all the above rules the priority o of channel x is the smallest with respect to the typing context $o < \mathrm{pr}(\Gamma)$ and types involved in the choice $o < \mathrm{pr}(\Gamma, A, B)$.

Figure 9 shows how syntactic sugar in PCP is well typed.

$$\text{T-UnboundSend}\quad \frac{\overline{P \vdash \Gamma, x : B} \qquad o < \mathrm{pr}(\Gamma, A, B)}{x\langle y\rangle.P \vdash \Gamma, x : A \otimes B, y : A^{\perp}} \quad \triangleq$$

$$\frac{\dfrac{\overline{z \leftrightarrow^{A} y \vdash y : A^{\perp}, z : A} \qquad \overline{P \vdash \Gamma, x : B}}{z \leftrightarrow^{A} y \parallel P \vdash \Gamma, x : B, y : A^{\perp}, z : A} \qquad o < \mathrm{pr}(\Gamma, A, B)}{x[z].(z \leftrightarrow^{A} y \parallel P) \vdash \Gamma, x : A \otimes B, y : A^{\perp}}$$

Figure 9: Typing Rules for Syntactic Sugar for PCP.

Finally, since our reduction relation is a strict subset of the reduction relation in the original [DG18b], we defer to their proof of subject reduction (Theorem 2 in [DG18b]). We prove progress for our version of PCP, see § 4.5.

4.4. **PCP and PLL.** In this subsection, we highlight the connection between PCP and linear logic. Dardha and Gay [DG18a] present PCP–consequently also PCP given in this paper–in a way which can be viewed both as Classical Processes with restriction (T-Res) and parallel composition (T-Par) typing rules, and as a new version of linear logic, which they call Priority Linear Logic (PLL). PLL builds on CLL by replacing the cut rule with two logical rules: a mix and a cycle rule—here corresponding to T-Par and T-Res, respectively. Dardha and Gay [DG18a, §4] prove cycle-elimination, in the same lines as cut-elimination for CLL. As a corollary of cycle-elimination for PLL, we obtain deadlock freedom for PCP (Theorem 3 in [DG18a, §4]). In summary, PLL is an extension of CLL and the authors show the correspondence of PCP and PLL. Notice however, that PCP is not in correspondence with CLL itself, since processes in PCP are graphs, whether CLL induces trees.

4.5. **Technical Developments.**

Definition 4.1. A process acts on an endpoint x if it is $x \leftrightarrow y$, $y \leftrightarrow x$, $x[y].P$, $x(y).P$, $x[].P$, $x().P$, $x \triangleleft \mathrm{inl}.P$, $x \triangleleft \mathrm{inr}.P$, $x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}$, or $x \triangleright \{\}$. A process is an action if it acts on some endpoint x.

Definition 4.2. A process P is in canonical form if it is either $\mathbf{0}$ or of the form $(\nu x_1 x_1') \ldots (\nu x_n x_n')(P_1 \parallel \cdots \parallel P_m)$ where $m > 0$ and each P_j is an action.

Lemma 4.3. *If $P \vdash \Gamma$, there exists some Q such that $P \equiv Q$ and Q is in canonical form.*

Proof. If $P = \mathbf{0}$, we are done. Otherwise, we move any ν-binders to the top using SC-ResExt, and discard any superfluous occurrences of $\mathbf{0}$ using SC-ParNil. □

The proof for progress (below) follows the same reasoning by Kobayashi [Kob06] used in the proof of deadlock freedom for closed processes (Theorem 2).

Theorem 4.4. *If $P \vdash \varnothing$, then either $P = \mathbf{0}$ or there exists a Q such that $P \Longrightarrow Q$.*

Proof (Sketch). This proof follows the exact same reasoning and proof sketch given by Kobayashi in [Kob06] and later adopted by Dardha and Gay for PCP in their technical report [DG18b].

By Lemma 4.3, we rewrite P to canonical form. If the resulting process is $\mathbf{0}$, we are done. Otherwise, it is of the form

$$(\nu x_1 x_1')\ldots(\nu x_n x_n')(P_1 \parallel \cdots \parallel P_m) \vdash \varnothing$$

where $m > 0$ and each $P_i \vdash \Gamma_i$ is an action.

Consider processes $P_1 \parallel \cdots \parallel P_m$. Among them, we pick the process with the smallest priority $\mathrm{pr}\,(\Gamma_i)$ for all i. Let this process be P_i and the priority of the top prefix be o. P_i acts on some endpoint $y : A \in \Gamma_i$. We must have $\mathrm{pr}\,(\Gamma_i) = \mathrm{pr}\,(A) = o$, since the other actions in P_i are guarded by the action on $y : A$, thus satisfying law (i) of priorities.

If P_i is a link $y{\leftrightarrow}z$ or $z{\leftrightarrow}y$, we apply E-Link.

Otherwise, P_i is an input/branching or output/selection action on endpoint y of type A with priority o. Since process P is closed and consequently it respects law (ii) of priorities, there must be a co-action y' of type $A^\perp$ where y and y' are dual endpoints of the same channel (by application of rule T-Res). By duality, $\mathrm{pr}\,(A) = \mathrm{pr}\,(A^\perp) = o$. In the following we show that: y' is the subject of a top level action of a process P_j with $i \neq j$. This allows for the communication among P_i and P_j to happen immediately over channel endpoints y and y'.

Suppose that y' is an action not in a different parallel process P_j but rather of P_i itself. That means that the action on y' must be prefixed by the action on y, which is top level in P_i. To respect law (i) of priorities we must have $o < o$, which is absurd. This means that y' is in another parallel process P_j for $i \neq j$.

Suppose that y' in P_j is not at top level. In order to respect law (i) of priorities, it means that y' is prefixed by actions that are smaller than its priority o. This leads to a contradiction because stated that o is the smallest priority. Hence, y' must be the subject of a top level action.

We have two processes, acting on dual endpoints. We apply the appropriate reduction rule, *i.e.*, E-Send, E-Close, E-Select-Inl, or E-Select-Inr. □

4.6. **Correspondence between PGV and PCP.** We illustrate the relation between PCP and PGV by defining a translation $(\!|\cdot|\!)_c$ from PCP processes to PGV configurations which satisfies operational correspondence.

The translation $(\!|\cdot|\!)_c$ translates PCP processes to PGV configurations by translating as much as possible of the π-calculus constructs from PCP to the identical configuration constructs in PGV, *i.e.*, all top-level ν and $\parallel$ constructs. When it encounters the first action, it translates the remainder of the processs to a term using $(\!|\cdot|\!)_M$. The translation $(\!|\cdot|\!)_M$ translates PCP processes to PGV terms by mapping the π-calculus constructs from PCP (*e.g.*, ν, $\parallel$, $\cdot[\cdot].\cdot$, $\cdot[\cdot].\cdot$, $\cdot(\cdot).\cdot$, *etc.*) to the corresponding constants in PGV (*e.g.*, **new**, **spawn**, **send**, **recv**, *etc.*). Translating a process with $(\!|\cdot|\!)_c$ is the same as translating that process with $(\!|\cdot|\!)_M$ followed by several steps of configuration reduction. The translation $(\!|\cdot|\!)$ tranlates session types from PCP to session types in PGV matching the translation on processes.

The translation on types is defined as follows:

$$\begin{array}{lcllcl}
(\!|A \otimes^o B|\!) & = & !^o\overline{(\!|A|\!)}.(\!|B|\!) & (\!|\mathbf{1}^o|\!) & = & \mathbf{end}^o_! \\
(\!|A \oplus^o B|\!) & = & (\!|A|\!) \oplus^o (\!|B|\!) & (\!|\mathbf{0}^o|\!) & = & \oplus^o\{\} \\
\\
(\!|A \parr^o B|\!) & = & ?^o(\!|A|\!).(\!|B|\!) & (\!|\bot^o|\!) & = & \mathbf{end}^o_? \\
(\!|A \mathbin{\&^o} B|\!) & = & (\!|A|\!) \mathbin{\&^o} (\!|B|\!) & (\!|\top^o|\!) & = & \&^o\{\}
\end{array}$$

The translation $(\!|\cdot|\!)_M$ translates processes to *terms* and maps the π-calculus constructs from PCP to the corresponding constants in PGV:

$$
\begin{aligned}
(\!|x \leftrightarrow y|\!)_M &= \mathbf{link}\ (x, y) \\
(\!|(\nu xy)P|\!)_M &= \mathbf{let}\ (x, y) = \mathbf{new}\ ()\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|P \parallel Q|\!)_M &= \mathbf{spawn}\ (\lambda().(\!|P|\!)_M); (\!|Q|\!)_M \\
(\!|\mathbf{0}|\!)_M &= () \\
(\!|x[].P|\!)_M &= \mathbf{close}\ x; (\!|P|\!)_M \\
(\!|x().P|\!)_M &= \mathbf{wait}\ x; (\!|P|\!)_M \\
(\!|x[y].P|\!)_M &= \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x(y).P|\!)_M &= \mathbf{let}\ (y, x) = \mathbf{recv}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleleft \mathrm{inl}.P|\!)_M &= \mathbf{let}\ x = \mathbf{select}\ \mathbf{inl}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleleft \mathrm{inr}.P|\!)_M &= \mathbf{let}\ x = \mathbf{select}\ \mathbf{inr}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
(\!|x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\}|\!)_M &= \mathbf{offer}\ x\ \{\mathbf{inl}\ x \mapsto (\!|P|\!)_M; \mathbf{inr}\ x \mapsto (\!|Q|\!)_M\} \\
(\!|x \triangleright \{\}|\!)_M &= \mathbf{offer}\ x\ \{\}
\end{aligned}
$$

Unfortunately, the operational correspondence along $(\!|\cdot|\!)_M$ is unsound, as it translates ν-binders and parallel compositions to **new** and **spawn**, which can reduce to their equivalent configuration constructs using E-New and E-Spawn. The same goes for ν-binders which are inserted when translating bound send to unbound send. For instance, the process $x[y].P$ is blocked, but its translation uses **new** and can reduce. To address this issue, we introduce a second translation, $(\!|\cdot|\!)_C$, which is equivalent to translating with $(\!|\cdot|\!)_M$ then reducing with E-New and E-Spawn:

$$
\begin{aligned}
(\!|(\nu xy)P|\!)_C &= (\nu xy)(\!|P|\!)_C \\
(\!|P \parallel Q|\!)_C &= (\!|P|\!)_C \parallel (\!|Q|\!)_C \\
(\!|x[y].P|\!)_C &= (\nu yz)(\circ\ \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|x \triangleleft \mathrm{inl}.P|\!)_C &= (\nu yz)(\circ\ \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|x \triangleleft \mathrm{inr}.P|\!)_C &= (\nu yz)(\circ\ \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
(\!|P|\!)_C &= \circ(\!|P|\!)_M, \quad \text{if none of the above apply}
\end{aligned}
$$

Typing environments are translated pointwise, and sequents $P \vdash \Gamma$ are translated as $(\!|\Gamma|\!) \vdash^{\circ} (\!|P|\!)_C$, where $\circ$ indicates a child thread, since translated processes do not have a main thread. The translations $(\!|\cdot|\!)_M$ and $(\!|\cdot|\!)_C$ preserve typing, and the latter induces a sound and complete operational correspondence.

Lemma 4.5. *If* $P \vdash \Gamma$, *then* $(\!|\Gamma|\!) \vdash^{p} (\!|P|\!)_M : \mathbf{1}$.

Proof. By induction on the derivation of $P \vdash \Gamma$.

Case T-Link, T-Res, T-Par, and T-Halt. *See Fig. 10.*

Case T-Close, and T-Wait. *See Fig. 11.*

Case T-Send. *See Fig. 12.*

Case T-Recv. *See Fig. 13.*

$$\textsc{T-Link}\quad \frac{}{x \leftrightarrow^{A} y \vdash x : A, y : A^{\perp}} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{}{\mathbf{link} : (\!|A|\!) \times \overline{(\!|A|\!)} \multimap \mathbf{1}} \qquad \dfrac{\dfrac{}{x : (\!|A|\!) \vdash^{\perp} x : (\!|A|\!)} \qquad \dfrac{}{y : \overline{(\!|A|\!)} \vdash^{\perp} y : \overline{(\!|A|\!)}}}{x : (\!|A|\!), y : \overline{(\!|A|\!)} \vdash^{\perp} (x, y) : (\!|A|\!) \times \overline{(\!|A|\!)}}}{x : (\!|A|\!), y : \overline{(\!|A|\!)} \vdash^{\perp} \mathbf{link}\ (x, y) : \mathbf{1}}$$

$$\textsc{T-Res}\quad \frac{P \vdash \Gamma, x : A, y : A^{\perp}}{(\nu xy) P \vdash \Gamma} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\dfrac{}{\mathbf{new} : \mathbf{1} \multimap (\!|A|\!) \times \overline{(\!|A|\!)}} \qquad \dfrac{}{\varnothing \vdash^{\perp} () : \mathbf{1}}}{\varnothing \vdash^{\perp} \mathbf{new}\ () : (\!|A|\!) \times \overline{(\!|A|\!)}} \qquad (\!|\Gamma|\!), x : (\!|A|\!), y : \overline{(\!|A|\!)} \vdash^{p} (\!|P|\!)_M : \mathbf{1}}{(\!|\Gamma|\!) \vdash^{p} \mathbf{let}\ (x, y) = \mathbf{new}\ ()\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}}$$

$$\textsc{T-Par}\quad \frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \parallel Q \vdash \Gamma, \Delta} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\dfrac{}{\mathbf{spawn} : (\mathbf{1} \multimap^{\mathrm{pr}\,(\Gamma), p} \mathbf{1}) \multimap \mathbf{1}} \qquad \dfrac{(\!|\Gamma|\!) \vdash^{p} (\!|P|\!)_M : \mathbf{1}}{(\!|\Gamma|\!) \vdash^{\perp} \lambda().(\!|P|\!)_M : \mathbf{1} \multimap^{\mathrm{pr}\,(\Gamma), p} \mathbf{1}}}{(\!|\Gamma|\!) \vdash^{\perp} \mathbf{spawn}\ (\lambda().(\!|P|\!)_M) : \mathbf{1}} \qquad (\!|\Delta|\!) \vdash^{q} (\!|Q|\!)_M : \mathbf{1}}{(\!|\Gamma|\!), (\!|\Delta|\!) \vdash^{q} \mathbf{spawn}\ (\lambda().(\!|P|\!)_M); (\!|Q|\!)_M : \mathbf{1}}$$

$$\textsc{T-Halt}\quad \frac{}{\mathbf{0} \vdash \varnothing} \xrightarrow{(\!|\cdot|\!)_M} \frac{}{\varnothing \vdash^{\perp} () : \mathbf{1}}$$

Figure 10: Translation $(\!|\cdot|\!)_M$ preserves typing (T-Link, T-Res, T-Par, and T-Halt).

Case T-Select-Inl, T-Select-Inr, and T-Offer. *See Fig. 14.*

□

Theorem 4.6. *If* $P \vdash \Gamma$, *then* $(\!|\Gamma|\!) \vdash^{\circ} (\!|P|\!)_c$.

Proof. By induction on the derivation of $P \vdash \Gamma$.

Case T-Res. *Immediately, from the induction hypothesis.*

$$\textsc{T-Res}\quad \frac{\Gamma, x : A, y : A^{\perp} \vdash P}{\Gamma \vdash (\nu xy) P} \xrightarrow{(\!|\cdot|\!)_c} \frac{(\!|\Gamma|\!), x : (\!|A|\!), y : (\!|B|\!) \vdash^{\circ} (\!|P|\!)_c}{(\!|\Gamma|\!) \vdash^{\circ} (\nu xy) (\!|P|\!)_c}$$

$$\text{T-Close}\quad \frac{P \vdash \Gamma \qquad o < \mathrm{pr}(\Gamma)}{x[].P \vdash \Gamma, x : \mathbf{1}^o} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\overline{\mathbf{close} : \mathbf{end}_!^o \multimap^{\top,o} \mathbf{1}} \qquad \overline{x : \mathbf{end}_!^o \vdash^{\bot} x : \mathbf{end}_!^o}}{x : \mathbf{end}_!^o \vdash^o \mathbf{close}\ x : \mathbf{1}} \qquad (\!|\Gamma|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}((\!|\Gamma|\!))}{(\!|\Gamma|\!), x : \mathbf{end}_!^o \vdash^{o \sqcup p} \mathbf{close}\ x; (\!|P|\!)_M : \mathbf{1}}$$

$$\text{T-Wait}\quad \frac{P \vdash \Gamma \qquad o < \mathrm{pr}(\Gamma)}{x().P \vdash \Gamma, x : \mathbf{1}^o} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\overline{\mathbf{wait} : \mathbf{end}_?^o \multimap^{\top,o} \mathbf{1}} \qquad \overline{x : \mathbf{end}_?^o \vdash^{\bot} x : \mathbf{end}_?^o}}{x : \mathbf{end}_?^o \vdash^o \mathbf{wait}\ x : \mathbf{1}} \qquad (\!|\Gamma|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}((\!|\Gamma|\!))}{(\!|\Gamma|\!), x : \mathbf{end}_?^o \vdash^{o \sqcup p} \mathbf{wait}\ x; (\!|P|\!)_M : \mathbf{1}}$$

Figure 11: Translation $(\!|\cdot|\!)_M$ preserves typing (T-Close and T-Wait).

$$\text{T-Send}\quad \frac{P \vdash \Gamma, y : A, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x[y].P \vdash \Gamma, x : A \otimes^o B} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\text{(A)}\quad \frac{\overline{\mathbf{new} : \mathbf{1} \multimap (\!|A|\!) \times \overline{(\!|A|\!)}} \qquad \overline{\varnothing \vdash^{\bot} () : \mathbf{1}}}{\varnothing \vdash^{\bot} \mathbf{new}\ () : (\!|A|\!) \times \overline{(\!|A|\!)}}$$

$$\text{(B)}\quad \frac{\dfrac{\overline{\mathbf{send} : \overline{(\!|A|\!)} \times\, !^o\overline{(\!|A|\!)}.(\!|B|\!) \multimap^{\top,o} (\!|B|\!)}}{\dfrac{\overline{z : \overline{(\!|A|\!)} \vdash^{\bot} x : \overline{(\!|A|\!)}} \qquad \overline{x : !^o\overline{(\!|A|\!)}.(\!|B|\!) \vdash^{\bot} x : !^o\overline{(\!|A|\!)}.(\!|B|\!)}}{x : !^o\overline{(\!|A|\!)}.(\!|B|\!), z : \overline{(\!|A|\!)} \vdash^{\bot} (z, x) : \overline{(\!|A|\!)} \times\, !^o\overline{(\!|A|\!)}.(\!|B|\!)}}{x : !^o\overline{(\!|A|\!)}.(\!|B|\!), z : \overline{(\!|A|\!)} \vdash^o \mathbf{send}\ (z, x) : (\!|B|\!)}$$

$$\text{(A)}\quad \frac{\text{(B)} \qquad (\!|\Gamma|\!), y : (\!|A|\!), x : (\!|B|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}((\!|\Gamma|\!), (\!|A|\!), (\!|B|\!))}{(\!|\Gamma|\!), x : !^o\overline{(\!|A|\!)}.(\!|B|\!), y : (\!|A|\!), z : \overline{(\!|A|\!)} \vdash^{o \sqcup p} \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}}$$
$$(\!|\Gamma|\!), x : !^o\overline{(\!|A|\!)}.(\!|B|\!) \vdash^{o \sqcup p} \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in\ let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}$$

Figure 12: Translation $(\!|\cdot|\!)_M$ preserves typing (T-Send).

$$\textsc{T-Recv}\ \frac{P \vdash \Gamma, y : A, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x(y).P \vdash \Gamma, x : A \parr^o B} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{}{\mathbf{recv} : ?^o(\!|A|\!).(\!|B|\!) \multimap^{\top,o} (\!|A|\!) \times (\!|B|\!)} \qquad \dfrac{}{x : ?^o(\!|A|\!).(\!|B|\!) \vdash^{\bot} x : ?^o(\!|A|\!).(\!|B|\!)}}{x : ?^o(\!|A|\!).(\!|B|\!) \vdash^o \mathbf{recv}\ x : (\!|A|\!) \times (\!|B|\!)}$$

$$\frac{(\text{A}) \qquad (\!|\Gamma|\!), y : (\!|A|\!), x : (\!|B|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}((\!|\Gamma|\!), (\!|A|\!), (\!|B|\!))}{(\!|\Gamma|\!), x : ?^o(\!|A|\!).(\!|B|\!), y : (\!|A|\!), z : (\!|A|\!) \vdash^{o \sqcup p} \mathbf{let}\ x = \mathbf{recv} x\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}}$$

Figure 13: Translation $(\!|\cdot|\!)_M$ preserves typing (T-Recv).

$$\textsc{T-Select-Inl}\ \frac{P \vdash \Gamma, x : A \qquad o < \mathrm{pr}(\Gamma)}{x \triangleleft \mathrm{inl}.P \vdash \Gamma, x : A \oplus^o B} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\dfrac{}{\mathbf{select\ inl} : (\!|A|\!) \oplus^o (\!|B|\!) \multimap^{\top,o} (\!|A|\!)} \qquad \dfrac{}{x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^{\bot} x : (\!|A|\!) \oplus^o (\!|B|\!)}}{x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^o \mathbf{select\ inl}\ x : (\!|A|\!)} \qquad \Gamma, x : (\!|A|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}(\Gamma)}{\Gamma, x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^{o \sqcup p} \mathbf{let}\ x = \mathbf{select\ inl}\ x\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}}$$

$$\textsc{T-Select-Inr}\ \frac{P \vdash \Gamma, x : A \qquad o < \mathrm{pr}(\Gamma)}{x \triangleleft \mathrm{inr}.P \vdash \Gamma, x : A \oplus^o B} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{\dfrac{}{\mathbf{select\ inr} : (\!|A|\!) \oplus^o (\!|B|\!) \multimap^{\top,o} (\!|B|\!)} \qquad \dfrac{}{x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^{\bot} x : (\!|A|\!) \oplus^o (\!|B|\!)}}{x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^o \mathbf{select\ inr}\ x : (\!|B|\!)} \qquad \Gamma, x : (\!|B|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}(\Gamma)}{\Gamma, x : (\!|A|\!) \oplus^o (\!|B|\!) \vdash^{o \sqcup p} \mathbf{let}\ x = \mathbf{select\ inr}\ x\ \mathbf{in}\ (\!|P|\!)_M : \mathbf{1}}$$

$$\textsc{T-Offer}\ \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B \qquad o < \mathrm{pr}(\Gamma, A, B)}{x \triangleright \{\mathrm{inl} : P; \mathrm{inr} : Q\} \vdash \Gamma, x : A \mathbin{\&^o} B} \xrightarrow{(\!|\cdot|\!)_M}$$

$$\frac{\dfrac{}{x : (\!|A|\!) \mathbin{\&^o} (\!|B|\!) \vdash^{\bot} x : (\!|A|\!) \mathbin{\&^o} (\!|B|\!)} \qquad (\!|\Gamma|\!), x : (\!|A|\!) \vdash^p (\!|P|\!)_M : \mathbf{1} \qquad (\!|\Gamma|\!), x : (\!|B|\!) \vdash^p (\!|Q|\!)_M : \mathbf{1} \qquad o < \mathrm{pr}((\!|\Gamma|\!), (\!|A|\!), (\!|B|\!))}{(\!|\Gamma|\!), x : (\!|A|\!) \mathbin{\&^o} (\!|B|\!) \vdash^{o \sqcup p} \mathbf{offer}\ x\ \{\mathbf{inl}\ x \mapsto (\!|P|\!)_M; \mathbf{inr}\ x \mapsto (\!|Q|\!)_M\} : \mathbf{1}}$$

Figure 14: Translation $(\!|\cdot|\!)_M$ preserves typing (T-Select-Inl, T-Select-Inr, and T-Offer).

Case T-Par. *Immediately, from the induction hypotheses.*

$$\dfrac{\dfrac{\Gamma \vdash P \qquad \Delta \vdash Q}{\Gamma, \Delta \vdash P \parallel Q}\ \text{T-Par}}{} \xrightarrow{(\!|\cdot|\!)_{\mathcal{C}}} \dfrac{(\!|\Gamma|\!) \vdash^{\circ} (\!|P|\!)_{\mathcal{C}} \qquad (\!|\Delta|\!) \vdash^{\circ} (\!|Q|\!)_{\mathcal{C}}}{(\!|\Gamma|\!), (\!|\Delta|\!) \vdash^{\circ} (\!|P|\!)_{\mathcal{C}} \parallel (\!|Q|\!)_{\mathcal{C}}}$$

Case $*$. *By Lemma 4.5*

$$\Gamma \vdash P \xrightarrow{(\!|\cdot|\!)_{\mathcal{C}}} \dfrac{(\!|\Gamma|\!) \vdash^{p} (\!|P|\!)_{M} : \mathbf{1}}{(\!|\Gamma|\!) \vdash^{\circ} \circ\, (\!|P|\!)_{M}}$$

□

Theorem 4.7. *If* $P \vdash \Gamma$ *and* $(\!|P|\!)_{\mathcal{C}} \longrightarrow_{\mathcal{C}} \mathcal{C}$*, there exists a* Q *such that* $P \Longrightarrow^{+} Q$ *and* $\mathcal{C} \longrightarrow_{\mathcal{C}}^{\star} (\!|Q|\!)_{\mathcal{C}}$

Proof. By induction on the derivation of $(\!|P|\!)_{\mathcal{C}} \longrightarrow_{\mathcal{C}} \mathcal{C}$. We omit the cases which cannot occur as their left-hand side term forms are not in the image of the translation function, *i.e.*, E-New, E-Spawn, and E-LiftM.

Case E-Link.

$$(\nu x x')(\mathcal{F}[\mathbf{link}\ (w, x)] \parallel \mathcal{C}) \longrightarrow_{\mathcal{C}} \mathcal{F}[()] \parallel \mathcal{C}\{w/x'\}$$

The source for $\mathbf{link}\ (w, x)$ must *be* $w \leftrightarrow x$*. None of the translation rules introduce an evaluation context around the recursive call, hence* $\mathcal{F}$ *must be the empty context. Let* P *be the source term for* $\mathcal{C}$, i.e., $(\!|P|\!)_{\mathcal{C}} = \mathcal{C}$*. Hence, we have:*

$$\begin{array}{ccc}
(\nu x x')(w \leftrightarrow x \parallel P) & \xrightarrow{\Longrightarrow} & P\{w/x'\} \\
\downarrow{\scriptstyle (\!|\cdot|\!)_{\mathcal{C}}} & & \\
(\nu x x')(\circ\, \mathbf{link}\ (w, x) \parallel (\!|P|\!)_{\mathcal{C}}) & & \downarrow{\scriptstyle (\!|\cdot|\!)_{\mathcal{C}}} \\
\downarrow{\scriptstyle \longrightarrow_{\mathcal{C}}^{+}} & & \\
(\!|P|\!)_{\mathcal{C}}\{w/x'\} & \xrightarrow{=} & (\!|P\{w/x'\}|\!)_{\mathcal{C}}
\end{array}$$

Case E-Send.

$$(\nu x x')(\mathcal{F}[\mathbf{send}\ (V, x)] \parallel \mathcal{F}'[\mathbf{recv}\ x']) \longrightarrow_{\mathcal{C}} (\nu x x')(\mathcal{F}[x] \parallel \mathcal{F}'[(V, x')])$$

There are three possible sources for **send** *and* **recv***:* $x[y].P$ *and* $x'(y').Q$*;* $x \triangleleft inl.P$ *and* $x' \triangleright \{inl : Q; inr : R\}$*; or* $x \triangleleft inr.P$ *and* $x' \triangleright \{inl : Q; inr : R\}$*.*

Subcase $x[y].P$ *and* $x'(y').Q$*. None of the translation rules introduce an evaluation context around the recursive call, hence* $\mathcal{F}$ *must be* $\circ\, \mathbf{let}\ x = \square\ \mathbf{in}\ (\!|P|\!)_{M}$*. Similarly,* $\mathcal{F}'$ *must be* $\circ\, \mathbf{let}\ (y', x') = \square\ \mathbf{in}\ (\!|Q|\!)_{M}$*. The value* V *must be an endpoint* y*, bound by the name restriction* $(\nu y y')$ *introduced by the translation. Hence, we have:*

$$\begin{array}{ccc}
(\nu x x')(x[y].P \parallel x'(y').Q) & \xrightarrow{\Longrightarrow} & (\nu x x')(\nu y y')(P \parallel Q) \\
\downarrow{\scriptstyle (\!|\cdot|\!)_{\mathcal{C}}} & & \\
(\nu x x')(\nu y y')\left(\begin{array}{l} \circ\, \mathbf{let}\ x = \mathbf{send}\ (y, x)\ \mathbf{in}\ (\!|P|\!)_{M} \parallel \\ \circ\, \mathbf{let}\ (y', x') = \mathbf{recv}\ x'\ \mathbf{in}\ (\!|Q|\!)_{M} \end{array}\right) & & \downarrow{\scriptstyle (\!|\cdot|\!)_{\mathcal{C}}} \\
\downarrow{\scriptstyle \equiv \longrightarrow_{\mathcal{C}}^{+}} & & \\
(\nu x x')(\nu y y')(\circ\, (\!|P|\!)_{M} \parallel \circ\, (\!|Q|\!)_{M}) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_{\mathcal{C}}^{\star}} & (\nu x x')(\nu y y')((\!|P|\!)_{\mathcal{C}} \parallel (\!|Q|\!)_{\mathcal{C}})
\end{array}$$

Subcase $x \triangleleft inl.P$ *and* $x' \triangleright \{inl : Q; inr : R\}$. *None of the translation rules introduce an evaluation context around the recursive call, hence* $\mathcal{F}$ *must be*

$$\circ \, \mathbf{let}\ x = \mathbf{close}\ \Box; y\ \mathbf{in}\ (\!|P|\!)_M.$$

Similarly, $\mathcal{F}'$ *must be*

$$\circ \, \mathbf{let}\ (y', x') = \Box\ \mathbf{in}\ \mathbf{wait}\ x'; \mathbf{case}\ y'\ \left\{\mathbf{inl}\ y' \mapsto (\!|Q|\!)_M;\ \mathbf{inr}\ y' \mapsto (\!|R|\!)_M\right\}.$$

Hence, we have:

$$\begin{array}{ccc}
(\nu x x')(x \triangleleft inl.P \parallel x \triangleright \{inl : Q; inr : R\}) & \Longrightarrow & (\nu x x')(P \parallel Q) \\
\Big\downarrow (\!|\cdot|\!)_M & & \\
(\nu x x')\left(\begin{array}{l} \circ\ \mathbf{let}\ x = \mathbf{select}\ \mathbf{inl}\ x\ \mathbf{in}\ (\!|P|\!)_M \parallel \\ \circ\ \mathbf{offer}\ x'\ \{\mathbf{inl}\ x' \mapsto (\!|Q|\!)_M; \mathbf{inr}\ x' \mapsto (\!|R|\!)_M\} \end{array}\right) & & \Big\downarrow (\!|\cdot|\!)_C \\
\Big\downarrow \longrightarrow_C^+ & & \\
(\nu x x')(\circ\ (\!|P|\!)_M \parallel \circ\ (\!|Q|\!)_M) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_C^\star} & (\nu x x')((\!|P|\!)_C \parallel (\!|Q|\!)_C)
\end{array}$$

Subcase $x \triangleleft inr.P$ *and* $x' \triangleright \{inl : Q; inr : R\}$. *None of the translation rules introduce an evaluation context around the recursive call, hence* $\mathcal{F}$ *must be*

$$\circ \, \mathbf{let}\ x = \mathbf{close}\ \Box; y\ \mathbf{in}\ (\!|P|\!)_M.$$

Similarly, $\mathcal{F}'$ *must be*

$$\circ \, \mathbf{let}\ (y', x') = \Box\ \mathbf{in}\ \mathbf{wait}\ x'; \mathbf{case}\ y'\ \left\{\mathbf{inl}\ y' \mapsto (\!|Q|\!)_M;\ \mathbf{inr}\ y' \mapsto (\!|R|\!)_M\right\}.$$

Hence, we have:

$$\begin{array}{ccc}
(\nu x x')(x \triangleleft inr.P \parallel x \triangleright \{inl : Q; inr : R\}) & \Longrightarrow & (\nu x x')(P \parallel Q) \\
\Big\downarrow (\!|\cdot|\!)_M & & \\
(\nu x x')\left(\begin{array}{l} \circ\ \mathbf{let}\ x = \mathbf{select}\ \mathbf{inr}\ x\ \mathbf{in}\ (\!|P|\!)_M \parallel \\ \circ\ \mathbf{offer}\ x'\ \{\mathbf{inl}\ x' \mapsto (\!|Q|\!)_M; \mathbf{inr}\ x' \mapsto (\!|R|\!)_M\} \end{array}\right) & & \Big\downarrow (\!|\cdot|\!)_C \\
\Big\downarrow \longrightarrow_C^+ & & \\
(\nu x x')(\circ\ (\!|P|\!)_M \parallel \circ\ (\!|R|\!)_M) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_C^\star} & (\nu x x')((\!|P|\!)_C \parallel (\!|R|\!)_C)
\end{array}$$

Case E-Close.

$$(\nu x x')(\mathcal{F}[\mathbf{wait}\ x] \parallel \mathcal{F}'[\mathbf{close}\ x']) \longrightarrow_C \mathcal{F}[()] \parallel \mathcal{F}'[()]$$

The source for **wait** *and* **close** must *be* $x().P$ *and* $x'[].Q$.

(The translation for $x \triangleright \{inl : P; inr : Q\}$ *also introduces a* **wait**, *but it is blocked on another communication, and hence cannot be the first communication on a translated term. The translations for* $x \triangleleft inl.P$ *and* $x \triangleleft inr.P$ *also introduce a* **close**, *but these are similarly blocked.)*

None of the translation rules introduce an evaluation context around the recursive call, hence $\mathcal{F}$ *must be* $\Box; (\!|P|\!)_M$*. Similarly,* $\mathcal{F}'$ *must be* $\Box; (\!|Q|\!)_M$*. Hence, we have:*

$$\begin{array}{ccc}
(\nu x x')(x[].P \parallel x'().Q) & \xrightarrow{\quad\Longrightarrow\quad} & P \parallel Q \\
\Big\downarrow (\!|\cdot|\!)_M & & \\
(\nu x x')(\circ\, \mathbf{close}\ x; (\!|P|\!)_M \parallel \circ\, \mathbf{wait}\ x'; (\!|Q|\!)_M) & & \Big\downarrow (\!|\cdot|\!)_C \\
\Big\downarrow \longrightarrow_C^+ & & \\
\circ\, (\!|P|\!)_M \parallel \circ\, (\!|Q|\!)_M & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_C^\star} & (\!|P|\!)_C \parallel (\!|Q|\!)_C
\end{array}$$

Case E-LiftC. *By the induction hypothesis and* E-LiftC.

Case E-LiftSC. *By the induction hypothesis,* E-LiftSC*, and Lemma 4.9.* □

Lemma 4.8. *For any* P*, either:*

- $\circ\, (\!|P|\!)_M = (\!|P|\!)_C$*; or*
- $\circ\, (\!|P|\!)_M \longrightarrow_C^+ (\!|P|\!)_C$*, and for any* $\mathcal{C}$*, if* $\circ\, (\!|P|\!)_M \longrightarrow_C \mathcal{C}$*, then* $\mathcal{C} \longrightarrow_C^\star (\!|P|\!)_C$*.*

Proof. By induction on the structure of P.

Case $(\nu x y)P$*. We have:*

$$\begin{array}{rcl}
(\!|(\nu x y)P|\!)_M & = & \circ\, \mathbf{let}\ (x, y) = \mathbf{new}\ ()\ \mathbf{in}\ (\!|P|\!)_M \\
& \longrightarrow_C^+ & (\nu x y)(\circ\, (\!|P|\!)_M) \\
& \longrightarrow_C^\star & (\nu x y)(\!|P|\!)_C \\
& = & (\!|(\nu x y)P|\!)_C
\end{array}$$

Case $P \parallel Q$.

$$\begin{array}{rcl}
(\!|P \parallel Q|\!)_M & = & \circ\, \mathbf{spawn}\ (\lambda().(\!|P|\!)_M); (\!|Q|\!)_M \\
& \longrightarrow_C^+ & \circ\, (\!|P|\!)_M \parallel \circ\, (\!|Q|\!)_M \\
& \longrightarrow_C^\star & (\!|P|\!)_C \parallel (\!|Q|\!)_C \\
& = & (\!|P \parallel Q|\!)_C
\end{array}$$

Case $x[y].P$.

$$\begin{array}{rcl}
(\!|x[y].P|\!)_M & = & \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M \\
& \longrightarrow_C^+ & (\nu y z)(\circ\, \mathbf{let}\ x = \mathbf{send}\ (z, x)\ \mathbf{in}\ (\!|P|\!)_M) \\
& = & (\!|x[y].P|\!)_C
\end{array}$$

Case $x \triangleleft \mathrm{inl}.P$.

$$\begin{array}{rcl}
(\!|x[y].P|\!)_M & = & \mathbf{let}\ x = \mathbf{select}\ \mathbf{inl}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
& \triangleq & \mathbf{let}\ x = \mathbf{let}\ (y, z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M \\
& \longrightarrow_C^+ & (\nu y z)(\circ\, \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inl}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
& = & (\!|x[y].P|\!)_C
\end{array}$$

Case $x \triangleleft \mathrm{inr}.P$.

$$\begin{array}{rcl}
(\!|x[y].P|\!)_M & = & \mathbf{let}\ x = \mathbf{select}\ \mathbf{inr}\ x\ \mathbf{in}\ (\!|P|\!)_M \\
& \triangleq & \mathbf{let}\ x = \mathbf{let}\ (y,z) = \mathbf{new}\ ()\ \mathbf{in}\ \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M \\
& \longrightarrow_{\mathcal{C}}^{+} & (\nu yz)(\circ\ \mathbf{let}\ x = \mathbf{close}\ (\mathbf{send}\ (\mathbf{inr}\ y, x)); z\ \mathbf{in}\ (\!|P|\!)_M) \\
& = & (\!|x[y].P|\!)_C
\end{array}$$

Case $*$. *In all other cases,* $\circ\ (\!|P|\!)_M = (\!|P|\!)_C$. □

Lemma 4.9. *If* $P \vdash \Gamma$ *and* $P \equiv Q$, *then* $(\!|P|\!)_C \equiv (\!|Q|\!)_C$.

Proof. Every axiom of the structural congruence in PCP maps directly to the axiom of the same name in PGV. □

Theorem 4.10.
If $P \vdash \Gamma$ *and* $P \Longrightarrow Q$, *then* $(\!|P|\!)_C \longrightarrow_{\mathcal{C}}^{+} (\!|Q|\!)_C$.

Proof. By induction on the derivation of $P \Longrightarrow Q$.

Case E-Link.

$$\begin{array}{ccc}
(\nu xx')(w{\leftrightarrow}x \parallel P) & \xrightarrow{\Longrightarrow} & P\{w/x'\} \\
\downarrow{\scriptstyle (\!|\cdot|\!)_C} & & \\
(\nu xx')(\circ\ \mathbf{link}\ (w,x) \parallel (\!|P|\!)_C) & & \downarrow{\scriptstyle (\!|\cdot|\!)_C} \\
\downarrow{\scriptstyle \longrightarrow_{\mathcal{C}}^{+}} & & \\
(\!|P|\!)_C\{w/x'\} & \xrightarrow{=} & (\!|P\{w/x'\}|\!)_C
\end{array}$$

Case E-Send.

$$\begin{array}{ccc}
(\nu xx')(x[y].P \parallel x'(y').Q) & \xrightarrow{\Longrightarrow} & (\nu xx')(\nu yy')(P \parallel Q) \\
\downarrow{\scriptstyle (\!|\cdot|\!)_M} & & \\
(\nu xx')\left(\begin{array}{l}\circ\left(\begin{array}{l}\mathbf{let}\ (y,y') = \mathbf{new}\ ()\ \mathbf{in} \\ \mathbf{let}\ x = \mathbf{send}\ (y,x)\ \mathbf{in}\ (\!|P|\!)_M\end{array}\right) \parallel \\ \circ\ \ \mathbf{let}\ (y',x') = \mathbf{recv}\ x'\ \mathbf{in}\ (\!|Q|\!)_M\end{array}\right) & & \downarrow{\scriptstyle (\!|\cdot|\!)_C} \\
\downarrow{\scriptstyle \longrightarrow_{\mathcal{C}}^{+}} & & \\
(\nu xx')(\nu yy')(\circ\ (\!|P|\!)_M \parallel \circ\ (\!|Q|\!)_M) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_{\mathcal{C}}^{\star}} & (\nu xx')(\nu yy')((\!|P|\!)_C \parallel (\!|Q|\!)_C)
\end{array}$$

Case E-Close.

$$\begin{array}{ccc}
(\nu xx')(x[].P \parallel x'().Q) & \xrightarrow{\Longrightarrow} & P \parallel Q \\
\downarrow{\scriptstyle (\!|\cdot|\!)_M} & & \\
(\nu xx')(\circ\ \mathbf{close}\ x; (\!|P|\!)_M \parallel \circ\ \mathbf{wait}\ x'; (\!|Q|\!)_M) & & \downarrow{\scriptstyle (\!|\cdot|\!)_C} \\
\downarrow{\scriptstyle \longrightarrow_{\mathcal{C}}^{+}} & & \\
\circ\ (\!|P|\!)_M \parallel \circ\ (\!|Q|\!)_M & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_{\mathcal{C}}^{\star}} & (\!|P|\!)_C \parallel (\!|Q|\!)_C
\end{array}$$

Case E-Select-Inl.

$$\begin{array}{ccc}
(\nu x x')(x \triangleleft inl.P \parallel x \triangleright \{inl : Q; inr : R\}) & \xrightarrow{\quad\Longrightarrow\quad} & (\nu x x')(P \parallel Q) \\
\downarrow (\!|\cdot|\!)_M & & \\
(\nu x x')\left(\begin{array}{l} \circ\ \mathbf{let}\ x = \mathbf{select\ inl}\ x\ \mathbf{in}\ (\!|P|\!)_M \parallel \\ \circ\ \mathbf{offer}\ x'\ \{\mathbf{inl}\ x' \mapsto (\!|Q|\!)_M; \mathbf{inr}\ x' \mapsto (\!|R|\!)_M\}\end{array}\right) & & \downarrow (\!|\cdot|\!)_C \\
\downarrow \longrightarrow_C^+ & & \\
(\nu x x')(\circ\ (\!|P|\!)_M \parallel \circ\ (\!|Q|\!)_M) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_C^\star} & (\nu x x')((\!|P|\!)_C \parallel (\!|Q|\!)_C)
\end{array}$$

Case E-Select-Inr.

$$\begin{array}{ccc}
(\nu x x')(x \triangleleft inr.P \parallel x \triangleright \{inl : Q; inr : R\}) & \xrightarrow{\quad\Longrightarrow\quad} & (\nu x x')(P \parallel R) \\
\downarrow (\!|\cdot|\!)_M & & \\
(\nu x x')\left(\begin{array}{l} \circ\ \mathbf{let}\ x = \mathbf{select\ inr}\ x\ \mathbf{in}\ (\!|P|\!)_M \parallel \\ \circ\ \mathbf{offer}\ x'\ \{\mathbf{inl}\ x' \mapsto (\!|Q|\!)_M; \mathbf{inr}\ x' \mapsto (\!|R|\!)_M\}\end{array}\right) & & \downarrow (\!|\cdot|\!)_C \\
\downarrow \longrightarrow_C^+ & & \\
(\nu x x')(\circ\ (\!|P|\!)_M \parallel \circ\ (\!|R|\!)_M) & \xrightarrow[\textit{(by Lemma 4.8)}]{\longrightarrow_C^\star} & (\nu x x')((\!|P|\!)_C \parallel (\!|R|\!)_C)
\end{array}$$

Case E-LiftRes. *By the induction hypothesis and* E-LiftC.

Case E-LiftPar. *By the induction hypotheses and* E-LiftC.

Case E-LiftSC. *By the induction hypothesis,* E-LiftSC*, and Lemma 4.9.* □

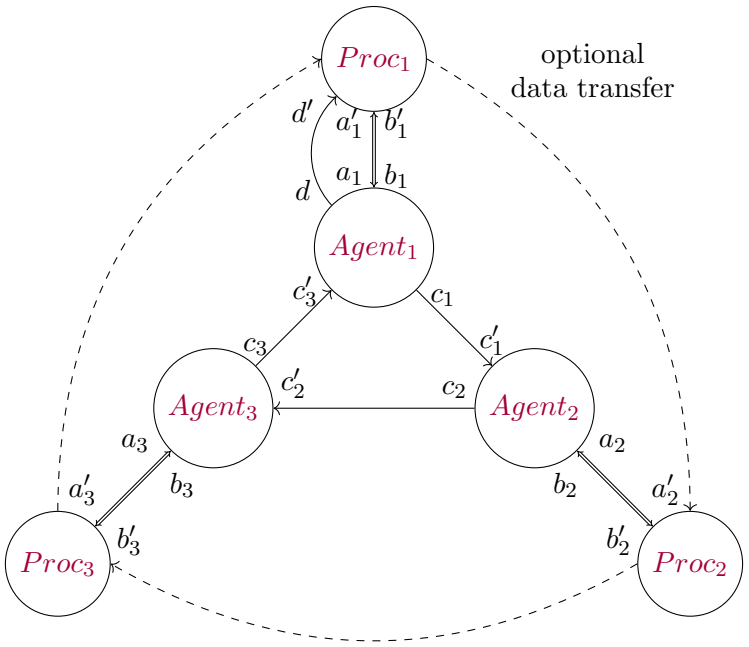
5. Milner's Cyclic Scheduler

As an example of a deadlock-free cyclic process, Dardha and Gay [DG18b] introduce an implementation of Milner's cyclic scheduler [Mil89] in Priority CP. We reproduce that scheduler here, and show its translation to Priority GV.

Example 5.1. A set of processes $Proc_i$, $1 \leq i \leq n$, is scheduled to perform some tasks in cyclic order, starting with $Proc_1$, ending with $Proc_n$, and notifying $Proc_1$ when all processes have finished.

Our scheduler $Sched$ consists of set of agents $Agent_i$, $1 \leq i \leq n$, each representing their respective process. Each process $Proc_i$ waits for the signal to start their task on a'_i, and signals completion on b'_i. Each agent signals their process to start on a_i, waits for their process to finish on b_i, and then signals for the next agent to continue on c_i. The agent $Agent_1$ initiates, then waits for every other process to finish, and signals $Proc_1$ on d. Every other agent $Agent_i$, $2 \leq i \leq n$ waits on c'_{i-1} for the signal to start. Each of the channels in the scheduler is of a terminated type, and is merely used to synchronise.

Below is a diagram of our scheduler instantiated with three processes:



We implement the scheduler as follows, using $\prod_I P_i$ to denote the parallel composition of the processes P_i, $i \in I$, and $P[Q]$ to denote the plugging of Q in the one-hole process-context P. The process-contexts P_i represent the computations performed by each process $Proc_i$. The process-contexts Q_i represent any post-processing, and any possible data transfer from $Proc_i$ to $Proc_{i+1}$. Finally, Q_1 should contain $d'()$.

$$
\begin{array}{lcl}
Sched & \triangleq & (\nu a_1 a'_1) \ldots (\nu a_n a'_n)(\nu b_1 b'_1) \ldots (\nu b_n b'_n)(\nu c_1 c'_1) \ldots (\nu c_n c'_n)(\nu d d') \\
 & & (Proc_1 \parallel Agent_1 \parallel \textstyle\prod_{2 \leq i \leq n} (Proc_i \parallel c'_{i-1}().Agent_i)) \\
Agent_1 & \triangleq & a_i[].b_i().c_i[].c'_n().d[].\mathbf{0} \\
Agent_i & \triangleq & a_i[].b_i().c_i[].\mathbf{0} \\
Proc_i & \triangleq & a'_i().P_i[b'_i[].Q_i]
\end{array}
$$

Example 5.2. The PGV scheduler has exactly the same behaviour as the PCP version in Example 5.1. It is implemented as follows, using $\prod_I \mathcal{C}_i$ to denote the parallel composition of the processes $\mathcal{C}_i$, $i \in I$, and $M[N]$ to denote the plugging of N in the one-hole term-context M. For simplicity, we let **sched** be a configuration. The terms M_i represent the computations performed by each process $\mathbf{proc}_i$. The terms N_i represent any post-processing, and any possible data transfer from $\mathbf{proc}_i$ to $\mathbf{proc}_{i+1}$. Finally, N_1 should contain $\mathbf{wait}\ d'$.

$$
\begin{array}{lcl}
\mathbf{sched} & \triangleq & (\nu a_1 a'_1) \ldots (\nu a_n a'_n)(\nu b_1 b'_1) \ldots (\nu b_n b'_n)(\nu c_1 c'_1) \ldots (\nu c_n c'_n)(\nu d d') \\
 & & (\quad \phi\ \mathbf{proc}_1 \parallel \circ\ \mathbf{agent}_1; \mathbf{wait}\ c'_n; \mathbf{close}\ d \\
 & & \parallel\ \textstyle\prod_{2 \leq i \leq n} (\circ\ \mathbf{proc}_i \parallel \circ\ \mathbf{wait}\ c'_{i-1}; \mathbf{agent}_i) \quad) \\
\mathbf{agent}_i & \triangleq & \mathbf{close}\ a_i; \mathbf{wait}\ b_i; \mathbf{close}\ c_i \\
\mathbf{proc}_i & \triangleq & \mathbf{wait}\ a'_i; M_i[\mathbf{close}\ b'_i; N_i]
\end{array}
$$

If $(\!|P_i|\!)_M = M_i$ and $(\!|Q_i|\!)_M = N_i$, then the translation of $Sched$ (Example 5.1), $(\!|Sched|\!)_{\mathcal{C}}$, is exactly **sched** (Example 5.2).

6. Related Work and Conclusion

Deadlock freedom and progress. Deadlock freedom and progress are well studied properties in the π-calculus. For the "standard" typed π-calculus—types for channels used in input and output, an important line of work stems from Kobayashi's approach to deadlock freedom [Kob98], where priorities are values from an abstract poset. Kobayashi [Kob06] simplifies the abstract poset to pairs of naturals, called *obligations* and *capabilities*. Padovani simplifies these further to a single natural, called a *priority* [Pad14], and adapts obligations/capabilities to session types [Pad13]. Later work by Kobayashi and co-authors [GKL14, KL17] address deadlock detection for a value-passing CCS (a predecessor of the π-calculus) where the number of nodes in a network is arbitrary, namely modelling unbounded networks. The authors define a sound inference algorithm for their type system, which guarantees deadlock freedom for these more complex kinds of communication networks. This type system is more expressive than previous work by Kobayashi.

For the session-typed π-calculus, an important line of work stems from Dezani and co-authors. In their work, Dezani *et al.* [DCMYD06] guarantee progress by allowing only one active session at a time. As sessions do not interleave, consequently they do not deadlock. Later on, Dezani *et al.* [DCdY07] introduce a partial order on channels, similar to Kobayashi [Kob98] and produce a type system for progress.

Carbone and Debois [CD10] define progress for session-typed π-calculus in terms of a *catalyser*, which provides the missing counterpart to a process. Intuitively, either the process is deadlock-free in which case the catalyser is structurally congruent to the inaction process, or if the process is "stuck" for *e.g.*, an input process $x(y).\mathbf{0}$ (using CP syntax) then the catalyser simply provides the missing output required for communication. Carbone *et al.* [CDM14] use catalysers to show that progress is a compositional form of livelock freedom and can be lifted to session types via the encoding of session types to linear types [Kob07, DGS12, Dar14, DGS17]. In this work, Carbone *et al.* [CDM14] sistematise and compare different notions of liveness properties: progress, deadlock freedom and livelock freedom. Their technique of using the encoding of session types combined with Kobayashi's type systems for deadlock and livelock freedom allows for a more flexible deadlock/livelock detection. It is worth noting that livelock freedom is a stronger property than deadlock freedom in the presence of recursion, as the former also discards useless divergent processes; for finite processes the two properties coincide.

Vieira and Vasconcelos [VV13] use single priorities and an abstract partial order to guarantee deadlock freedom in a binary session-typed π-calculus and building on conversation types, which is akin to session types.

While our work focuses on binary session types, it is worth discussing related work on Multiparty Session Types (MPST), which describe communication among multiple agents in a distributed setting. The line of work on MPST starts with Honda *et al.* [HYC08], which guarantees deadlock freedom within a single session by design, but the property does not hold for session interleaving. Bettini *et al.* [BCD$^+$08] follow a technique similar to Kobayashi's for MPST. The main difference with our work is that we associate priorities with communication actions, where Bettini *et al.* [BCD$^+$08] associate them with channels. Coppo *et al.* [CDPY13, CDYP16] present a deterministic, sound and complete, and compositional type inference algorithm for an *interaction type system*, which is used to guarantee global progress for processes in a calculus based on asynchronous as well as dynamically interleaved

and interfered multiparty sessions. The interaction type system allows to infer *causalities*—much in the line of Kobayashi's priorities —of channels, thus guaranteeing that session-typed processes do not get stuck at intermediate stages of their sessions. Carbone and Montesi [CM13] combine MPST with choreographic programming and obtain a formalism that satisfies deadlock freedom. In the same vein as MPST, choreographic programming specifies communication among all participants in a distributed system. While MPST target mainly protocol descriptions, choreographies have mainly targetted implementations and programming languages as they are suitable for describing concrete system implementations. Deniélou and Yoshida [DY13] introduce *multiparty compatibility*, which generalises the notion of duality in binary session types. They synthesise safe and deadlock-free *global types*–specifying communication among all involved participants, from *local session types*–specifying communication from the viewpoint of one participant. To do so, they leverage Labelled Transition Systems (LTSs) and communicating automata. Castellani *et al.* [CDGH20] guarantee livelock freedom, a stronger property than deadlock freedom, for MPST with internal delegation, where participants in the same session are allowed to delegate tasks to each other, and internal delegation is captured by the global type. Scalas and Yoshida [SY19] provide a revision of the foundations for MPST, and offer a less complicated and more general theory, by removing duality/consistency. The type systems is parametric and type checking is decidable, but allows for a novel integration of model checking techniques. More protocols and processes can be typed and are guaranteed to be free of deadlocks.

Neubauer and Thiemann [NT04] and Vasconcelos *et al.* [VRG04, VGR06] introduce the first functional language with session types. Such works did not guarantee deadlock freedom until GV [LM15, Wad14]. Toninho *et al.* [TCP12] present a translation of simply-typed λ-calculus into session-typed π-calculus, but their focus is not on deadlock freedom. Fowler *et al.* [FKD$^+$21] present Hypersequent GV (HGV), which is a variation of GV that uses hyper-environments, much in the same line as Hypersequent CP, and enjoys deadlock freedom, confluence, and strong normalisation.

Ties with logic. The correspondence between logic and types lays the foundation for functional programming [Wad15]. Since its inception by Girard [Gir87], linear logic has been a candidate for a foundational correspondence for concurrent programs. A correspondence with linear π-calculus was established early on by Abramsky [Abr94] and Bellin and Scott [BS94]. Many years later, several correspondences between linear logic and the π-calculus with binary session types were proposed. Caires and Pfenning [CP10] propose a correspondence with dual intuitionistic linear logic, while Wadler [Wad12] proposes a correspondence with classical linear logic. Both works guarantee deadlock freedom as a consequence of adopting cut elimination. Building on a previous work [CP10], Toninho *et al.* [TCP13] present a Curry-Howard correspondence between session types and linear logic for functional language via linear contextual monads, which are first-class values, thus giving rise to a higher-order session-typed language. In addition to the more standard results, the authors also prove a global progress theorem. Qian *et al.* [QKB21] extend Classical Linear Logic with co-exponentials, which allows to model servers receiving requests from an arbitrary set of clients, yielding an extension to the Curry-Howard correspondence between logic and session typed processes. Dardha and Gay [DG18b] define Priority CP by integrating Kobayashi and Padovani's work on priorities [Kob06, Pad14] with CP, which as described in the introduction, weakens its ties to linear logic in exchange for expressivity. However, they show how PCP can be also viewed as a an extension of linear logic, which they call Priority

Linear Logic (PLL), and uses mix and cycle rules as opposed to the cut rule. Dardha and Pérez [DP18, DP15, DP22] compare priorities à la Kobayashi with tree restrictions à la CP, and show that the latter is a subsystem of the former. In addition, they give a detailed account of comparing several type systems for deadlock freedom spanning across session types, linear logic, and linear types. Carbone *et al.* [CMSY15, CLM$^+$16] give a logical view of MPST with a generalised duality. Caires and Pérez [CP16] give a presentation of MPST in terms of binary session types and the use of a *medium process* which guarantee protocol fidelity and deadlock freedom. Their binary session types are rooted in linear logic. Ciobanu and Horne [CH15] give the first Curry-Howard correspondence between MPST and BV [Gug07], a conservative extension of linear logic with a non-commutative operator for sequencing. Horne [Hor20] give a system for subtyping and multiparty compatibility where compatible processes are race free and deadlock free using a Curry-Howard correspondence, similar to the approach in [CH15]. Balzer *et al.* [BP17] introduce sharing at the cost of deadlock freedom, which they restore using *worlds*, an approach similar to priorities [BTP19]. Staying on sharing, Rocha and Caires [RC21] introduce an imperative feature, that of shared mutable states into a functional language based on Curry-Howard correspondence with linear logic. Their type system is thus able to capture programs which were not possible in previous works. The authors prove extensive technical results, including session fidelity, progress, confluence and normalisation. Lastly, Jacobs *et al.* [JBK22] present a novel technique to guarantee deadlock freedom based on the notion of connectivity graph, which is an abstract representation of the topology of concurrent systems, and separation logic used to substructurally treat connectivity graph edges and labels.

Conclusion and Future Work. We answered our research question by presenting Priority GV, a session-typed functional language which allows cyclic communication structures and uses priorities to ensure deadlock freedom. We showed its relation to Priority CP [DG18b] via an operational correspondence.

Our formalism so far only captures the core of GV. In future work, we plan to explore recursion, following Lindley and Morris [LM16] and Padovani and Novara [PN15], and sharing, following Balzer and Pfenning [BP17] or Voinea *et al.* [VDG19].

Acknowledgements. The authors would like to thank the anonymous reviewers for their detailed feedback, which helped produce a more complete and polished work. Also, the authors would like to thank Simon Fowler, April Gonçalves, and Philip Wadler for their comments on the manuscript.

References

[Abr94] Samson Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, 1994. doi:10.1016/0304-3975(94)00103-0.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. 1996. URL: https://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/ECS-LFCS-96-347.pdf.

[BBN$^+$18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. of POPL*, 2:1–29, 2018. doi:10.1145/3158093.

[BCD+08] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *Proc. of CONCUR*, volume 5201 of *Lect. Notes Comput. Sci.*, pages 418–433. Springer, 2008. doi:10.1007/978-3-540-85361-9_33.

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, 2017. doi:10.1145/3110281.

[BS94] Gianluigi Bellin and Philip J. Scott. On the π-calculus and linear logic. *Theor. Comput. Sci.*, 135(1):11–65, 1994. doi:10.1016/0304-3975(94)00104-9.

[BTP19] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *Proc. of ESOP*, volume 11423 of *Lect. Notes Comput. Sci.*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.

[CD10] Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *Proc. of ICE*, volume 38 of *Electron. Proc. in Theor. Comput. Sci.*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

[CDGH20] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. Global types with internal delegation. *Theor. Comput. Sci.*, 807:128–153, 2020. doi:10.1016/j.tcs.2019.09.027.

[CDM14] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *Proc. of COORDINATION*, volume 8459 of *Lect. Notes Comput. Sci.*, pages 49–64. Springer, 2014. doi:10.1007/978-3-662-43376-8_4.

[CDPY13] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of global progress properties for dynamically interleaved multiparty sessions. In *Proc. of COORDINATION*, volume 7890 of *Lect. Notes Comput. Sci.*, pages 45–59. Springer, 2013. doi:10.1007/978-3-642-38493-6_4.

[CDYP16] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

[CH15] Gabriel Ciobanu and Ross Horne. Behavioural analysis of sessions using the calculus of structures. In *Proc. of PSI*, volume 9609 of *Lect. Notes Comput. Sci.*, pages 91–106. Springer, 2015. doi:10.1007/978-3-319-41579-6_8.

[CLM+16] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *Proc. of CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CONCUR.2016.33.

[CM13] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proc. of POPL*, pages 263–274, 2013. doi:10.1145/2480359.2429101.

[CMS18] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distrib. Comput.*, 31(1):51–67, 2018. doi:10.1007/978-3-662-44584-6_5.

[CMSY15] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. In *Proc. of CONCUR*, volume 42 of *LIPIcs*, pages 412–426. Leibniz-Zentrum für Informatik, 2015. doi:10.1007/s00236-016-0285-y.

[CP10] Luìs Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. of CONCUR*, volume 6269 of *Lect. Notes Comput. Sci.*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.

[CP16] Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *Proc. of FORTE*, volume 9688 of *Lect. Notes Comput. Sci.*, pages 74–95. Springer, 2016. doi:10.1007/978-3-319-39570-8_6.

[Dar14] Ornela Dardha. Recursive session types revisited. In *Proc. of BEAT*, volume 162 of *Electron. Proc. in Theor. Comput. Sci.*, pages 27–34, 2014. doi:10.4204/EPTCS.162.4.

[DCdY07] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *Proc. of TGC*, volume 4912 of *Lect. Notes Comput. Sci.*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *Proc. of ECOOP*, volume 4067 of *Lect. Notes Comput. Sci.*, pages 328–352. Springer, 2006. doi:10.1007/11785477_20.

[DG18a] Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *Proc. of FoSSaCS*, volume 10803 of *Lect. Notes Comput. Sci.*, pages 91–109. Springer, 2018. doi:10.1007/978-3-319-89366-2_5.

[DG18b] Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session typed processes. Extended version of [DG18a]. Available at http://www.dcs.gla.ac.uk/~ornela/publications/DG18-Extended.pdf, 2018.

[DGS12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Proc. of PPDP*, pages 139–150. ACM, 2012. doi:10.1145/2370776.2370794.

[DGS17] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

[DP15] Ornela Dardha and Jorge A. Pérez. Comparing deadlock-free session typed processes. In *Proc. of EXPRESS/SOS*, volume 190 of *Electron. Proc. in Theor. Comput. Sci.*, pages 1–15, 2015. doi:10.4204/EPTCS.190.1.

[DP18] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock-freedom. *CoRR*, abs/1810.00635, 2018. URL: http://arxiv.org/abs/1810.00635, arXiv:1810.00635.

[DP22] Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022. doi:10.1016/j.jlamp.2021.100717.

[DY13] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *Proc. of ICALP*, volume 7966 of *Lect. Notes Comput. Sci.*, pages 174–186. Springer, 2013. doi:10.1007/978-3-642-39212-2_18.

[FKD+21] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *Proc. of CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.36.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[GKL14] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proc. of CONCUR*, volume 8704 of *Lect. Notes Comput. Sci.*, pages 63–77. Springer, 2014. doi:10.1007/978-3-662-44584-6_6.

[Gug07] Alessio Guglielmi. A system of interaction and structure. *ACM Trans. Comput. Log.*, 8(1):1, 2007. doi:10.1145/1182613.1182614.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

[Hon93] Kohei Honda. Types for dyadic interaction. In *Proc. of CONCUR*, volume 715 of *Lect. Notes Comput. Sci.*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

[Hor20] Ross Horne. Session subtyping and multiparty compatibility using circular sequents. In *Proc. of CONCUR*, volume 171 of *LIPIcs*, pages 12:1–12:22. Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.12.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *Lect. Notes Comput. Sci.*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of POPL*, volume 43(1), pages 273–284. ACM, 2008. doi:10.1145/2827695.

[JBK22] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

[KD21a] Wen Kokke and Ornela Dardha. Deadlock-free session types in Linear Haskell. 2021. URL: https://arxiv.org/abs/2103.14481.

[KD21b] Wen Kokke and Ornela Dardha. Prioritise the best variation. In *Proc. of FORTE*, volume 12719 of *Lect. Notes Comput. Sci.*, pages 100–119. Springer, 2021. doi:10.1007/978-3-030-78089-0_6.

[KL17] Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017. doi:10.1016/j.ic.2016.03.004.

[KMP19a] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290337.

[KMP19b] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. In *Proc. of Linearity & TLLA*, volume 292 of *Electron. Proc. in Theor. Comput. Sci.*, pages 90–103. Open Publishing Association, 2019. doi:10.4204/EPTCS.292.5.

[Kob98] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998. doi:10.1145/276393.278524.

[Kob06] Naoki Kobayashi. A new type system for deadlock-free processes. In *Proc. of CONCUR*, volume 4137 of *Lect. Notes Comput. Sci.*, pages 233–247. Springer, 2006. doi:10.1007/11817949_16.

[Kob07] Naoki Kobayashi. Type systems for concurrent programs. 2007.

[LM15] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *Proc. of ESOP*, pages 560–584, 2015. doi:10.1007/978-3-662-46669-8_23.

[LM16] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *Proc. of ICFP*. ACM, 2016. doi:10.1145/2951913.2951921.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. doi:10.5555/63446.

[MP18] Fabrizio Montesi and Marco Peressotti. Classical transitions. 2018. URL: https://arxiv.org/abs/1803.01049.

[NT04] Matthias Neubauer and Peter Thiemann. An implementation of session types. In Bharat Jayaraman, editor, *Proc. of PADL*, volume 3057 of *Lect. Notes Comput. Sci.*, pages 56–70. Springer, 2004. doi:10.1007/978-3-540-24836-1_5.

[Pad13] Luca Padovani. From lock freedom to progress using session types. In *Proc. of PLACES*, volume 137, pages 3–19. Electron. Proc. in Theor. Comput. Sci., 2013. doi:10.4204/EPTCS.137.2.

[Pad14] Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Proc. of CSL-LICS*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

[PG15] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *Proc. of FoSSaCS*, volume 9034 of *Lect. Notes Comput. Sci.*, pages 3–22. Springer, 2015. doi:10.1007/978-3-662-46678-0_1.

[PN15] Luca Padovani and Luca Novara. Types for deadlock-free higher-order programs. In *Proc. of FORTE*, volume 9039 of *Lect. Notes Comput. Sci.*, pages 3–18. Springer, 2015. doi:10.1007/978-3-319-19195-9_1.

[QKB21] Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proc. of ACM Program. Lang.*, 5(ICFP):1–31, 2021. doi:10.1145/3473567.

[RC21] Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473584.

[SY19] Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290343.

[TCP12] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In *Proc. of FoSSaCS*, volume 7213 of *Lect. Notes Comput. Sci.*, pages 346–360. Springer, 2012. doi:10.1007/978-3-642-28729-9_23.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proc. of ESOP*, volume 7792 of *Lect. Notes Comput. Sci.*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proc. of PARLE*, volume 817 of *Lect. Notes Comput. Sci.*, pages 398–413. Springer, 1994. doi:10.1007/3-540-58184-7_118.

[VDG19] A. Laura Voinea, Ornela Dardha, and Simon J. Gay. Resource sharing via capability-based multiparty session types. In *Proc. of IFM*, volume 11918 of *Lect. Notes Comput. Sci.*, pages 437–455. Springer, 2019. doi:10.1007/978-3-030-34968-4_24.

[VGR06] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

[VRG04] Vasco Vasconcelos, António Ravara, and Simon J. Gay. Session types for functional multithreading. In *Proc. of CONCUR*, volume 3170 of *Lect. Notes Comput. Sci.*, pages 497–511. Springer, 2004. doi:10.1007/978-3-540-28644-8_32.

[VV13] Hugo Torres Vieira and Vasco Thudichum Vasconcelos. Typing progress in communication-centred systems. In *Proc. of COORDINATION*, volume 7890 of *Lect. Notes Comput. Sci.*, pages 236–250. Springer, 2013. doi:10.1007/978-3-662-43376-8_10.

[Wad12] Philip Wadler. Propositions as sessions. In *Proc. of ICFP*, pages 273–286, 2012. doi:10.1145/2398856.2364568.

[Wad14] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

[Wad15] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. doi:10.1145/2699407.

5.3 Discussion

This section proceeds as follows:

- In § 5.3.1, we discuss the relation between PCP and CP.
- In § 5.3.2, we define sound and complete priority inference for PCP.

In this section, PCP's processes and PGV's terms are printed in red, both of their types are printed in blue, and both of their priorities are printed in yellow, and all are rendered in a sans-serif font. To save on accessible colour combinations, the processes and terms, types, and priorities of *any other system* are printed in *pink*, *green*, and *periwinkle*, respectively, all are rendered in an italicised font with serif, and any relations, such as typing and reduction, are marked by a subscript.

5.3.1 Relation to Classical Processes

One notable omission in the literature is any proof that Priority CP is an extension of CP or that Priority CLL (PLL) is an extension of CLL. To discuss this matter formally, we must define what we mean by "extension". What does it mean for one system to extend another? We consider two options:

- If PLL extends the *proofs* of CLL, any valid CLL proof is a valid PLL proof, and any well-typed CP process is a well-typed PCP process.
- If PLL extends the *theorems* of CLL, any proposition provable in CLL is provable in PLL, and any type inhabited in CP is inhabited in PCP.

An extension of the *proofs* is a much stronger notion than an extension of the *theorems*, and, from the perspective of a process calculus, it is much more useful. Unfortunately, the stronger notion does not hold, as we discuss shortly. At the time of writing, I do not know if the weaker notion—an extension of theorems—holds.

The question is complicated by the priority annotations. What does it mean for a CLL proposition to be provable in PLL? Does it suffice if the proposition is provable with *some* priority assignment? Or should it be provable with *every* priority assignment? Priorities leak a fair amount of the structure of the underlying proof. For instance, any proposition where the priorities decrease as we descend into the proposition is unprovable, other than by absurdity or the axiom, e.g. there is no proof of (a) even though there is a proof of (b):

$$\text{(a)}\ \nvdash \mathbf{1}^{1} \otimes^{3} \mathbf{1}^{2} \qquad \text{(b)}\ \vdash \mathbf{1}^{2} \otimes^{1} \mathbf{1}^{3}$$

Hence, it would seem overly strict to require that a CLL proposition must be provable in PLL with *every* priority assignment.

PCP Conflates Tensor and Par

As Dardha and Gay [2018] discuss, PLL admits Mix and Mix_0, under the names T-Par and T-Halt. Consequently, it conflates one/bottom and partially conflates tensor/par

$$\mathbf{1}^{p} \multimap^{o} \bot^{q} \quad \text{and} \quad A \otimes^{p} B \multimap^{o} A \parr^{q} B$$

For instance, the conversion of tensor to par holds by the following derivations, which hold as long as $o < p, q$ and either $p < q$ or $q < p$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{A}, A} \quad \dfrac{}{\vdash \overline{B}, B}}{\vdash \overline{A}, \overline{B}, A, B}\, q < \mathrm{pr}(A,B)}{\vdash \overline{A}, \overline{B}, A \parr^{q} B}\, p < q}{\vdash \overline{A} \parr^{p} \overline{B}, A \parr^{q} B}\, o < p, q}{\vdash \overline{A} \parr^{p} \overline{B} \parr^{o} A \parr^{q} B}}{\vdash A \otimes^{p} B \multimap^{o} A \parr^{q} B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{A}, A} \quad \dfrac{}{\vdash \overline{B}, B}}{\vdash \overline{A}, \overline{B}, A, B}\, p < \mathrm{pr}(A,B)}{\vdash \overline{A} \parr^{p} \overline{B}, A, B}\, q < p}{\vdash \overline{A} \parr^{p} \overline{B}, A \parr^{q} B}\, o < p, q}{\vdash \overline{A} \parr^{p} \overline{B} \parr^{o} A \parr^{q} B}}{\vdash A \otimes^{p} B \multimap^{o} A \parr^{q} B}$$

These conflations are the natural consequence of admitting Mix and Mix0, even in the absence of priorities. PLL also admits MultiCut, which can be derived from T-Res and T-Par, which fully conflates tensor/par

$$A \parr^{p} B \multimap^{o} A \otimes^{q} B$$

The conversion of par to tensor holds by the following derivations, which hold as long as $o < p, q$ and either $p < q$ or $q < p$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{A}, A} \quad \dfrac{}{\vdash \overline{B}, B}}{\vdash \overline{A}, \overline{B}, A, B}\, q < \mathrm{pr}(A,B)}{\vdash \overline{A}, \overline{B}, A \otimes^{q} B}\, p < q}{\vdash \overline{A} \otimes^{p} \overline{B}, A \otimes^{q} B}\, o < p, q}{\vdash \overline{A} \otimes^{p} \overline{B} \parr^{o} A \otimes^{q} B}}{\vdash A \parr^{p} B \multimap^{o} A \otimes^{q} B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \overline{A}, A} \quad \dfrac{}{\vdash \overline{B}, B}}{\vdash \overline{A}, \overline{B}, A, B}\, p < \mathrm{pr}(A,B)}{\vdash \overline{A} \otimes^{p} \overline{B}, A, B}\, q < p}{\vdash \overline{A} \otimes^{p} \overline{B}, A \otimes^{q} B}\, o < p, q}{\vdash \overline{A} \otimes^{p} \overline{B} \parr^{o} A \otimes^{q} B}}{\vdash A \parr^{p} B \multimap^{o} A \otimes^{q} B}$$

In CLL, tensor and par capture independence and interdependence:

- If $\vdash_c \Gamma, A \otimes B$, the resources used by A and B are independent.
- If $\vdash_c \Delta, A \parr B$, the resources used by A and B are interdependent. How they depend on one another is unknown, but they are guaranteed to be free from cyclic dependencies.

These are naturally compositional. In $\vdash_c \Delta, A \parr B$, we do not know how the resources in A and B are used, so the only option that is guaranteed to avoid cyclic dependencies is to ensure that in $\vdash_c \Gamma, A \otimes B$ the resources used by A and B are independent.

PLL's "tensor" and "par" are not truly a tensor and par, and do not capture independence and interdependence. Morally, they are the

same self-dual connective, which may capture either independence or interdependence, but must specify *exactly* if and how its resources interdepend. The motivation to separate this self-dual connective into two dual connectives, "$\otimes^\circ$" and "$\parr^\circ$", comes from the process calculus interpretation, rather than the logic itself, as the separate connectives are used to guarantee session fidelity under the interpretation where "$\otimes^\circ$" is send and "$\parr^\circ$" is receive.

PCP Does Not Extend CP

The priorities in PCP expose information about the order in which a process uses its resources. Consequently, the additives in PCP are less expressive than the additives in CP, and, while we have not discussed the exponentials in this thesis, the exponentials in PCP are also less expressive than in CP.

Let us consider the typing rules for the offer in CP and PCP:

$$\frac{P \vdash_c \Gamma, x : A \qquad Q \vdash_c \Gamma, x : B}{x \triangleright \{\text{inl}: P; \text{inr}: Q\} \vdash_c \Gamma, x : A \,\&\, B}\ \text{T-Offer}$$

$$\frac{\mathsf{P} \vdash \Gamma, \mathsf{x} : \mathsf{A} \qquad \mathsf{Q} \vdash \Gamma, \mathsf{x} : \mathsf{B} \qquad \mathsf{o} < \mathrm{pr}(\Gamma)}{\mathsf{x} \triangleright \{\text{inl}: \mathsf{P}; \text{inr}: \mathsf{Q}\} \vdash \Gamma, \mathsf{x} : \mathsf{A} \,\&^\circ\, \mathsf{B}}\ \text{T-Offer}$$

The two typing rules ostensibly quite similar, with the only difference being the added priority constraint in PCP's typing rule. However, recall that, while the processes P and Q may differ in their use of the sessions A and B, the priority annotations on the typing environment Γ mean that both processes must use all other resources in the exact same order. CP's type system, on the other hand, imposes no such restriction.

Counterexample 5.1 (Extension)**.** *It is not true that every well-typed CP process is a well-typed PCP process.*

For example, the process

$$a \triangleright \{\text{inl}: x().\, y().\, a[];\ \text{inr}: y().\, x().\, a[]\}$$

is typeable in CP but not PCP.

In CP, its typing derivation is as follows:

$$\frac{\dfrac{\dfrac{}{a[] \vdash_c a : \mathbf{1}}}{y().\, a[] \vdash_c y : \bot, a : \mathbf{1}}}{x().\, y().\, a[] \vdash_c x : \bot, y : \bot, a : \mathbf{1}} \qquad \frac{\dfrac{\dfrac{}{a[] \vdash_c a : \mathbf{1}}}{x().\, a[] \vdash_c x : \bot, a : \mathbf{1}}}{y().\, x().\, a[] \vdash_c x : \bot, y : \bot, a : \mathbf{1}}}{a \triangleright \{\text{inl}: x().\, y().\, a[];\ \text{inr}: y().\, x().\, a[]\} \vdash_c x : \bot, y : \bot, a : \mathbf{1} \,\&\, \mathbf{1}}$$

In PCP, the process is not typeable, because the typing derivation requires that $\mathsf{p} < \mathsf{q}$ and $\mathsf{q} < \mathsf{p}$, where p and q are the priorities associated with the

actions on x and y.

$$\dfrac{\dfrac{\dfrac{}{a[] \vdash a : \mathbf{1}^{o_1}}\ q < o_1}{y().\, a[] \vdash y : \bot^{q}, a : \mathbf{1}^{o_1}}\ p < q \qquad \dfrac{\dfrac{}{a[] \vdash a : \mathbf{1}^{o_2}}\ p < o_2}{x().\, a[] \vdash x : \bot^{p}, a : \mathbf{1}^{o_1}}\ q < p}{a \triangleright \{\mathrm{inl}: x().\, y().\, a[];\ \mathrm{inr}: y().\, x().\, a[]\} \vdash x : \bot^{p}, y : \bot^{q}, a : \mathbf{1}^{o_1} \mathbin{\&^{o}} \mathbf{1}^{o_2}}$$

Therefore, not every well-typed CP process is a well-typed PCP process.

For the additives, we can partially work around this restriction by combining the entire typing environment into a single type, offering a session of type (omitting the priority annotations on the ⅋'s, to reduce eyestrain)

$$(\bot^{p_1} \parr \bot^{p_1} \parr \mathbf{1}^{o_1}) \mathbin{\&^{o}} (\bot^{p_2} \parr \bot^{q_2} \parr \mathbf{1}^{o_2})$$

However, this requires a global process transformation and changes the type of the session.

A similar example exists for PCP's exponentials. We can construct two sessions of type $\bot^{p} \parr^{o} \bot^{q}$ that use the sessions corresponding to p and q in opposite orders, and apply dereliction and contracting the resulting sessions. The process

$$?x[x_1, x_2].\, (\nu z \bar{z})(?x_1[x].\, x(y).\, x().\, y().\, z[] \parallel ?x_2[x].\, x(y).\, y().\, x().\, \bar{z}().\, a[])$$

is well-typed in CP with exponentials, using the typing rules and alternative notation from Wadler [2014, 3.4], reproduced below:

$$\textsc{T-Derelict}\ \dfrac{P \vdash_c \Gamma, y : A}{?x[y].\, P \vdash_c \Gamma, x : ?A} \qquad \textsc{T-Contract}\ \dfrac{P \vdash_c \Gamma, x_1 : ?A, x_2 : ?A}{?x[x_1, x_2].\, P \vdash_c \Gamma, x : ?A}$$

However, it is not typeable in PCP, using the typing rules from Dardha and Gay [2018, Figure 2], as typing derivation requires that p < q and q < p, reproduced below:

$$\textsc{T-Derelict}\ \dfrac{P \vdash \Gamma, y : A \quad o < \mathrm{pr}(\Gamma)}{?x[y].\, P \vdash \Gamma, x : ?^{o}A} \qquad \textsc{T-Contract}\ \dfrac{P \vdash \Gamma, x_1 : ?^{o_1}A, x_2 : ?^{o_2}A \quad o \le o_1 \quad o \le o_2 \quad o < \mathrm{pr}(\Gamma)}{?x[x_1, x_2].\, P \vdash \Gamma, x : ?^{o}A}$$

This restriction might be worked around to some extent by using the additives to encode the different usages of the typing environment and the duplicated session. However, this requires a global process transformation, and changes the type of the session.

PGV Does Not Extend GV

The counterexample to extension given for CP and PCP in the previous section is easily adapted to provide a counterexample to extension for GV and PGV.

Counterexample 5.2 (Extension)**.** *It is not true that every well-typed GV term is a well-typed PGV term.*

For example, the term

$$\begin{array}{l} \mathit{case}\ x\ \{\mathit{inl}\ x \mapsto x; M;\ \mathit{inr}\ x \mapsto x; N\} \\ \quad \textit{where} \\ \qquad M \triangleq \mathit{wait}\ y; \mathit{wait}\ z \\ \qquad N \triangleq \mathit{wait}\ z; \mathit{wait}\ y \end{array}$$

is typeable in GV but not in PGV.

In GV, using the typing rules in Figure I.2, its typing derivation is as follows. (In the following derivation, the premises for TM-App are stacked and the first premise of TM-LetUnit is omitted.)

$$\dfrac{\dfrac{\vdots}{y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} M : \mathbf{1}}}{x : \mathbf{1}, y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} x; M : \mathbf{1}} \quad \dfrac{\dfrac{\vdots}{y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} N : \mathbf{1}}}{x : \mathbf{1}, y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} x; N : \mathbf{1}}}{x : \mathbf{1} + \mathbf{1}, y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} \mathit{case}\ x\ \{\mathit{inl}\ x \mapsto x; M;\ \mathit{inr}\ x \mapsto x; N\} : \mathbf{1}}$$

where the typing derivations for M and N are

$$\dfrac{\dfrac{\dfrac{}{\vdash_{GV} \mathit{wait} : \mathit{end}_? \multimap \mathbf{1}}}{y : \mathit{end}_? \vdash_{GV} y : \mathit{end}_?}}{y : \mathit{end}_? \vdash_{GV} \mathit{wait}\ y : \mathbf{1}} \quad \dfrac{\dfrac{\dfrac{}{\vdash_{GV} \mathit{wait} : \mathit{end}_? \multimap \mathbf{1}}}{z : \mathit{end}_? \vdash_{GV} z : \mathit{end}_?}}{z : \mathit{end}_? \vdash_{GV} \mathit{wait}\ z : \mathbf{1}}}{y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} M : \mathbf{1}}$$

and

$$\dfrac{\dfrac{\dfrac{}{\vdash_{GV} \mathit{wait} : \mathit{end}_? \multimap \mathbf{1}}}{z : \mathit{end}_? \vdash_{GV} z : \mathit{end}_?}}{z : \mathit{end}_? \vdash_{GV} \mathit{wait}\ z : \mathbf{1}} \quad \dfrac{\dfrac{\dfrac{}{\vdash_{GV} \mathit{wait} : \mathit{end}_? \multimap \mathbf{1}}}{y : \mathit{end}_? \vdash_{GV} y : \mathit{end}_?}}{y : \mathit{end}_? \vdash_{GV} \mathit{wait}\ y : \mathbf{1}}}{y : \mathit{end}_?, z : \mathit{end}_? \vdash_{GV} N : \mathbf{1}}$$

respectively.

In PGV, using the typing rules in Figure II.2, its typing derivation is as follows. (In the following derivation, the premises for T-App are stacked and the first premise of T-LetUnit is omitted.)

$$\dfrac{\dfrac{\vdots}{y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} M : \mathbf{1}}}{x : \mathbf{1}, y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} x; M : \mathbf{1}} \quad \dfrac{\dfrac{\vdots}{y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} N : \mathbf{1}}}{x : \mathbf{1}, y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} x; N : \mathbf{1}}}{x : \mathbf{1} + \mathbf{1}, y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} \mathit{case}\ x\ \{\mathit{inl}\ x \mapsto x; M;\ \mathit{inr}\ x \mapsto x; N\} : \mathbf{1}}$$

where the typing derivations for M and N are

$$\dfrac{\dfrac{\dfrac{}{\vdash^{\bot} \mathit{wait} : \mathit{end}^{p}_? \multimap^{\top, p} \mathbf{1}}}{y : \mathit{end}^{p}_? \vdash^{\bot} y : \mathit{end}^{p}_?}}{y : \mathit{end}^{p}_? \vdash^{p} \mathit{wait}\ y : \mathbf{1}} \quad \dfrac{\dfrac{\dfrac{}{\vdash^{\bot} \mathit{wait} : \mathit{end}^{q}_? \multimap^{\top, q} \mathbf{1}}}{z : \mathit{end}^{q}_? \vdash^{\bot} z : \mathit{end}^{q}_?}}{z : \mathit{end}^{q}_? \vdash^{q} \mathit{wait}\ z : \mathbf{1}}}{y : \mathit{end}^{p}_?, z : \mathit{end}^{q}_? \vdash^{p \sqcup q} M : \mathbf{1}}\ p < q$$

and

$$\dfrac{\dfrac{\vdash^{\bot} \mathit{wait} : \mathit{end}_?^{q} \multimap^{\top,q} \mathbf{1}}{z : \mathit{end}_?^{q} \vdash^{\bot} z : \mathit{end}_?^{q}} \quad \dfrac{\vdash^{\bot} \mathit{wait} : \mathit{end}_?^{p} \multimap^{\top,p} \mathbf{1}}{y : \mathit{end}_?^{p} \vdash^{\bot} y : \mathit{end}_?^{p}}}{\dfrac{z : \mathit{end}_?^{q} \vdash^{q} \mathit{wait}\ z : \mathbf{1}}{y : \mathit{end}_?^{p}, z : \mathit{end}_?^{q} \vdash^{p \sqcup q} N : \mathbf{1}} \quad \dfrac{y : \mathit{end}_?^{p} \vdash^{p} \mathit{wait}\ y : \mathbf{1}}{}} \; q < p$$

respectively, which require that $p < q$ *and* $q < p$*, respectively, where* p *and* q *are the priorities associated with the actions on* y *and* z*, respectively.*

Therefore, not every well-typed GV term is a well-typed PGV term.

There is no corresponding counterexample for the exponentials, since the presentation of PGV in Paper II omits the exponentials. However, any variant of PGV that introduces replication, following the work by Lindley and Morris [2015] and Dardha and Gay [2018], without any further changes, would certainly admit such an example.

My Priorities Leave Me No Choice

What is the issue with priorities that causes PCP and PGV to be non-extensions of CP and GV, respectively?

The priorities impose a linear order in which a collection of resources is to be used. For instance, under the typing environment $x : A^{1}, y : B^{2}$, the resource x must be used first and the resource y must be used second. When the same typing environment is shared by different branches in a program, as is the case for the additives and the exponentials, each branch is required to use those resources in the same order, which fundamentally limits the expressiveness of branching. The structure offered by the ordering on priorities—which are the natural numbers extended with a lower and upper bound—is insufficient to capture different uses across branches.

To address this issue, we would have to extend the structure of priorities to be able to capture branching, e.g. by adding branching structure to the priorities, or by marking constraints by the choices under which they must hold. Any solution would likely be closely related to the introduction of proof-boxes in the proof nets for linear logic [Girard, 1987, p. 43].

On the other side, since the issue is that it is not generally possible to impose a linear order on processes with branching, this is a good indication that priorities *do* extend the multiplicative fragment of linear logic, as well as the fragment with fixed points [e.g. Lindley and Morris, 2016b], since neither introduces branching.

Identity Expansion Fails for Priority CP

The type system for PCP does not satisfy identity expansion.

Counterexample 5.3 (Identity Expansion)**.** *It is not true that, if there exists a proof of* $\vdash \Gamma$ *that uses T-LINK, then there exists a proof of* $\vdash \Gamma$ *that does not use T-LINK.*

For example, given the following proof

$$\frac{}{\vdash \mathbf{1}^{1} \otimes^{3} \mathbf{1}^{2}, \bot^{1} \parr^{3} \bot^{2}}\ \textit{T-LINK}$$

there exists no proof with the same conclusion that does not use T-LINK.

The reason is that PCP's type system omits certain constraints that are, generally speaking, syntactically impossible to violate. For instance, the typing rule T-RECV introduces the type $A \parr^{o} B$, but does not require that o is smaller than the priorities on A and B. This constraint is not required, since a process that uses an endpoint of type $A^{1} \parr^{3} B^{2}$ would have to act on the endpoint of type A^{1} before receiving it. I refer to these constraints as the *priority tree* of a type, since they capture the tree structure of the type. For instance, for the type $A^{p} \parr^{o} B^{q}$, the priority tree contains the constraints $o < p$ and $o < q$ and all constraints imposed by the structure of A and B. The priority tree of a type, written T_A, is a set of constraints, defined as follows:

$$\begin{aligned} T_{A \otimes^{o} B}, T_{A \parr^{o} B}, T_{A \oplus^{o} B}, T_{A \&^{o} B} &\triangleq \{o < \mathrm{pr}(A), o < \mathrm{pr}(B)\} \cup T_A \cup T_B \\ T_{\mathbf{1}^{o}}, T_{\bot^{o}}, T_{\mathbf{0}^{o}}, T_{\top^{o}} &\triangleq \varnothing \end{aligned}$$

PCP's type system omits the constraints in the priority tree for all types. This includes the types introduced by links and absurd offers. Unfortunately, the syntactic argument does not apply in these cases. Consequently, PCP's type system allows links, such as the process in Counterexample 5.3, whose priorities violate the constraints imposed by the priority trees. Since no process, other than the link and the absurd offer, can introduce such types, these links are, in essence, dead, as they can only ever be connected to other links or absurd offers, and, as mentioned, they do not satisfy identity expansion.

To address this issue, I conjecture that it suffices to require that the constraints in priority tree for all types introduced by links and absurd offers must hold.

5.3.2 Priority Inference

In this section, I introduce priority inference for PCP. Usually, a type inference system is presented as an algorithmic variant of the type system, whose derivations take terms as input and produce types as output. Such type inference derivations are compositional, and produce locally correct information.

Deadlock freedom is a global property. While CP and HCP guarantee deadlock freedom locally and compositionally, they do so by enforcing a much stronger and more restrictive invariant: the tree and forest structure of the connection graph. In essence, PCP guarantees global deadlock freedom simply by checking it, *globally*. Consequently, PCP's typing derivations are not meaningfully compositional. Priorities witness global deadlock freedom, and, as such, capture global information. Morally, if you connect two PCP processes, you must re-check if they are deadlock free. (Practically, you can connect two PCP processes if and only if you already chose their priorities to witness the global deadlock freedom of the resulting process.) As typing derivations are not compositional, there is little hope for a local, compositional priority inference system for PCP. Instead, I structure priority inference as a two-stage process, which factors out the local, compositional portion into the first stage, and defers the global check to the second stage.

Priority inference is structured as follows:

1. I define pre-processes and pre-types, which replace priorities with priority metavariables, and pre-typing, which ensures communication safety, but not deadlock freedom. Crucially, pre-typing is local and compositional.
2. I prove that a pre-process is a process only if it is deadlock-free.

In this section, processes, types, and priorities are printed in red, blue, and yellow, respectively, and all three are rendered in a sans-serif font, whereas pre-processes, pre-types, and priority metavariables are printed in *pink*, *green*, and *periwinkle*, respectively, and all three are rendered in an italicised font with serif. The pre-typing sequent is marked by a subscript "PI".

Priority metavariables are names, whereas priorities are natural numbers. Let o, p, and q range over priority metavariables. Priority metavariables should be unique, i.e. no priority metavariable should occur more than once in a pre-process, pre-type, pre-typing environment, or pre-typing derivation. I use Barendregt's convention for priority metavariables, and assume this uniqueness, rather than explicitly renaming duplicate priority metavariables.

Pre-processes are the same as processes, but annotated by pre-types rather than types. Let x, y, z, and w range over endpoint names, and let P, Q, and R range over pre-processes. Binding for pre-processes is the same as binding for processes.

Pre-types are the same as types, but annotated by priority metavariables rather than priorities. Pre-types are well-formed if and only if each connective is annotated with a distinct priority metavariable. Let A, B, C, and D as well as $\bar{A}$, $\bar{B}$, $\bar{C}$, and $\bar{D}$ range over pre-types. Following our convention for endpoints, I write A and $\bar{A}$ to imply that the types

associated with these pre-types are dual, which I define shortly. The set of priority metavariables in a pre-type A, written $\mathrm{fp}(A)$, is the set of all priority metavariables that occur in the pre-type A. Two pre-types are *equivalent*, written $A \approx B$, if they are equal up to priority metavariables.

$$\begin{array}{llll} A_1 \otimes^p A_2 \approx B_1 \otimes^q B_2 & \iff & A_1 \approx B_1 \wedge A_2 \approx B_2 & \mathbf{1}^p \approx \mathbf{1}^q \\ A_1 \parr^p A_2 \approx B_1 \parr^q B_2 & \iff & A_1 \approx B_1 \wedge A_2 \approx B_2 & \bot^p \approx \bot^q \\ A_1 \oplus^p A_2 \approx B_1 \oplus^q B_2 & \iff & A_1 \approx B_1 \wedge A_2 \approx B_2 & \mathbf{0}^p \approx \mathbf{0}^q \\ A_1 \mathbin{\&}^p A_2 \approx B_1 \mathbin{\&}^q B_2 & \iff & A_1 \approx B_1 \wedge A_2 \approx B_2 & \top^p \approx \top^q \end{array}$$

Two pre-types are dual, written $A \sim \bar{A}$, if they are dual up to priority metavariables.

$$\begin{array}{llll} A_1 \otimes^p A_2 \sim \bar{A}_1 \parr^q \bar{A}_2 & \iff & A_1 \sim \bar{A}_1 \wedge A_2 \sim \bar{A}_2 & \mathbf{1}^p \sim \bot^q \\ A_1 \parr^p A_2 \sim \bar{A}_1 \otimes^q \bar{A}_2 & \iff & A_1 \sim \bar{A}_1 \wedge A_2 \sim \bar{A}_2 & \bot^p \sim \mathbf{1}^q \\ A_1 \oplus^p A_2 \sim \bar{A}_1 \mathbin{\&}^q \bar{A}_2 & \iff & A_1 \sim \bar{A}_1 \wedge A_2 \sim \bar{A}_2 & \mathbf{0}^p \sim \top^q \\ A_1 \mathbin{\&}^p A_2 \sim \bar{A}_1 \oplus^q \bar{A}_2 & \iff & A_1 \sim \bar{A}_1 \wedge A_2 \sim \bar{A}_2 & \top^p \sim \mathbf{0}^q \end{array}$$

Pre-typing environments are the same as typing environments, but contain pre-type assignments, rather than type assignments. Let Γ and Δ range over pre-typing environments. Pre-typing environments are well-formed if and only if the priority metavariables in all pre-types are distinct. The set of priority metavariables in a pre-typing environment Γ, written $\mathrm{fp}(\Gamma)$, is the set of priority metavariables that occur in all the pre-types in the pre-typing environment Γ.

Priority graph (ranged over by G, H) are mixed graphs whose vertices are priority metavariables. I informally revisit the relevant definitions. For a detailed discussion, see § A.1.

- A *mixed graph* G has a set of vertices (denoted V_G, ranged over by u, v), a set of edges (denoted E_G), a set of arcs (denoted A_G). Edges are unordered pairs denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$. The set of edges may not contain loops uu. Arcs are ordered pairs denoted by juxtaposition overset with an arrow to indicate the direction, i.e. $\overrightarrow{uv} \triangleq (u, v)$. The set of arcs may not contain loops $\overrightarrow{uu}$.
- For any graph G with vertices $u, v \in V_G$, u is *adjacent* to v when there exists some edge $uv \in E_G$ or some arc $\overrightarrow{uv} \in A_G$.
- A *walk* w is a sequence of pairwise adjacent vertices.
- A *path* p is a walk with no repeated vertices, except possibly the first and last.
- A *cycle* c is a path that begins and ends at the same vertex.
- A walk is *essentially directed* when it contains at least one arc.
- A graph is *essentially acyclic* if and only if it contains no essentially directed cycles.
- The *undirected reachability* relation (denoted by $\sim_G$) is the equivalence closure over E_G.
- The *essentially directed reachability* relation (denoted by $<_G$) is the transitive closure over A_G quotiented by $\sim_G$.

- The *empty graph* with vertices V, written $\bar{K}_V$, is the graph consisting of vertices V, with no edges or arcs.
- The *graph union* of G_1 and G_2, denoted by $G_1 \cup G_2$, is defined by, for each projection, taking the union of the projection of G_1 and G_2, e.g. $V_{G_1 \cup G_2} \triangleq V_{G_1} \cup V_{G_2}$, $E_{G_1 \cup G_2} \triangleq E_{G_1} \cup E_{G_2}$, etc..
- The *directed rooting* of G in v, written $v \lessdot G$, is the graph formed by adding the vertex v and adding an arc from v to every other vertex of G, i.e. $V_{v \lessdot G} \triangleq \{v\} \cup V_G$ and $A_{v \lessdot G} \triangleq \{\overrightarrow{vu} | u \in V_G\} \cup A_G$. Directed rooting preserves the remaining projections, e.g. $E_{v \lessdot G} \triangleq E_G$.

The *priority tree* of a pre-type, written T_A, is the priority graph whose vertices are the priority metavariables in A, with arcs flowing along the structure of the type.

$$\begin{aligned} T_{A \otimes^o B}, T_{A \parr^o B}, T_{A \oplus^o B}, T_{A \& ^o B} &\triangleq o \lessdot (T_A \cup T_B) \\ T_{\mathbf{1}^o}, T_{\bot^o}, T_{\mathbf{0}^o}, T_{\top^o} &\triangleq o \end{aligned}$$

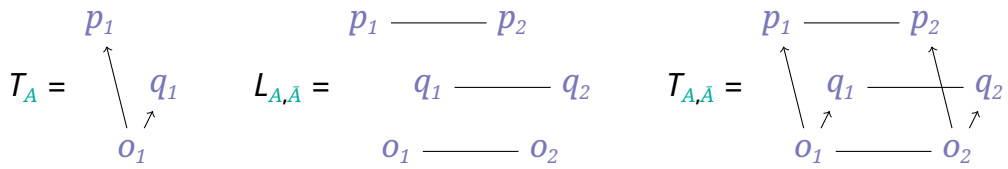
The *priority link* of two pre-types, written $L_{A,\bar{A}}$, is the bipartite priority graph with edges between the corresponding priority metavariables in A and $\bar{A}$. The priority link is defined if and only if $A \sim \bar{A}$.

$$\begin{aligned} L_{A \otimes^p B, \bar{A} \parr^q \bar{B}}, L_{A \parr^p B, \bar{A} \otimes^q \bar{B}}, L_{A \oplus^p B, \bar{A} \& ^q \bar{B}}, L_{A \& ^p B, \bar{A} \oplus^q \bar{B}} &\triangleq pq \cup L_{A,\bar{A}} \cup L_{B,\bar{B}} \\ L_{\mathbf{1}^p, \bot^q}, L_{\bot^p, \mathbf{1}^q}, L_{\mathbf{0}^p, \top^q}, L_{\top^p, \mathbf{0}^q} &\triangleq pq \end{aligned}$$

The *priority link-tree* of two pre-types, written $T_{A,\bar{A}}$, is the union of the two priority trees for A and $\bar{A}$ and the priority link for A and $\bar{A}$.

$$T_{A,\bar{A}} \triangleq T_A \cup T_{\bar{A}} \cup L_{A,\bar{A}}$$

To illustrate these definitions, let us look at an example. Let A and $\bar{A}$ be the pre-types $\mathbf{1}^{p_1} \otimes^{o_1} \mathbf{1}^{q_1}$ and $\bot^{p_1} \parr^{o_2} \bot^{q_2}$, respectively. The priority tree T_A, priority link $L_{A,\bar{A}}$, and priority link-tree $T_{A,\bar{A}}$ are as follows:



The pre-typing judgment $P \vdash_{PI} \Gamma \mid G$ means that P is well-typed if, for each pre-type assignment $x : A$ in Γ, exactly one pre-process in P uses the endpoint x according to the session pre-type A. The priority graph G is an output of the pre-typing derivation.

Definition 5.4 (Pre-Typing)**.** *A pre-process P is well-typed under some pre-typing environment Γ if there exists a pre-typing derivation with conclusion $P \vdash_{PI} \Gamma \mid G$ for some G that uses the pre-typing rules in Figure 5.1.*

A *priority substitution* assigns a priority to each priority metavariable. Let σ range over priority substitutions. A priority substitution translates pre-processes to processes, pre-types to types, and pre-typing environments

$$\frac{}{x{\leftrightarrow}y \vdash_{PI} x : A, y : \bar{A} \mid L_{A,\bar{A}}}\ \text{PI-Link} \qquad \frac{P \vdash_{PI} \Gamma, x : A, \bar{x} : \bar{A} \mid G}{(\nu x\bar{x})P \vdash_{PI} \Gamma \mid L_{A,\bar{A}} \cup G}\ \text{PI-Res}$$

$$\frac{P \vdash_{PI} \Gamma \mid G \qquad Q \vdash_{PI} \Delta \mid H}{P \parallel Q \vdash_{PI} \Gamma, \Delta \mid G \cup H}\ \text{PI-Par} \qquad \frac{}{0 \vdash_{PI} \varnothing \mid \bar{K}_{\varnothing}}\ \text{PI-Halt}$$

$$\frac{P \vdash_{PI} \Gamma, y : A, x : B \mid G}{x[y].P \vdash_{PI} \Gamma, x : A \otimes^{o} B \mid o < G}\ \text{PI-Send} \qquad \frac{P \vdash_{PI} \Gamma, y : A, x : B \mid G}{x(y).P \vdash_{PI} \Gamma, x : A \parr^{o} B \mid o < G}\ \text{PI-Recv}$$

$$\frac{P \vdash_{PI} \Gamma \mid G}{x[].P \vdash_{PI} \Gamma, x : \mathbf{1}^{o} \mid o < G}\ \text{PI-Close} \qquad \frac{P \vdash_{PI} \Gamma \mid G}{x().P \vdash_{PI} \Gamma, x : \bot^{o} \mid o < G}\ \text{PI-Wait}$$

$$\frac{P \vdash_{PI} \Gamma, x : A \mid G}{x \triangleleft \mathrm{inl}.P \vdash_{PI} \Gamma, x : A \oplus^{o} B \mid o < G}\ \text{PI-Select}_1$$

$$\frac{P \vdash_{PI} \Gamma, x : B \mid G}{x \triangleleft \mathrm{inr}.P \vdash_{PI} \Gamma, x : A \oplus^{o} B \mid o < G}\ \text{PI-Select}_2$$

$$\frac{P \vdash_{PI} \Gamma, x : A \mid G_1 \qquad Q \vdash_{PI} \Gamma, x : B \mid G_2}{x \triangleright \{\mathrm{inl}: P; \mathrm{inr}: Q\} \vdash_{PI} \Gamma, x : A \mathbin{\&}^{o} B \mid o < (G_1 \cup G_2)}\ \text{PI-Offer}$$

$$\frac{N = \mathrm{fn}(\Gamma)}{x \lightning N \vdash_{PI} \Gamma, x : \top^{o} \mid \bar{K}_{\mathrm{fn}(\Gamma, x:\top^{o})}}\ \text{PI-Absurd}$$

Figure 5.1: Pre-Typing Rules for Priority CP

to typing environments, by pointwise applying the priority substitution to each priority metavariable.

Priority inference is *sound*. If the priority graph produced by the pre-typing derivation is essentially acyclic, there exists some priority substitution such that the resulting process is typeable in PCP.

Proposition 5.5 (Soundness)**.** *If* $P \vdash_{\mathit{PI}} \Gamma \mid G$ *and* G *is essentially acyclic, then there exists some priority substitution* σ *such that* $\sigma(P) \vdash \sigma(\Gamma)$*.*

Proof. Let H be the quotient graph of G by its edges E_G, i.e. G/E_G. Consequently, H has no edges and is a directed graph. Since G is essentially acyclic, H is acyclic. By topological sort, there exists a linear ordering S_n of the vertices of H such that, if $\vec{uv} \in A_H$, then u comes before v in S_n. Let the priority substitution $\sigma : V_G \to \mathbb{N}$ be the function that maps each priority metavariable to its position in the linear ordering S_n, i.e. $\sigma = \{o \mapsto i \mid o \in S_i\}$.

The typing derivation for $\sigma(P) \vdash \sigma(\Gamma)$ is constructed by induction on the typing derivation for $P \vdash_{\mathit{PI}} \Gamma \mid G$.

- In the cases for PI-Send, PI-Recv, PI-Close, PI-Wait, PI-Select$_1$, PI-Select$_2$, PI-Offer, and PI-Absurd, the result follows from the induction hypothesis and the rules T-Send, T-Recv, T-Close, T-Wait, T-Select$_1$, T-Select$_2$, T-Offer, and T-AbsurdZap, respectively. The constraint $\mathsf{o} < \mathrm{pr}(\Gamma)$ is satisfied by the definition of σ and the rooting of the priority graph $o < G$.
- In the case for PI-Link, the result follows from the rule PI-Link. In the case for PI-Res, the result follows from the induction hypothesis and the rule PI-Res. In both cases, duality is satisfied by the definition of σ and the priority link $L_{A,\bar{A}}$.
- In the case for PI-Par, the result follows from the induction hypotheses and the rule T-Par.
- In the case for PI-Halt, the result follows from the rule T-Halt.

□

Priority erasure replaces priority annotations with fresh priority metavariables, and translates processes to pre-processes, types to pre-types, and typing environments to pre-typing environments, by pointwise applying the priority erasure to each priority metavariable.

Definition 5.6 (Priority Erasure)**.** Priority erasure, *written* $\lfloor \cdot \rfloor_{\mathit{PI}}$*, maps processes to pre-processes, types to pre-types, typing environments to pre-typing environments, and typing derivations to pre-typing derivations, by replacing all priorities with fresh priority metavariables.*

Priority inference satisfies *completeness*. If the process is well-typed in PCP, then the priority graph produced by the pre-typing derivation

is essentially acyclic, and the order of the original priority assignment respects reachability in the priority graph.

Proposition 5.7 (Completeness)**.** *If* $\mathsf{P} \vdash \Gamma$*, then* $\lfloor \mathsf{P} \rfloor_{PI} \vdash_{PI} \lfloor \Gamma \rfloor_{PI} \mid G$ *such that the produced priority graph* G *is essentially acyclic, and, if* σ *is the priority substitution such that* $\sigma(P) = \mathsf{P}$ *and* $\sigma(\Gamma) = \Gamma$*, then* $p <_G q \implies \sigma(p) < \sigma(q)$*.*

Proof. By induction on the derivation of $\mathsf{P} \vdash \Gamma$. In each case, we construct the priority substitution σ by matching the priority metavariables in the erased types to the priorities on the original type, and prove that σ is a graph homomorphism from the produced priority graph G to the graph corresponding to the order on the natural numbers. Consequently, σ witnesses the fact that G is linearisable, and therefore is essentially acyclic.

□

5.4 Conclusion

In this chapter, we introduced Priority GV with its typing rules, reduction semantics, and metatheory, and we revisited Priority CP with its typing rules, reduction semantics, and metatheory, and introduced an operational correspondence between PCP and PGV. Priority CP and Priority GV are variants of CP and GV that use priority typing to permit processes with benign cyclic connections. For PGV, we proved *preservation* (Theorem II.3.5) and *global progress* (Theorem II.3.14). For PCP, we dropped the commuting conversions, which cause Dardha and Gay's PCP to be non-confluent, from the reduction semantics, we added the additive units, and we proved *preservation* (Theorem II.3.5), and proved that *global progress* (Theorem II.3.14) continues to hold. We related PCP to PGV by means of a translation, proved that it preserves types (Theorem II.4.6), and proved that it gives rise to a complete (Theorem II.4.7) and sound (Theorem II.4.10) operational correspondence. I demonstrated that PCP and PGV are not extensions of CP and GV, respectively, and that PCP does not satisfy identity expansion. Finally, I introduced priority inference for PCP.

I conjecture that multiplicative PCP—the fragment $(\otimes, \mathbf{1}, \parr, \bot)$—does extend multiplicative CP, and proving this would be an interesting topic for future work. It is easily verified that most typing rules for multiplicative CP preserve essential acyclicity of the priority graph. The exception is cut. I conjecture that a successful proof for cut proceeds by induction on the cut formula, and relies on *splitting*—the property that the subgraphs of the priority graph which are reachable from the arguments of a tensor are disjoint.

It would be interesting to investigate the exact relation between priority

graphs and proof nets. Certainly, they are not equivalent, since priority graphs encode the order in which the sequent calculus rules are used. However, I conjecture that if priority graphs were extended with type information, they would contain sufficient information to reconstruct the entire typing derivation, and hence, they encode the entire process.

It would be interesting to extend PCP and PGV with fixed points, following Lindley and Morris [2016b] and Padovani [2014], and with first- and second-order quantifiers, interpreted as priority and type polymorphism.

Finally, it would be interesting to investigate solutions for the expressivity of additives in priority-based calculi by extending the structure of priority graphs with choice, e.g. by boxing, as done in the proof nets of linear logic [Girard, 1987].

Part IV

Deadlock-Free Session Types in Linear Haskell

Chapter 6

Implementations

In previous chapters, I have made the claim that GV's session-typed communication is easy to implement as a library, especially when the host language already supports linear types. To evidence this claim, this chapter presents an implementation of both HGV and PGV's session-typed concurrency primitives as a library, named `priority-sesh`, implemented in Linear Haskell [Marlow et al., 2010, Bernardy et al., 2018].

The library only implements HGV and PGV's concurrency primitives. This is a double-edged sword. Only implementing the concurrency primitives makes GV easy to implement, but also easy to implement incorrectly, since this makes a number of assumptions about the host language that are difficult to prove.

- It relies on the host language to provide the basic ingredients of a functional language—e.g. functions, pairs, sums, etc..—and basic concurrency primitives—i.e. threads and channels.
- It assumes that the host language and GV agree on their evaluation strategy, or that GV's correctness guarantees are invariant under the choice of evaluation strategy. (For Haskell, we can say with confidence that the first option does not hold, since GV's semantics are call-by-value, whereas Haskell's semantics are call-by-need.)
- It assumes that the host language and GV agree on the semantics for threads and channels.

The bulk of the chapter consists of the paper *Deadlock-Free Session Types in Linear Haskell* by Kokke and Dardha [2021b], hereafter referred to as Paper III. References made from the main body of this thesis into Paper III will be prefixed by an "III", e.g. "Theorem III.3.1". This chapter proceeds as follows:

- In § 6.1, I provide a legend and an errata for Paper III.
- In § 6.2, we present an implementation of both HGV and PGV's session-typed communication as a library in Linear Haskell. This

section consists entirely of Paper III, and proceeds as follows:

- In § III.2.1, we implement one-shot channels.
- In § III.2.2, we implement session-typed channels.
- In § III.2.3, we show that a restriction of the interface defined in § III.2.2 corresponds to HGV's communication primitives.
- In § III.2.4, we implement priority-based channels.
- In § III.3, we show that the interface defined in § III.2.4 is the monadic reflection of PGV's communication primitives.
- In § III.4, we discuss the related work. Notably, we compare a variety of session types in Haskell, extending the analysis provided by Orchard and Yoshida [2017].
- In § III.5, we discuss our experience using Linear Haskell as well as potential directions for future work.

6.1 Legend and Errata

The conventions and terminology in Paper I are different from those used in the rest of this thesis.

- The concurrency primitives, session types, and priorities in the Linear Haskell implementation as well as the terms, types, and priorities of Priority GV are printed in red, blue, and green, respectively, and are rendered in an italicised or bolded font with serif.

There are several other notable differences between HGV and PGV and their implementations in the library.

First, the library implements several extensions to GV.

- The library implements session cancellation, following Exceptional GV [EGV, Fowler et al., 2019].

  Strictly speaking, this means the implementation is *affine* rather than *linear*, in the sense that values can be left unused. This is important in any practical implementation of session types, regardless of what typing facilities are offered by the host language. We can design a session-typed language in the simplified setting where communication always succeeds and no process ever crashes, but this assumption is immediately shattered upon contact with reality.

  This extension *also* allows us to implement GV as a library in host languages whose type systems are affine rather than linear, such as the predecessor to the library, *Rusty Variation* by Kokke [2019], which is implemented in Rust [Matsakis and Klock, 2014].

- The library has polymorphic and recursive session types, simply

by virtue of being implemented in Haskell. The recursive sessions in the library do not provide any termination guarantees. This is unsurprising, since Haskell does not offer any termination guarantees. Hence, the recursive sessions in the library do not correspond to the recursive sessions in Lindley and Morris' µGV, but they may correspond to the variant that identifies the greatest and least fixed points [Lindley and Morris, 2016b, § 2.3, under "Nontermination"].

Second, the library makes several simplifications that are common in practical variants of GV, which relax the correspondence with CLL.

- The link primitive is omitted from both the implementation of session-typed channels (in § III.2.2) and the implementation of priority-based session-typed channels (in § III.2.4). This is common in implementations of GV, or variants of GV which do not aim to prove a correspondence with a variant of CP.
- The type system conflates $\mathrm{end}_!$ and $\mathrm{end}_?$ into *End*. This is equivalent to postulating Mix and Mix_0 in CLL, i.e. it preserves deadlock-freedom, but relaxes the structure of the connection graph from a tree to a forest. Consequently, processes may become disconnected. Consequently, exceptions raised in disconnected child processes are not propagated to the main process. This is not a serious problem, as it cannot affect the outcome of the main computation, and does not justify the added complexity of separating $\mathrm{end}_!$ and $\mathrm{end}_?$.
  (If the library's concurrency primitives are used together with other forms of inter-process communication, exceptions raised in disconnected child processes can affect the outcome of the main computation, but such uses already lose the guarantee of deadlock-freedom.)

Finally, the library makes several changes that are required to correctly embed HGV and PGV into Linear Haskell.

- The library relies on Linear Haskell to provide linearity checking. However, Linear Haskell and GV have different notions of linearity. In GV, linearity is a property of *values*. If a value is linear, it must be used exactly once. In Linear Haskell, linearity is a property of *functions*. If a function is linear, it must use its argument exactly once. To accommodate this difference, it is important that session-typed endpoints, which must be used linearly, are only ever obtained as the arguments of linear functions.

  In HGV's implementation, the concurrency primitive fork is replaced with the equivalent concurrency primitive connect from Wadler's GV (see § III.2.3). Whereas fork returns the dual endpoint, connect passes it to another continuation. As endpoints can only be

created by connect, this guarantees they are used linearly.

$$\mathsf{fork} : (\mathsf{S} \multimap \mathsf{end}_!) \multimap \overline{\mathsf{S}} \qquad \mathsf{connect} : (\mathsf{S} \multimap \mathsf{end}_!) \multimap (\overline{\mathsf{S}} \multimap \mathsf{T}) \multimap \mathsf{T}$$

(The type of "*connect*" in the library differs from the type of connect presented above. We shall discuss this shortly.)

In PGV's implementation, the concurrency primitives are lifted into a linear graded monad (see "*Sesh*", in § III.2.4, under "The Communication Monad"). As the bind operation for this monad is linear, this guarantees that endpoints are used linearly.

- In HGV's implementation, the type of "*connect*" differs from the type of Wadler's connect, as presented above.

$$\mathit{connect} :: \mathit{Session}\ s \Rightarrow (s \multimap \mathit{IO}\ ()) \multimap (\mathit{Dual}\ s \multimap \mathit{IO}\ a) \multimap \mathit{IO}\ a$$

The function corresponding to the child thread has type $s \multimap \mathit{IO}\ ()$ instead of $\mathsf{S} \multimap \mathsf{end}_!$, which means it returns the unit, rather than an endpoint of type $\mathsf{end}_!$.

The change of type is due to two relaxations of GV's typing. The first is the conflation of $\mathsf{end}_!$ and $\mathsf{end}_?$ into *End*, discussed above. The second is the switch from the synchronous *End* to the asynchronous unit, which lets child processes terminate asynchronously, without synchronising with their parents.

- In PGV's implementation, the concurrency connectives are lifted into the graded linear monad Sesh^q_p, which permits us to encode PGV's constraints in Haskell's type system as type-level constraints.

There are minor errors in Paper III:

- A phrase in the first paragraph on Page III.6 reads "For instance, for *totallyFine* we can assign the number 0 to send (ch_{s1}) and recv (ch_{r2}), and 1 to send (ch_{s2}) and recv (ch_{r1})", but should read "For instance, for *totallyFine* we can assign the number 0 to send (ch_{s1}) and recv (ch_{r1}), and 1 to send (ch_{s2}) and recv (ch_{r2})".

6.2 Paper III: Deadlock-Free Session Types in Linear Haskell

This section contains the paper with the same title, in collaboration with Ornela Dardha, which was originally published in the proceedings of the ACM SIGPLAN International Symposium on Haskell (Haskell 2021).

The work presented in the paper was conceived of by all the authors. I implemented the Haskell library, developed the refined version of Priority GV's type system and its monadic reflection, wrote the initial draft of the paper, and co-authored the comparison with existing Haskell libraries for session-typed channels.

Deadlock-Free Session Types in Linear Haskell

Wen Kokke
University of Edinburgh
Edinburgh, Scotland
wen.kokke@ed.ac.uk

Ornela Dardha
University of Glasgow
Glasgow, Scotland
ornela.dardha@glasgow.ac.uk

Abstract

Priority Sesh is a library for session-typed communication in Linear Haskell which offers strong compile-time correctness guarantees. Priority Sesh offers two deadlock-free APIs for session-typed communication. The first guarantees deadlock freedom by restricting the process structure to trees and forests. It is simple and composable, but rules out cyclic structures. The second guarantees deadlock freedom via priorities, which allows the programmer to safely use cyclic structures as well.

Our library relies on Linear Haskell to guarantee linearity, which leads to easy-to-write session types and more idiomatic code, and lets us avoid the complex encodings of linearity in the Haskell type system that made previous libraries difficult to use.

CCS Concepts: • **Theory of computation** → *Linear logic*; *Type theory*.

Keywords: session types, linear haskell, deadlock freedom

ACM Reference Format:
Wen Kokke and Ornela Dardha. 2021. Deadlock-Free Session Types in Linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471874.3472979

1 Introduction

Session types are a type formalism used to specify and verify communication protocols [26–28, 62]. They've been studied extensively in the context of the π-calculus [58], a process calculus for communication an concurrency, and in the context of concurrent λ-calculi, such as the GV family of languages ["Good Variation", 20, 23, 40, 64].

Session types have been implemented in various programming languages. We give a detailed overview in section 4, and Orchard and Yoshida [49] provide a complete survey of session type implementations in Haskell.

The main difficulty when implementing session types in most programming languages is *linearity*, *i.e.*, the guarantee that each channel endpoint is used *exactly once*. There are several different approaches to guaranteeing linearity, but the main distinction is between *dynamic* [52, 59, 60] and *static* [41, 42, 56] usage checks. With dynamic checks, using a channel endpoint more than once simply throws a runtime error. With static checks, usage is *somehow* encoded into the type system of the host language usually by encoding the entire linear typing environment into the type system using a parameterised or graded monad. Such encodings are only possible if the type system of the host language is expressive enough. However, such encodings are often quite complex, and result in a trade-off between easy-to-write session types and idiomatic programs.

Moreover, these implementations only focus on the most basic features of session types and often ignore more advanced ones, such as channel delegation or deadlock freedom: Neubauer and Thiemann [44] only provide single session channels; Pucella and Tov [56] provide multiple channels, but only the building blocks for channel delegation; Imai et al. [30] extend Pucella and Tov [56] and provide full delegation. None of these works address deadlock freedom. Lindley and Morris [41] provide an implementation of GV into Haskell building on the work of Polakow [55]. To the best of our knowledge, this is the only work that guarantees deadlock freedom of session types in Haskell, albeit in a simple form. In GV, all programs must have *tree-shaped* process structures. The process structure of a program is an undirected graph, where nodes represent processes, and edges represent the channels connecting them. (We explore this in more detail in section 2.3.) Therefore, deadlock freedom is guaranteed by design: session types rule out deadlocks over a single channel, and the tree-restriction rules out sharing multiple channels between two processes. While Lindley and Morris [41] manage to implement more advanced properties, the tree restriction rules out many interesting programs which have *cyclic* process structure, but are deadlock free.

Recent works by Padovani and Novara [53] and Kokke and Dardha [PGV, 35] integrate *priorities* [32, 51] into functional languages. Priorities are natural numbers that abstractly represent the time at which a communication action happens.

Haskell '21, August 26–27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8615-9/21/08…$15.00
https://doi.org/10.1145/3471874.3472979

Priority-based type systems check that there are *no* cycles in the communication graph. The communication graph is a directed graph where nodes represent dual communication actions, and directed edges represent one action must happen before another. (We explore this in more detail in section 2.4.) Such type systems are *more* expressive, as they allow programs to have *cyclic* process structure, as long as they have an *acyclic* communication graph.

With the above in mind, our research goals are as follows:

Q1 Can we have easy-to-write session types, easy linearity checks and idiomatic code at the same time?

Q2 Can we address not only the main features of session types, but also advanced ones, such as full delegation, recursion, and deadlock freedom of programs with cyclic process structure?

Our `priority-sesh` library answers both questions *mostly* positively. We sidestep the problems with encoding linearity in Haskell by using Linear Haskell [4], which has native support for linear types. The resulting session type library presented in sections 2.2 and 2.3 has both easy-to-write session types, easy linearity checks, and idiomatic code. Moving to **Q2**, the library has full delegation, recursion, and the variant in section 2.3 guarantees deadlock freedom, albeit by restricting the process structure to trees and forests. In section 2.4, we implement another variant which uses priorities to ensure deadlock freedom of programs with cyclic processes structure. The ease-of-writing suffers a little, as the programmer has to manually write priorities, though this isn't a huge inconvenience. Unfortunately, GHC's ability to reason about type-level naturals currently is *not* as powerful as to allow the programmer to easily write priority-polymorphic code, which is required for recursion. Hence, while we address deadlock freedom for cyclic process structures, we do so only for the finite setting.

Contributions. In section 2, we present Priority Sesh, an implementation of deadlock free session types in Linear Haskell which is:

- the *first* implementation of session types to take advantage of Linear Haskell for linearity checking, and producing easy-to-write session types and more idiomatic code;
- the *first* implementation of session types in Haskell to guarantee deadlock freedom of programs with cyclic process structure via *priorities*; and
- the *first* embedding of priorities into an existing mainstream programming language.

In section 3, we:

- present a variant of Priority GV [35]—the calculus upon which Priority Sesh is based—with asynchronous communication and session cancellation following Fowler et al. [20] and *explicit* lower bounds on the sequent, rather than lower bounds inferred from the typing environment; and
- show that Priority Sesh is related to Priority GV via monadic reflection.

2 What is Priority Sesh?

In this section we introduce Priority Sesh in three steps:

- in section 2.1, we build a small library of *linear* or *one-shot channels* based on MVars [54];
- in section 2.2, we use these one-shot channels to build a small library of *session-typed channels* [12]; and
- in section 2.4, we decorate these session types with *priorities* to guarantee deadlock-freedom [35].

It is important to notice that the meaning of linearity in *one-shot channels* differs from linearity in *session channels*. A linear or one-shot channel originates from the linear π-calculus [33, 58], where each endpoint of a channel must be used for *exactly one* send or receive operation, whereas linearity in the context of session-typed channels, it means that each step in the protocol is performed *exactly once*, but the channel itself is used multiple times.

Priority Sesh is written in Linear Haskell [4]. The type $\multimap$ is syntactic sugar for the linear arrow `%1->`. Familiar definitions refer to linear variants packaged with `linear-base`[1] (*e.g.*, *IO*, *Functor*, *Bifunctor*, *Monad*) or with Priority Sesh (*e.g.*, *MVar*).

We colour the Haskell definitions which are a part of Sesh: **red** for functions and constructors; **blue** for types and type families; and **emerald** for priorities and type families acting on priorities.

2.1 One-shot Channels

We start by building a small library of *linear* or *one-shot channels*, *i.e.*, channels that must be use *exactly once* to send or receive a value.

The one-shot channels are at the core of our library, and their efficiency is crucial to the overall efficiency of Priority Sesh. However, we do not aim to present an efficient implementation here, rather we aim to present a compact implementation with the correct behaviour.

Channels. A one-shot channel has two endpoints, $Send_1$ and $Recv_1$, which are two copies of the same *MVar*.

```
newtype Send₁ a = Send₁ (MVar a)
newtype Recv₁ a = Recv₁ (MVar a)

new₁ :: IO (Send₁ a, Recv₁ a)
new₁ = do
  (mvar_s, mvar_r) ← dup2 ⟨$⟩ newEmptyMVar
  return (Send₁ (unur mvar_s), Recv₁ (unur mvar_r))
```

The *newEmptyMVar* function returns an *unrestricted MVar*, which may be used non-linearly, *i.e.*, as many times as one

[1] https://hackage.haskell.org/package/linear-base

wants. The *dup2* function creates two (unrestricted) copies of the *MVar*. The *unur* function casts each *unrestricted* copy to a *linear* copy. Thus, we end up with two copies of an *MVar*, each of which must be used *exactly once*.

We implement $send_1$ and $recv_1$ as aliases for the corresponding *MVar* operations.

```
send₁ :: Send₁ a ⊸ a ⊸ IO ()
send₁ (Send₁ mvar_s) x = putMVar mvar_s x

recv₁ :: Recv₁ a ⊸ IO a
recv₁ (Recv₁ mvar_r) = takeMVar mvar_r
```

The *MVar* operations implement the correct blocking behaviour for asynchronous one-shot channels: the $send_1$ operation is non-blocking, and the $recv_1$ operations blocks until a value becomes available.

Synchronisation. We use $Send_1$ and $Recv_1$ to implement a construct for one-shot synchronisation between two processes, $Sync_1$, which consists of two one-shot channels. To synchronise, each process sends a unit on the one channel, then waits to receive a unit on the other channel.

```
data Sync₁ = Sync₁ (Send₁ ()) (Recv₁ ())

newSync₁ :: IO (Sync₁, Sync₁)
newSync₁ = do (ch_s1, ch_r1) ← new₁
              (ch_s2, ch_r2) ← new₁
              return (Sync₁ ch_s1 ch_r2, Sync₁ ch_s2 ch_r1)

sync₁ :: Sync₁ ⊸ IO ()
sync₁ (Sync₁ ch_s ch_r) = do send₁ ch_s (); recv₁ ch_r
```

Cancellation. We implement *cancellation* for one-shot channels. One-shot channels are created in the linear *IO* monad, so *forgetting* to use a channel results in a complaint from the type-checker. However, it is possible to *explicitly* drop values whose types implement the *Consumable* class, using $consume :: a \multimap ()$. The ability to cancel communications is important, as it allows us to safely throw an exception *without violating linearity*, assuming that we cancel all open channels before doing so.

One-shot channels implement *Consumable* by simply dropping their *MVars*. The Haskell runtime throws an exception when a "thread is blocked on an *MVar*, but there are no other references to the *MVar* so it can't ever continue."[2] Practically, *consumeAndRecv* throws a *BlockedIndefinitelyOnMVar* exception, whereas *consumeAndSend* does not:

```
consumeAndRecv = do
  (ch_s, ch_r) ← new₁
  fork $ return (consume ch_s)
  recv₁ ch_r

consumeAndSend = do u
  (ch_s, ch_r) ← new₁
  fork $ return (consume ch_r)
  send₁ ch_s ()
```

[2] https://downloads.haskell.org/~ghc/9.0.1/docs/html/libraries/base-4.15.0.0/Control-Exception.html#t:BlockedIndefinitelyOnMVar

Where *fork* forks off a new thread using a linear *forkIO*. (In GV, this operation is called *spawn*.)

As the *BlockedIndefinitelyOnMVar* check is performed by the runtime, it'll even happen when a channel is dropped for reasons other than consume, such as a process crashing.

2.2 Session-typed Channels

We use the one-shot channels to build a small library of *session-typed channels* based on the *continuation-passing style* encoding of session types in linear types by Dardha [9], Dardha et al. [12] and in line with other libraries for Scala [59, 60], OCaml [52], and Rust [34].

An Example. Let's look at a simple example of a session-typed channel—a multiplication service, which receives two integers, sends back their product, and then terminates:

```
type MulServer = Recv Int (Recv Int (Send Int End))
type MulClient = Send Int (Send Int (Recv Int End))
```

We define *mulServer*, which acts on a channel of type *MulServer*, and *mulClient*, which acts on a channel of the *dual* type:

```
mulServer (s :: MulServer)
  = do (x, s) ← recv s
       (y, s) ← recv s
       s ← send (x * y, s)
       close s
       return ()

mulClient (s :: MulClient)
  = do s ← send (32, s)
       s ← send (41, s)
       (z, s) ← recv s
       close s
       return z
```

In order to encode the *sequence* of a session type using one-shot types, each action on a session-typed channel returns a channel for the *continuation* of the session—save for *close*, which ends the session. Furthermore, *mulServer* and *mulClient* act on endpoints with *dual* types. Duality is crucial to session types as it ensures that when one process sends, the other is ready to receive, and vice versa. This is the basis for communication safety guaranteed by a session type system.

Channels. We start by defining the *Session* type class, which has an associated type *Dual*. You may think of *Dual* as a type-level function associated with the *Session* class with *one* case for each instance. We encode the various restrictions on duality as constraints on the type class. Each session type must have a dual, which must itself be a session type—*Session* (*Dual* *s*) means the dual of *s* must also implement *Session*. Duality must be *injective*—the annotation *result* → *s* means *result* must uniquely determine *s* and *involutive*—*Dual* (*Dual* *s*) ∼ *s* means *Dual* (*Dual* *s*) must equal *s*. These constraints are all captured by the *Session* class, along with *new* for constructing channels:

```
class (Session (Dual s), Dual (Dual s) ∼ s) ⇒ Session s
  where
```

```
  type Dual s = result | result → s
  new :: IO (s, Dual s)
```

There are three primitive session types: *Send*, *Recv*, and *End*.

```
newtype Send a s = Send (Send₁ (a, Dual s))
newtype Recv a s = Recv (Recv₁ (a, s))
newtype End      = End Sync₁
```

By following Dardha et al. [12], a channel *Send* wraps a one-shot channel $Send_1$ over which we send some value—which is the intended value sent by the session channel, and the channel over which *the communicating partner process* continues the session—it'll make more sense once you read the definition for *send*. A channel *Recv* wraps a one-shot channel $Recv_1$ over which we receive some value and the channel over which *we* continue the session. Finally, an channel *End* wraps a synchronisation.

We define duality for each session type—*Send* is dual to *Recv*, *Recv* is dual to *Send*, and *End* is dual to itself:

```
instance Session s ⇒ Session (Send a s)
  where
    type Dual (Send a s) = Recv a (Dual s)
    new = do (chs, chr) ← new₁
             return (Send chs, Recv chr)
instance Session s ⇒ Session (Recv a s)
  where
    type Dual (Recv a s) = Send a (Dual s)
    new = do (chs, chr) ← new₁
             return (Recv chr, Send chs)
instance Session End
  where
    type Dual End = End
    new = do (chsync1, chsync2) ← newSync₁
             return (End chsync1, End chsync2)
```

The *send* operation constructs a channel for the continuation of the session, then sends one endpoint of that channel, along with the value, over its one-shot channel, and returns the other endpoint:

```
send :: Session s ⇒ (a, Send a s) ⊸ IO s
send (x, Send chs) = do (here, there) ← new
                        send₁ chs (x, there)
                        return here
```

The *recv* and *close* operations simply wrap their corresponding one-shot operations:

```
recv :: Recv a s ⊸ IO (a, s)
recv (Recv chr) = recv₁ chr

close :: End ⊸ IO ()
close (End chsync) = sync₁ chsync
```

Cancellation. We implement session *cancellation* via the *Consumable* class. For convenience, we provide the *cancel* function:

```
cancel :: Session s ⇒ s ⊸ IO ()
cancel s = return (consume s)
```

As with one-shot channels, *consume* simply drops the channel, and relies on the *BlockedIndefinitelyOnMVar* check, which means that *cancelAndRecv* throws an exception and *cancelAndSend* does not:

```
cancelAndRecv = do                 cancelAndSend = do u
  (chs, chr) ← new                   (chs, chr) ← new
  fork $ cancel chs                  fork $ cancel chr
  ((), ()) ← recv chr                () ← send chs ()
  return ()                          return ()
```

These semantics correspond to EGV [20].

Asynchronous Close. We don't always *want* session-end to involve synchronisation. Unfortunately, the *close* operation is synchronous.

An advantage of defining session types via a type class is that its an *open* class, and we can add new primitives whenever. Let's make the unit type, (), a session type:

```
instance Session s ⇒ Session ()
  where
    type Dual () = ()
    new = return ((), ())
```

Units are naturally affine—they contain *zero* information, so dropping them won't harm—and the linear *Monad* class allows you to silently drop unit results of monadic computations. They're ideal for *asynchronous* session end!

Using () allows us to recover the semantics of one-shot channels while keeping a session-typed language for idiomatic protocol specification.

Choice. So far, we've only presented sending, receiving, and synchronisation. It is, however, possible to send and receive *channels* as well as values, and we leverage that to implement most other session types by using these primitives only!

For instance, we can implement *binary* choice by sending/receiving *Either* of two session continuations:

```
type Select s1 s2 = Send (Either (Dual s1) (Dual s2)) ()
type Offer  s1 s2 = Recv (Either s1 s2) ()

selectLeft :: (Session s1) ⇒ Select s1 s2 ⊸ IO s1
selectLeft s = do (here, there) ← new
                  send (Left there, s)
                  return here

offerEither :: Offer s1 s2 ⊸ (Either s1 s2 ⊸ IO a) ⊸ IO a
offerEither s match = do (e, ()) ← recv s; match e
```

Differently from (), we don't have to implement the *Session* class for *Select* and *Offer*. They're already session types!

Recursion. We can write recursive session types by writing them as recursive Haskell types. Unfortunately, we cannot write recursive type synonyms, so we have to use a newtype. For instance, we can write the type for a recursive summation service, which receives numbers until the client indicates they're done, and then sends back the sum. We specify *two* newtypes:

```
newtype SumSrv
  = SumSrv (Offer (Recv Int SumSrv) (Send Int End))
newtype SumCnt
  = SumCnt (Select (Send Int SumCnt) (Recv Int End))
```

We implement the summation server as a recursive function:

```
sumSrv :: Int ⊸ SumSrv ⊸ IO ()
sumSrv tot (SumSrv s) = offerEither s $ λe. case x of
  Left  s → do (x, s) ← recv s; sumSrv (tot + x) s
  Right s → do s ← send (tot, s); close s
```

As *SumSrv* and *SumCnt* are new types, we must provide instances of the *Session* class for them.

```
instance Session SumSrv
  where
    type Dual SumSrv = SumCnt
    new = do (ch_srv, ch_cnt) ← new
             return (SumSrv ch_srv, SumCnt ch_cnt)
```

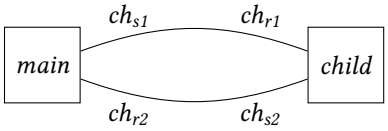
2.3 Deadlock Freedom via Process Structure

The session-typed channels presented in section 2.2 can be used to write deadlocking programs, *e.g.*, by receiving before sending:

```
woops :: IO Void
woops = do (ch_s1, ch_r1) ← new
           (ch_s2, ch_r2) ← new
           fork $ do (void, ()) ← recv ch_r1
                     send (void, ch_s2)
           (void, ()) ← recv ch_r2
           let (void, void_copy) = dup2 void
           send (void, ch_s1)
           return void_copy
```

Counter to what the type says, this program doesn't actually produce an inhabitant of the *uninhabited* type *Void*. Instead, it deadlocks! We'd like to help the programmer avoid such programs.

As discussed in section 1, we can *structurally* guarantee deadlock freedom by ensuring that the *process structure* is always a tree or forest. The process structure of a program is an undirected graph, where nodes represent processes, and edges represent the channels connecting them. For instance, the process structure of *woops* is cyclic:



This restriction works by ensuring that between two processes there is *at most* one (series of) channels over which the two can communicate. As duality rules out deadlocks on any one channel, such configurations must be deadlock free.

We can rule out cyclic process structures by hiding *new*, and only exporting *connect*, which creates a new channel and, *crucially*, immediately passes one endpoint to a new thread:

```
connect :: Session s ⇒
           (s ⊸ IO ()) ⊸ (Dual s ⊸ IO a) ⊸ IO a
connect k_1 k_2 = do (s_1, s_2) ← new; fork (k_1 s_1); k_2 s_2
```

You can view *connect* as the node constructor for a binary process tree. If the programmer *only* uses *connect*, their process structure is guaranteed to be a *tree*. If they also use standalone *fork*, their process structure is a *forest*. Either way, their programs are guaranteed to be deadlock free.

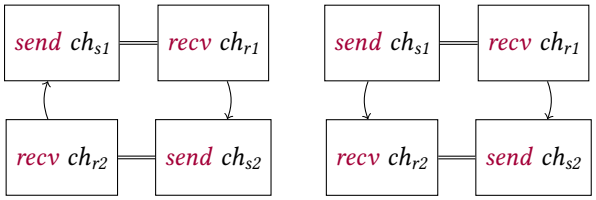
2.4 Deadlock Freedom via Priorities

The strategy for deadlock freedom presented in section 2.3 is simple, but *very* restrictive, since it rules out *all* cyclic communication structures, even the ones which don't deadlock:

```
totallyFine :: IO String
totallyFine = do (ch_s1, ch_r1) ← new
                 (ch_s2, ch_r2) ← new
                 fork $ do (x, ()) ← recv ch_r1
                           send (x, ch_s2)
                 send ("Hiya!", ch_s1)
                 (x, ()) ← recv ch_r2
                 return x
```

This process has *exactly the same* process structure as *woops*, but it's totally fine, and returns `"Hiya!"` as you'd expect. We'd like to enable the programmer to write such programs while still guaranteeing their programs don't deadlock.

As discussed in section 1, there is another way to rule out deadlocks—by using *priorities*. Priorities are an approximation of the *communication graph* of a program. The communication graph of a program is a *directed graph* where nodes represent *actions on channels*, and directed edges represent that one action happens before the other. Dual actions are connected with double undirected edges. (You may consider the graph contracted along these edges.) If the communication graph is cyclic, the program deadlocks. The communication graphs for *woops* and *totallyFine* are as follows:



If the communication graph is acyclic, then we can assign each node a number such that directed edges only ever point to nodes with *bigger* numbers. For instance, for *totallyFine* we can assign the number 0 to *send* ch_{s1} and *recv* ch_{r2}, and 1 to *recv* ch_{r2} and *send* ch_{s2}. These numbers are *priorities*.

In this section, we present a type system in which *priorities* are used to ensure deadlock freedom, by tracking the time a process starts and finishes communicating using a graded monad [21, 48]. The bind operation registers the order of its actions in the type, requiring the sequentiality of their duals.

Priorities. The priorities assigned to *communication actions* are always natural numbers, which represent, *abstractly*, at which time the action happens. When tracking the start and finish times of a program, however, we also use $\bot$ and $\top$ for programs which don't communicate. These are used as the identities for $\sqcap$ and $\sqcup$ in lower and upper bounds, respectively. We let o range over natural numbers, p over *lower bounds*, and q over *upper bounds*.

```
data Priority = ⊥ | Nat | ⊤
```

We define strict inequality ($<$), minimum ($\sqcap$), and maximum ($\sqcup$) on priorities as usual.

Channels. We define $Send^o$, $Recv^o$, and End^o, which decorate the *raw* sessions from section 2.2 with the priority o of the communication action, *i.e.*, it denoted when the communication happens. Duality (*Dual*) preserves these priorities. These are implemented exactly as in section 2.2.

The Communication Monad. We define a graded monad $Sesh^q_p$, which decorates *IO* with a lower bound p and an upper bound q on the priorities of its communication actions, *i.e.*, if you run the monad, it denotes when communication begins and ends.

```
newtype Sesh^q_p a = Sesh { runSeshIO :: IO a }
```

The monad operations for $Sesh^q_p$ merely wrap those for *IO*, hence trivially obeys the monad laws.

The *ireturn* function returns a *pure* computation—the type $Sesh^\bot_\top$ guarantees that all communications happen between $\top$ and $\bot$, hence there can be no communication at all.

```
ireturn :: a ⊸ Sesh^⊥_⊤ a
ireturn x = Sesh $ return x
```

The $\gg$ operator sequences two actions with types $Sesh^q_p$ and $Sesh^{q'}_{p'}$, and requires $q < p'$, *i.e.*, the first action must have finished before the second starts. The resulting action has lower bound $p \sqcap p'$ and upper bound $q \sqcup q'$.

```
(≫) :: (q < p') ⇒ Sesh^q_p a ⊸ (a ⊸ Sesh^{q'}_{p'} b) ⊸ Sesh^{q⊔q'}_{p⊓p'} b
mx ≫ mf = Sesh $ runSeshIO mx ≫ λx. runSeshIO (mf x)
```

In what follows, we implicitly use $\gg$ with do-notation. This can be accomplished in Haskell using `RebindableSyntax`.

We define decorated variants of the concurrency and communication primitives: *send*, *recv*, and *close* each perform a communication action with some priority o, and return a computation of type $Sesh^o_o$, *i.e.*, with *exact* bounds; *new* and *cancel* don't perform any communication action, and so return a *pure* computation of type $Sesh^\bot_\top$; *fork* takes a computation which performs communication actions as an argument, forks it off into a separate thread, and masks the upper bound in its return type.

```
new    :: Session s ⇒ Sesh^⊥_⊤ (s, Dual s)
fork   :: Sesh^q_p () ⊸ Sesh^⊥_p ()
cancel :: Session s ⇒ s ⊸ Sesh^⊥_⊤ ()
send   :: Session s ⇒ (a, Send^o a s) ⊸ Sesh^o_o s
recv   :: Recv^o a s ⊸ Sesh^o_o (a, s)
close  :: End^o ⊸ Sesh^o_o ()
```

From these, we derive decorated choice, as before:

```
type Select^o s₁ s₂ = Send^o (Either (Dual s₁) (Dual s₂)) ()
type Offer^o  s₁ s₂ = Recv^o (Either s₁ s₂) ()

selectLeft  :: (Session s₁) ⇒ Select^o s₁ s₂ ⊸ Sesh^o_o s₁
selectRight :: (Session s₂) ⇒ Select^o s₁ s₂ ⊸ Sesh^o_o s₂
offerEither :: (o < p) ⇒ Offer^o s₁ s₂ ⊸
               (Either s₁ s₂ ⊸ Sesh^q_p a) ⊸ Sesh^{o⊔q}_{o⊓p} a
```

Safe IO. We can use a trick from the *ST* monad [38] to define a "pure" variant of *runSesh*, which encapsulates all use of IO within the $Sesh^q_p$ monad. The idea is to index the $Sesh^q_p$ and every session type constructor with an extra type parameter *tok*, which we'll call the *session token*:

```
send :: Session s ⇒ (a, Send^o tok a s) ⊸ Sesh^o_o tok s
recv :: Recv^o tok a s ⊸ Sesh^o_o tok (a, s)
close :: End^o tok ⊸ Sesh^o_o tok ()
```

The session token should never be instantiated, except by *runSesh*, and every action under the same call to *runSesh* should use the same type variable *tok* as its session token:

```
runSesh :: (∀tok. Sesh^q_p tok a) ⊸ a
runSesh x = unsafePerformIO (runSeshIO x)
```

This ensures that none of the channels created in the session can escape out of the scope of *runSesh*.

We implement this encapsulation in `priority-sesh`, though the session token is the first argument, preceding the priority bounds.

Recursion. We could implement recursive session via priority-polymorphic types, or via priority-shifting [53]. For instance, we could give the *summation service* from section 2.2 the following type:

```
newtype SumSrv^o
  = SumSrv (Offer^o (Recv^{o+1} Int (SumSrv^{o+2}))
                    (Send^{o+1} Int (End^{o+2})))
```

We'd then like to assign *sumSrv* the following type:

```
sumSrv :: Int ⊸ SumSrv^o ⊸ Sesh^⊤_o ()
sumSrv tot (SumSrv s) = offerEither s $ λe. case x of
  Left  s → do (x, s) ← recv s; sumSrv (tot + x) s
  Right s → do s ← send (tot, s); weaken (close s)
```

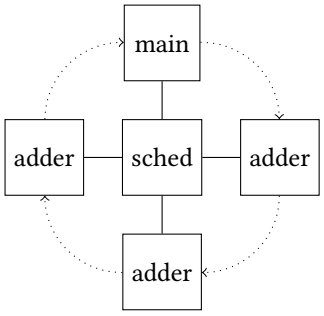
The upper bound for a recursive call should be $\top$, which ensures that recursive calls are only made in *tail* position [3, 22]. The recursive call naturally has upper bound $\top$. However, the *close* operation happens at some *concrete* priority $o + n$, which needs to be raised to $\top$, so we'd have to add a primitive $weaken : Sesh^q_p\ a \multimap Sesh^\top_p\ a$.

Unfortunately, writing such priority-polymorphic code relies heavily on GHC's ability to reason about type-level naturals, and GHC rejects *sumSrv* complaining that it cannot verify that $o < o + 1$, $o + 1 < o + 2$, *etc.* There's several possible solutions for this:

1. We could embrace the Hasochism [39], and provide GHC with explicit evidence, though this would make `priority-sesh` more difficult to use.
2. We could delegate *some* of these problems to a GHC plugin such as `type-nat-solver`[3] or `ghc-typelits-presburger`[4]. Unfortunately, $\sqcap$ and $\sqcup$ are beyond Presburger arithmetic, and `type-nat-solver` has not been maintained in recent years.
3. We could attempt to write type families which reduce in as many cases as possible. Unfortunately, a restriction in closed type families [16, §6.1] prevents us from checking *exactly these cases*.

Currently, the prioritised sessions don't support recursion, and implementing one of these solutions is future work.

Cyclic Scheduler. Dardha and Gay [10] and Kokke and Dardha [36] use a *finite* cyclic scheduler as an example. The cyclic scheduler has the following process structure, with the flow of information indicated by the dotted arrows:



We start by defining the types of the channels which connect each client process to the scheduler:

```
type SR^{o2}_{o1} a = Send^{o1} a (Recv^{o2} a ())
type RS^{o2}_{o1} a = Dual (SR^{o2}_{o1} a)
```

We then define the scheduler itself, which forwards messages from one process to the next in a cycle:

```
sched :: RS^7_0 a ⊸ SR^2_1 a ⊸ SR^4_3 a ⊸ SR^6_5 a ⊸ Sesh^7_0 ()
sched s1 s2 s3 s4 = do
  (x, s1) ← recv s1
  s2 ← send (x, s2); (x, ()) ← recv s2
  s3 ← send (x, s3); (x, ()) ← recv s3
  s4 ← send (x, s4); (x, ()) ← recv s4
  send (x, s1)
```

Finally, we define the *adder* and the *main* processes. The *adder* adds one to the value it receives, and the *main* process initiates the cycle and receives the result:

```
adder :: (o1 < o2) ⇒ RS^{o2}_{o1} Int ⊸ Sesh^{o2}_{o1} ()
adder s = do (x, s) ← recv s; send (x + 1, s)

main :: (o1 < o2) ⇒ Int ⊸ SR^{o2}_{o1} Int ⊸ Sesh^{o2}_{o1} Int
main x s = do ; s ← send (x, s); (x, ()) ← recv s; ireturn x
```

While the process structure of the cyclic scheduler *as presented* isn't cyclic, nothing prevents the user from adding communications between the various client processes, or from removing the scheduler and having the client processes communicate *directly* in a ring.

3 Relation to Priority GV

The `priority-sesh` library is based on a variant of Priority GV [35], which differs in three ways:

1. it marks lower bounds *explicitly* on the sequent, rather than implicitly inferring them from the typing environment;
2. it collapses the isomorphic types for session end, $\mathbf{end}^o_!$ and $\mathbf{end}^o_?$, into $\mathbf{end}^o$;
3. it is extended with asynchronous communication and session cancellation following Fowler et al. [20].

These changes preserve subject reduction and progress properties, and give us *tighter* bounds on priorities. To see why, note that PCP [10] and PGV [35] use the *smallest* priority in the typing environment as an approximation for the lower

[3] https://github.com/yav/type-nat-solver
[4] https://hackage.haskell.org/package/ghc-typelits-presburger

bound. Unfortunately, this *underestimates* the lower bound in the rules T-Var and T-Lam (check fig. 1). These rules type *values*, which are pure and could have lower bound $\top$, but the smallest priority in their typing environment is not necessarily $\top$.

Priority GV. We briefly revisit the syntax and type system of PGV, but a full discussion of PGV is out of scope for this paper. For a discussion of the *synchronous* semantics for PGV, and the proofs of subject reduction, progress, and deadlock freedom, please see Kokke and Dardha [35]. For a discussion of the *asynchronous* semantics and session cancellation, please see Fowler et al. [20].

As in section 2.4, we let o range over priorities, which are natural numbers, and p and q over priority bounds, which are either natural numbers, $\top$, or $\bot$.

PGV is based on the standard linear λ-calculus with product types ($\cdot \times \cdot$), sum types ($\cdot + \cdot$), and their units ($\mathbf{1}$ and $\mathbf{0}$). Linear functions ($\cdot \multimap^q_p \cdot$) are annotated with priority bounds which tell us–when the function is applied–when communication begins and ends.

Types and session types are defined as follows:

$$\begin{array}{lcl} S & ::= & !^o T.S \mid ?^o T.S \mid \mathbf{end}^o \\ T, U & ::= & T \times U \mid \mathbf{1} \mid T + U \mid \mathbf{0} \mid T \multimap^q_p U \mid S \end{array}$$

The types $!^o T.S$ and $?^o T.S$ mean "send" and "receive", respectively, and $\mathbf{end}^o$ means, well, session end.

The term language is the standard linear λ-calculus extended with concurrency primitives K:

$$\begin{array}{lcl} L, M, N & & \\ & ::= & x \mid K \mid \lambda x.M \mid M\ N \\ & \mid & () \mid M; N \\ & \mid & (M, N) \mid \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N \\ & \mid & \mathbf{absurd}\ M \\ & \mid & \mathbf{inl}\ M \mid \mathbf{inr}\ M \mid \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} \\ K & ::= & \mathbf{new} \mid \mathbf{fork} \mid \mathbf{send} \mid \mathbf{recv} \mid \mathbf{close} \end{array}$$

The concurrency primitives are uninterpreted in the term language. Rather, they are interpreted in a configuration language based on the π-calculus, which we omit from this paper (see Kokke and Dardha [35]).

We present the typing rules for PGV in fig. 1. A sequent $\Gamma \vdash^q_p M : T$ should be read as "M is well-typed PGV program with type T in typing environment Γ, and when run it starts communicating at time p and stops at time q."

Monadic Reflection. The graded monad $Sesh^q_p$ arises from the *monadic reflection* [17] of the typing rules in fig. 1. Monadic reflection is a technique for translating programs in an effectful language to *monadic* programs in a pure language. For instance, Filinski [17] demonstrates the reflection from programs of type T in a language with exceptions and handlers to programs of type $T + \mathbf{exn}$ in a pure language where $\mathbf{exn}$ is the type of exceptions.

We translate programs from PGV to Haskell programs in the $Sesh^q_p$ monad. First, let's look at the translation of types:

$$\begin{array}{lll} [\![T \multimap^q_p U]\!] = [\![T]\!] \multimap Sesh^q_p\ [\![U]\!] & [\![\mathbf{1}]\!] = () \\ [\![!^o T.S]\!] = Send^o\ [\![T]\!]\ [\![S]\!] & [\![T \times U]\!] = ([\![T]\!], [\![U]\!]) \\ [\![?^o T.S]\!] = Recv^o\ [\![T]\!]\ [\![S]\!] & [\![\mathbf{0}]\!] = Void \\ [\![\mathbf{end}^o]\!] = End^o & [\![T + U]\!] = Either\ [\![T]\!]\ [\![U]\!] \end{array}$$

Now, let's look at the translation of terms. A term of type T with lower bound p and upper bound q is translated to a Haskell program of type $Sesh^q_p\ [\![T]\!]$:

$$\begin{array}{ll} [\![x]\!] & = ireturn\ x \\ [\![\lambda x.L]\!] & = ireturn\ (\lambda x.\ [\![L]\!]) \\ [\![K]\!] & = ireturn\ [\![K]\!] \\ [\![L\ M]\!] & = [\![L]\!] \gg\!= \lambda f.\ [\![M]\!] \gg\!= \lambda x.\ f\ x \\ [\![()]\!] & = ireturn\ () \\ [\![\mathbf{let}\ () = L\ \mathbf{in}\ M]\!] & = [\![L]\!] \gg\!= \lambda().\ M \\ [\![(L, M)]\!] & = [\![L]\!] \gg\!= \lambda x.\ [\![M]\!] \gg\!= \lambda y.\ ireturn\ (x, y) \\ [\![\mathbf{let}\ (x, y) = L\ \mathbf{in}\ M]\!] & = [\![L]\!] \gg\!= \lambda(x, y).\ [\![M]\!] \\ [\![\mathbf{absurd}\ L]\!] & = [\![L]\!] \gg\!= \lambda x.\ absurd\ x \\ [\![\mathbf{inl}\ L]\!] & = [\![L]\!] \gg\!= \lambda x.\ ireturn\ (Left\ x) \\ [\![\mathbf{inr}\ L]\!] & = [\![L]\!] \gg\!= \lambda x.\ ireturn\ (Right\ x) \\ [\![\mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\}]\!] = & \\ \quad [\![L]\!] \gg\!= \lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \{Left\ x \to [\![M]\!]; Right\ y \to [\![N]\!]\} & \end{array}$$

We translate the communication primitives from PGV to those with the same name in `priority-sesh`, with some minor changes in the translations of **new** and **fork**, where PGV needs some unit arguments to create thunks in PGV, as it's call-by-value, which aren't needed in Haskell:

$$\begin{array}{l} [\![\mathbf{new} : \mathbf{1} \multimap^{\bot}_{\top} S \times \overline{S}]\!] \\ \quad = \lambda().\ new :: () \multimap ([\![S]\!], [\![(Dual\ S)]\!]) \\ [\![\mathbf{fork} : (\mathbf{1} \multimap^q_p \mathbf{1}) \multimap^{\bot}_{\top} \mathbf{1}]\!] \\ \quad = \lambda k.\ fork\ (k\ ()) :: (() \multimap Sesh^q_p\ ()) \multimap Sesh^{\bot}_{\top}\ () \end{array}$$

The rest of PGV's communication primitives line up exactly with those of `priority-sesh`:

$$\begin{array}{l} [\![\mathbf{send} : T \times !^o T.S \multimap^o_o S]\!] \\ \quad = send :: Session\ [\![S]\!] \Rightarrow ([\![T]\!], Send^o\ [\![T]\!]\ [\![S]\!]) \multimap Sesh^o_o\ [\![S]\!] \\ [\![\mathbf{recv} : ?^o T.S \multimap^o_o T \times S]\!] \\ \quad = recv :: Recv^o\ [\![T]\!]\ [\![S]\!] \multimap Sesh^o_o\ ([\![T]\!], [\![S]\!]) \\ [\![\mathbf{close} : \mathbf{end}^o \multimap^o_o \mathbf{1}]\!] \\ \quad = close :: End^o \multimap Sesh^o_o\ () \\ [\![\mathbf{cancel} : S \multimap^{\bot}_{\top} \mathbf{1}]\!] \\ \quad = cancel :: Session\ [\![S]\!] \Rightarrow [\![S]\!] \multimap Sesh^{\bot}_{\top}\ () \end{array}$$

These two translations, on types and terms, comprise a *monadic reflection* from PGV into `priority-sesh`, which preserves typing. We state this theorem formally, using $\Gamma \vdash x :: a$ to mean that the Haskell program x has type a in typing environment Γ:

Theorem 3.1. *If* $\Gamma \vdash^q_p M : T$, *then* $[\![\Gamma]\!] \vdash [\![M]\!] :: Sesh^q_p\ [\![T]\!]$.

Proof. Figure 2 presents the translation from typing derivations in PGV to abbreviated typing derivations in Haskell with `priority-sesh`. □

Static Typing Rules. $\boxed{\Gamma \vdash^{q}_{p} M : T}$

$$\textsc{T-Var}\ \frac{}{x : T \vdash^{\perp}_{\top} x : T} \qquad \textsc{T-Lam}\ \frac{\Gamma, x : T \vdash^{q}_{p} M : U}{\Gamma \vdash^{\perp}_{\top} \lambda x.M : T \multimap^{q}_{p} U} \qquad \textsc{T-Const}\ \frac{}{\varnothing \vdash^{\perp}_{\top} K : T}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash^{q}_{p} M : T \multimap^{q''}_{p''} U \qquad \Delta \vdash^{q'}_{p'} N : T \qquad q < p' \qquad q' < p''}{\Gamma, \Delta \vdash^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''} M\ N : U}$$

$$\textsc{T-Unit}\ \frac{}{\varnothing \vdash^{\perp}_{\top} () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma \vdash^{q}_{p} M : \mathbf{1} \qquad \Delta \vdash^{q'}_{p'} N : T \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ () = M\ \mathbf{in}\ N : T}$$

$$\textsc{T-Pair}\ \frac{\Gamma \vdash^{q}_{p} M : T \qquad \Delta \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} (M, N) : T \times U}$$

$$\textsc{T-LetPair}\ \frac{\Gamma \vdash^{q}_{p} M : T \times T' \qquad \Delta, x : T, y : T' \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ (x, y) = M\ \mathbf{in}\ N : U}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash^{q}_{p} M : T}{\Gamma \vdash^{q}_{p} \mathbf{inl}\ M : T + U} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash^{q}_{p} M : T}{\Gamma \vdash^{q}_{p} \mathbf{inr}\ M : T + U}$$

$$\textsc{T-CaseSum}\ \frac{\Gamma \vdash^{q}_{p} L : T + T' \qquad \Delta, x : T \vdash^{q'}_{p'} M : U \qquad \Delta, y : T' \vdash^{q'}_{p'} N : U \qquad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcup p'} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U} \qquad \textsc{T-Absurd}\ \frac{\Gamma \vdash^{q}_{p} M : \mathbf{0}}{\Gamma \vdash^{q}_{p} \mathbf{absurd}\ M : T}$$

Type Schemas for Constants. $\boxed{K : T}$

$$\mathbf{new} : \mathbf{1} \multimap^{\perp}_{\top} S \times \overline{S} \qquad \mathbf{fork} : (\mathbf{1} \multimap^{q}_{p} \mathbf{1}) \multimap^{\perp}_{\top} \mathbf{1} \qquad \mathbf{cancel} : S \multimap^{\perp}_{\top} \mathbf{1}$$

$$\mathbf{send} : T \times !^{o}T.S \multimap^{o}_{o} S \qquad \mathbf{recv} : ?^{o}T.S \multimap^{o}_{o} T \times S \qquad \mathbf{close} : \mathbf{end}^{o} \multimap^{o}_{o} \mathbf{1}$$

Figure 1. Typing rules for Priority GV.

4 Related Work

Session Types in Haskell. Orchard and Yoshida [49] discuss various approaches to implementing session types in Haskell. Their overview is reproduced below:

- Neubauer and Thiemann [44] give an encoding of first-order single-channel session-types with recursion;
- Using *parameterised monads*, Pucella and Tov [56] provide multiple channels, recursion, and some building blocks for delegation, but require manual manipulation of a session typing environment;
- Sackman and Eisenbach [57] provide an alternate approach where session types are constructed via a value-level witnesses;
- Imai et al. [30] extend Pucella and Tov [56] with delegation and a more user-friendly approach to handling multiple channels;
- Orchard and Yoshida [50] use an embedding of effect systems into Haskell via graded monads based on a formal encoding of session-typed π-calculus into PCF with an effect system;
- Lindley and Morris [41] provide a *finally tagless* embedding of the GV session-typed functional calculus into Haskell, building on a linear λ-calculus embedding due to Polakow [55].

With respect to linearity, all works above—except Neubauer and Thiemann [44]—guarantee linearity by encoding a linear typing environment in the Haskell type system, which leads to a trade-off between having easy-to-write session types and having idiomatic programs. We side-step this trade-off by relying on Linear Haskell to check linearity. Furthermore, our implementation supports all relevant features, including multiple channels, full delegation, recursion, and more idiomatic code.

With respect to deadlock freedom, none of the works above—except Lindley and Morris [41]—guarantee deadlock freedom. However, Lindley and Morris [41] guarantee deadlock freedom *structurally*, by implementing GV. As discussed in section 1, structure-based deadlock freedom is more restrictive than priority-based deadlock freedom, as it restricts communication graphs to *trees*, whereas the priority-based approach allows programs to have *cyclic* process structures.

Orchard and Yoshida [49] summarise the capabilities of the various implementations of session types in Haskell in a table, which we adapted in table 1 by adding columns for the various versions of `priority-sesh`. In general, you may read ✓ as "Kinda" and ✓ as a resounding "Yes!" For instance, Pucella and Tov [56] only provide *partial* delegation, Neubauer and Thiemann [44], Pucella and Tov [56], and Lindley and Morris [41] still need to use combinators instead of standard Haskell application, abstraction, or variables in

$$\frac{}{x : T \vdash^{\bot}_{\top} x : T} \;=\; \frac{x :: [\![T]\!] \vdash x :: [\![T]\!]}{\mathit{ireturn}\ x :: \mathit{Sesh}^{\bot}_{\top}\ [\![T]\!]}$$

$$\frac{\Gamma, x : T \vdash^{q}_{p} L : U}{\Gamma \vdash^{\bot}_{\top} \lambda x.L : T \multimap^{q}_{p} U} \;=\; \frac{[\![\Gamma]\!], x :: [\![T]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ [\![U]\!]}{\mathit{ireturn}\ (\lambda x.\ [\![L]\!]) :: \mathit{Sesh}^{\bot}_{\top}\ ([\![T]\!] \multimap \mathit{Sesh}^{q}_{p}\ [\![U]\!])}$$

$$\frac{}{\varnothing \vdash^{\bot}_{\top} K : T} \;=\; \frac{}{\mathit{ireturn}\ [\![K]\!] :: \mathit{Sesh}^{\bot}_{\top}\ [\![T]\!]}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \multimap^{q''}_{p''} U \quad \Delta \vdash^{q'}_{p'} M : T \quad q < p' \quad q' < p''}{\Gamma, \Delta \vdash^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''} L\ M : U} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ ([\![T]\!] \multimap \mathit{Sesh}^{q''}_{p''}\ [\![U]\!]) \quad [\![\Delta]\!] \vdash [\![M]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![T]\!]}{[\![L]\!] \gg\!= \lambda f.\ [\![M]\!] \gg\!= \lambda x.\ f\ x :: (q < p', q' < p'') \Rightarrow \mathit{Sesh}^{q \sqcup q' \sqcup q''}_{p \sqcap p' \sqcap p''}\ [\![U]\!]}$$

$$\frac{}{\varnothing \vdash^{\bot}_{\top} () : \mathbf{1}} \;=\; \frac{}{\mathit{ireturn}\ () :: \mathit{Sesh}^{\bot}_{\top}\ ()}$$

$$\frac{\Gamma \vdash^{q}_{p} L : \mathbf{1} \quad \Delta \vdash^{q'}_{p'} M : T \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ () = L\ \mathbf{in}\ M : T} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ () \quad [\![\Delta]\!] \vdash [\![M]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![T]\!]}{[\![L]\!] \gg\!= \lambda().\ M :: (p < q') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ [\![T]\!]}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \quad \Delta \vdash^{q'}_{p'} M : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} (L, M) : T \times U} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ [\![T]\!] \quad [\![\Delta]\!] \vdash [\![M]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![U]\!]}{[\![L]\!] \gg\!= \lambda x.\ [\![M]\!] \gg\!= \lambda y.\ \mathit{ireturn}\ (x, y) :: (q < p') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ ([\![T]\!], [\![U]\!])}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T \times T' \quad \Delta, x : T, y : T' \vdash^{q'}_{p'} M : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcap p'} \mathbf{let}\ (x, y) = L\ \mathbf{in}\ M : U} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ ([\![T]\!], [\![T']\!]) \quad [\![\Delta]\!], x :: [\![T]\!], y :: [\![T']\!] \vdash [\![M]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![U]\!]}{[\![L]\!] \gg\!= \lambda(x, y).\ [\![M]\!] :: (q < p') \Rightarrow \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ [\![U]\!]}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T}{\Gamma \vdash^{q}_{p} \mathbf{inl}\ L : T + U} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ [\![T]\!]}{[\![\Gamma]\!] \vdash [\![L]\!] \gg\!= \lambda x.\ \mathit{ireturn}\ (\mathit{Left}\ x) :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ [\![T]\!]\ [\![U]\!])}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T}{\Gamma \vdash^{q}_{p} \mathbf{inr}\ L : T + U} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ [\![U]\!]}{[\![\Gamma]\!] \vdash [\![L]\!] \gg\!= \lambda x.\ \mathit{ireturn}\ (\mathit{Right}\ x) :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ [\![T]\!]\ [\![U]\!])}$$

$$\frac{\Gamma \vdash^{q}_{p} L : T + T' \quad \Delta, x : T \vdash^{q'}_{p'} M : U \quad \Delta, y : T' \vdash^{q'}_{p'} N : U \quad q < p'}{\Gamma, \Delta \vdash^{q \sqcup q'}_{p \sqcup p'} \mathbf{case}\ L\ \{\mathbf{inl}\ x \mapsto M;\ \mathbf{inr}\ y \mapsto N\} : U} \;=$$

$$\frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ (\mathit{Either}\ [\![T]\!]\ [\![T']\!]) \quad [\![\Delta]\!] \vdash x :: [\![T]\!] \vdash [\![M]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![U]\!] \quad [\![\Delta]\!] \vdash y :: [\![T']\!] \vdash [\![N]\!] :: \mathit{Sesh}^{q'}_{p'}\ [\![U]\!]}{[\![\Gamma]\!], [\![\Delta]\!] \vdash [\![L]\!] \gg\!= \lambda x.\ \mathbf{case}\ x\ \{\mathit{Left}\ x \to [\![M]\!]; \mathit{Right}\ y \to [\![N]\!]\} :: \mathit{Sesh}^{q \sqcup q'}_{p \sqcap p'}\ [\![U]\!]}$$

$$\frac{\Gamma \vdash^{q}_{p} L : \mathbf{0}}{\Gamma \vdash^{q}_{p} \mathbf{absurd}\ L : T} \;=\; \frac{[\![\Gamma]\!] \vdash [\![L]\!] :: \mathit{Sesh}^{q}_{p}\ \mathit{Void}}{[\![\Gamma]\!] \vdash [\![L]\!] \gg\!= \lambda x.\ \mathit{absurd}\ x :: \mathit{Sesh}^{q}_{p}\ [\![T]\!]}$$

Figure 2. Translation from Priority GV to Sesh preserves types.

some places, and Neubauer and Thiemann [44] is only deadlock free on the technicality that they don't support multiple channels.

Session Types in other Programming Languages. Session types have been integrated in other programming language paradigms. Jespersen et al. [31], Padovani [52], Scalas and Yoshida [60] integrate *binary* session types in the *native* host language, without language extensions; this to

Table 1. Capabilities of various implementations of session types in Haskell [adapted from 49].

							`priority-sesh`		
	NT04	PT08	SE08	IYA10	OY16	LM16	section 2.2	section 2.3	section 2.4
Recursion	✓	✓	✓	✓	✓		✓	✓	
Delegation		✓	✓	✓	✓	✓	✓	✓	✓
Multiple channels		✓	✓	✓	✓	✓	✓	✓	✓
Idiomatic code	✓	✓		✓	✓	✓	✓	✓	✓
Easy-to-write session types	✓	✓	✓		✓	✓	✓	✓	✓
Deadlock freedom	✓					✓		✓	✓
via process structure	✓					✓		✓	
via priorities									✓

avoid hindering session types use in practice. To obtain this integration of session types without extensions Padovani [52], Scalas and Yoshida [60]) combine *static* typing of input and output actions with *runtime* checking of linearity of channel usage.

Implementations of *multiparty* session types (MPST) are less common than binary implementations. Scalas et al. [59] integrate MPST in Scala building upon Scalas and Yoshida [60] and a continuation-passing style encoding of session types into linear types Dardha et al. [11]. There are several works on implementations of MPST in Java: Sivaramakrishnan et al. [61] implement MPST leveraging an extension of Java with session primitives; Hu and Yoshida [29] develops a MPST-based API generation for Java leveraging CFSMs by Brand and Zafiropulo [7]; and Kouzapas et al. [37] implement session types in the form of *typestates* in Java. Demangeon et al. [13] implement MPST in Python and Fowler [18], Neykova and Yoshida [46] in Erlang, focusing on purely dynamic MPST verification via runtime monitoring. Neykova et al. [45], Neykova and Yoshida [47] extend the work by Demangeon et al. [13] with actors and timed specifications. Lopez et al. [43] adopt a dependently-typed MPST theory to verify MPI programs.

Session Types, Linear Logic and Deadlock Freedom. The main line of work regarding deadlock freedom in session-typed systems is that of Curry-Howard correspondences with linear logic [25]. Caires and Pfenning [8] defined a correspondence between session types and dual intuitionistic linear logic and Wadler [64] between session types and classical linear logic. These works guarantee deadlock freedom *by design* as the communication structures are restricted to trees and due to the *cut* rule, processes share *only* one channel between them. Dardha and Gay [10] extend Wadler [64] with *priorities* following Kobayashi [32], Padovani [51], thus allowing processes to share more than one channel in parallel, while guaranteeing deadlock freedom. Balzer et al. [2] introduce *sharing* and guarantee deadlock freedom via priorities. All the above works deal with deadlock freedom in a session-typed π-calculus. With regards to function languages, the original works on GV [23, 24] did not guarantee deadlock freedom. This was later addressed by Lindley and Morris [40], Wadler [65] via syntactic restrictions where communication once again follows a tree structure. Kokke and Dardha [35] introduce PGV–Priority GV, by following Dardha and Gay [10] and allowing for more flexible programming in GV. Fowler et al. [19] present Hypersequent GV (HGV), a core calculus for functional programming with session types that enjoys deadlock freedom, confluence, and strong normalisation.

Other works on deadlock freedom in session-typed systems include the works by Dezani-Ciancaglini et al. [15], where deadlock freedom is guaranteed by allowing only one active session at a time and by Dezani-Ciancaglini et al. [14], where priorities are used for correct interleaving of channels. Honda et al. [28] guarantee deadlock freedom *within a single* session of MPST, but not for session interleaving. Kokke [34] guarantees deadlock freedom of session types in Rust by enforcing a tree structure of communication actions.

5 Discussion and Future Work

We presented `priority-sesh`, an implementation of deadlock-free session types in Linear Haskell. Using Linear Haskell allows us to check linearity—or more accurately, have linearity guaranteed for us—without relying on complex type-level machinery. Consequently, we have easy-to-write session types and idiomatic code—in fact, probably *the most* idiomatic code when compared with previous work, though in fairness, all previous work predates Linear Haskell. Unfortunately, there are some drawbacks to using Linear Haskell. Most importantly, Linear Haskell is not very mature at this stage. For instance:

- Anonymous functions are assumed to be unrestricted rather than linear, meaning anonymous functions must be factored out into a let-binding or where-clause with *at least* a minimal type signature such as $\_ \multimap \_$.
- There is no integration with `base` or popular Haskell packages, and given that `LinearTypes` is an extension, there likely won't be for quite a while. There's `linear-base`, which provides linear variants of many

of the constructs in `base`. However, `linear-base` relies heavily on `unsafeCoerce`, which, ironically, may affect Haskell's performance.
- Generally, there is little integration with the Haskell ecosystem, *e.g.*, one other contribution we made are the formatting directives for Linear Haskell in lhs2TEX [1].

However, we believe that many of these drawbacks will disappear as the Linear Haskell ecosystem matures.

Our work also provides a library which guarantees deadlock freedom via *priorities*, which allows for more flexible typing than previous work on deadlock freedom via a tree process structure.

In the future, we plan to address the issue of priority-polymorphic code and recursion session types in our implementation. (While the versions of our library in sections 2.2 and 2.3 support recursion, that is not yet the case for the priority-based version in section 2.4.) This is a challenging task, as it requires complex reasoning about type-level naturals. We outlined various approaches in section 2.4. However, an alternative we would like to investigate, would be to implement priority-sesh in Idris2 [5, 6], which supports both linear types and complex type-level reasoning.

Acknowledgments

We thank Simon Fowler and April Gonçalves for comments on the manuscript. This work is supported by the EU HORIZON 2020 MSCA RISE project 778233 "Behavioural Application Program Interfaces" (BehAPI).

References

[1] Accessed: 2021-08-06. lhs2tex: Preprocessor for typesetting Haskell sources with LaTeX. https://hackage.haskell.org/package/lhs2tex.

[2] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Proc. of ESOP (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 611–639.

[3] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. 2014. On Duality Relations for Session Types. In *Trustworthy Global Computing*. Springer Berlin Heidelberg, 51–66. https://doi.org/10.1007/978-3-662-45917-1_4

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. of POPL* 2 (2018), 1–29. https://doi.org/10.1145/3158093

[5] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[6] Edwin Brady. 2017. Type-Driven Development of Concurrent Communicating Systems. *Computer Science* 18, 3 (2017), 219. https://doi.org/10.7494/csci.2017.18.3.1413

[7] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (April 1983), 323–342. https://doi.org/10.1145/322374.322380

[8] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[9] Ornela Dardha. 2016. *Type Systems for Distributed Programs: Components and Sessions*. Atlantis Studies in Computing, Vol. 7. Springer / Atlantis Press. https://doi.org/10.2991/978-94-6239-204-5

[10] Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *Proc. of FoSSaCS (LNCS, Vol. 10803)*. Springer, 91–109.

[11] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *Proc. of PPDP*. ACM, 139–150.

[12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002 Extended version of [11].

[13] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical Interruptible Conversations: Distributed Dynamic Verification with Multiparty Session Types and Python. *Formal Methods in System Design* (2015). https://doi.org/10.1007/s10703-014-0218-8

[14] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. 2009. On Progress for Structured Communications. In *Proc. of TGC (LNCS, Vol. 4912)*. Springer, 257–275.

[15] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *Proc. of ECOOP (LNCS, Vol. 4067)*. Springer, 328–352.

[16] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. https://doi.org/10.1145/2535838.2535856

[17] Andrzej Filinski. 1994. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. ACM Press. https://doi.org/10.1145/174675.178047

[18] Simon Fowler. 2016. An Erlang Implementation of Multiparty Session Actors. In *ICE*. https://doi.org/10.4204/EPTCS.223.3

[19] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. *CoRR* abs/2105.08996 (2021). arXiv:2105.08996 https://arxiv.org/abs/2105.08996

[20] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. of POPL* 3, Article 28 (2019), 29 pages. https://doi.org/10.1145/3290341

[21] Marco Gaboardi, Shin ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM. https://doi.org/10.1145/2951913.2951939

[22] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. *Electronic Proceedings in Theoretical Computer Science* 314 (April 2020), 23–33. https://doi.org/10.4204/eptcs.314.3

[23] Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *Journal of Functional Programming* 20, 1 (2010), 19–50.

[24] Simon J. Gay and Vasco T. Vasconcelos. 2012. Linear type theory for asynchronous session types. *JFP* 20, 1 (2012), 19–50. Extended version of [23].

[25] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

[26] Kohei Honda. 1993. Types for Dyadic Interaction. In *Proc. of CONCUR (LNCS, Vol. 715)*. Springer, 509–523.

[27] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of ESOP (LNCS, Vol. 1381)*. Springer, 122–138.

[28] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proc. of POPL*, Vol. 43(1). ACM, 273–284.

[29] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *Proc. of FASE*. https://doi.org/10.1007/978-3-662-49665-7_24

[30] Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session Type Inference in Haskell. In *Proc. pf PLACES (EPTCS, Vol. 69)*. 74–91. https://doi.org/10.4204/EPTCS.69.6

[31] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In *Proc. of WGP@ICFP*. https://doi.org/10.1145/2808098.2808100

[32] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *Proc. of CONCUR (LNCS, Vol. 4137)*. Springer, 233–247.

[33] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947. https://doi.org/10.1145/330249.330251

[34] Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *EPTCS* 304 (Sept. 2019), 48–60. https://doi.org/10.4204/eptcs.304.4 Renamed to Sesh.

[35] Wen Kokke and Ornela Dardha. 2021. Prioritise the Best Variation. In *Proc. of FORTE (Lect. Not. in Comput. Sci., Vol. 12719)*. Springer, 100–119. https://doi.org/10.1007/978-3-030-78089-0_6

[36] Wen Kokke and Ornela Dardha. 2021. Prioritise the Best Variation. *CoRR* abs/2103.14466 (2021). arXiv:2103.14466 https://arxiv.org/abs/2103.14466 Extended version of [35].

[37] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking protocols with Mungo and StMungo. In *PPDP*. 146–159.

[38] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proc. of PLDI* (Orlando, Florida, USA). ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

[39] Sam Lindley and Conor McBride. 2013. Hasochism. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell - Haskell '13*. ACM Press. https://doi.org/10.1145/2503778.2503786

[40] Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *Proc. of ESOP*. 560–584.

[41] Sam Lindley and J. Garrett Morris. 2016. Embedding session types in Haskell. In *Proc. of Haskell*. ACM, 133–145. https://doi.org/10.1145/2976002.2976018

[42] Sam Lindley and J Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*. River Publishers, 265–286.

[43] Hugo A. Lopez, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, Casar Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-Based Verification of Message-Passing Parallel Programs. In *OOPSLA*. https://doi.org/10.1145/2814270.2814302

[44] Matthias Neubauer and Peter Thiemann. 2004. An Implementation of Session Types. In *Proc. of PADL (Lecture Notes in Computer Science, Vol. 3057)*. Springer, 56–70. https://doi.org/10.1007/978-3-540-24836-1_5

[45] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed Runtime Monitoring for Multiparty Conversations. *Formal Aspects of Computing* (2017). https://doi.org/10.1007/s00165-017-0420-8

[46] Rumyana Neykova and Nobuko Yoshida. 2017. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*. https://doi.org/10.1145/3033019.3033031

[47] Rumyana Neykova and Nobuko Yoshida. 2017. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (March 2017). https://doi.org/10.23638/LMCS-13(1:17)2017

[48] Dominic Orchard, Philip Wadler, and Harley Eades. 2020. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science* 317 (May 2020), 18–38. https://doi.org/10.4204/eptcs.317.2

[49] Dominic Orchard and Nobuko Yoshida. 2017. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools* (2017), 219.

[50] Dominic A. Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *Proc. of POPL*. ACM, 568–581. https://doi.org/10.1145/2837614.2837634

[51] Luca Padovani. 2014. Deadlock and Lock Freedom in the Linear π-Calculus. In *Proc. of CSL-LICS*. ACM, 72:1–72:10.

[52] Luca Padovani. 2017. A simple library implementation of binary sessions. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796816000289 Website: http://www.di.unito.it/~padovani/Software/FuSe/FuSe.html.

[53] Luca Padovani and Luca Novara. 2015. Types for Deadlock-Free Higher-Order Programs. In *Proc. of FORTE (LNCS, Vol. 9039)*. Springer, 3–18.

[54] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. 1996. Concurrent Haskell. In *Proc. of POPL*. ACM, 295–308. https://doi.org/10.1145/237721.237794

[55] Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Proc/ of the Symposium on Haskell*. ACM. https://doi.org/10.1145/2804302.2804309

[56] Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Proc. of Haskell*. ACM. https://doi.org/10.1145/1411286.1411290

[57] Matthew Sackman and Susan Eisenbach. 2008. Session Types in Haskell Updating Message Passing for the 21st Century. (01 2008).

[58] Davide Sangiorgi and David Walker. 2001. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

[59] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *Proc. of ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[60] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

[61] K. C. Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek, and Patrick Eugster. 2010. Efficient Session Type Guided Distributed Interaction. In *Proc. of COORDINATION*, Vol. 6116. https://doi.org/10.1007/978-3-642-13414-2_11

[62] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-Based Language and its Typing System. In *Proc. of PARLE (LNCS, Vol. 817)*. Springer, 398–413.

[63] Philip Wadler. 2012. Propositions as sessions. In *Proc. of ICFP*. 273–286.

[64] Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24, 2-3 (Jan. 2014), 384–418. Extended version of [63].

[65] Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.

6.3 Conclusion

In this chapter, we introduced a library in Linear Haskell which implements session-typed channels based on GV and Priority GV. We defined the monadic reflection of PGV's type system into a graded linear monad, to permit an easy embedding of PGV's priority typing into Linear Haskell's type system. Finally, we compared our Haskell library to existing Haskell libraries for session-typed channels.

In the future, it would be interesting to implement a variant of the library that uses type-level programming for priority inference, as the current design requires the user to manually annotate actions with their priorities. Furthermore, it would be interesting to describe the relation between the library and GV more formally, and formalise the correspondence using a proof-assistant. Finally, it would be useful to extend the library with the link primitive, both from a practical standpoint, and to more closely align the implementation with the formal calculus.

Appendix A

Glossary

A.1 Graph Theory

In this section, I introduce various graph theoretic notions that are used throughout the thesis. I need undirected labelled graphs when discussing connection graphs in § 2.2.5 and § 3.2.5, mixed graphs when discussing dependency graphs and priority graphs in § 2.2.4, § 3.2.4, and § 5.3.2, and undirected multigraphs when discussing abstract process structure in I.3.1.

Commonly, graphs are introduced as tuples of their components. However, since I require undirected, directed, and mixed graphs, I intend to save some ink by defining graphs in terms of their projections.

In the following, if some projection is undefined, it is the empty set, e.g. for an undirected graph, the set of directed arcs is empty, and, consequently, directed reachability is the empty relation.

- A *graph* (ranged over by G) has a set of vertices, denoted by V_G (ranged over by u, v).
- In an abuse of notation, I write u for the singleton graph consisting of the single vertex u, i.e. $V_u \triangleq \{u\}$.
- Two graphs G_1 and G_2 are *disjoint* if and only if their sets of vertices are disjoint, i.e. $V_{G_1} \cap V_{G_2} = \varnothing$.
- An *edge* (ranged over by e) is an unordered pair of vertices, denoted by juxtaposition, i.e. $uv \triangleq \{u, v\}$.
- An *edge-loop* or *loop* is an edge uu that connects a vertex to itself.
- In an abuse of notation, I write uv for the graph consisting of the single edge uv, i.e. $V_{uv} \triangleq \{u, v\}$ and $E_{uv} \triangleq \{uv\}$.
- An *undirected graph* G is a graph with a set of undirected edges with no edge-loops, denoted by E_G, i.e. $E_G \subseteq \{uv \mid u, v \in V_G \wedge u \neq v\}$.

- An *arc* (ranged over by a) is an ordered pair of vertices, denoted by juxtaposition overset with an arrow to indicate the direction, i.e. $\overrightarrow{uv} \triangleq (u, v)$.
- An *arc-loop* or *loop* is an arc $\overrightarrow{uu}$ that connects a vertex to itself.
- In an abuse of notation, I write $\overrightarrow{uv}$ for the graph consisting of the single arc $\overrightarrow{uv}$, i.e. $V_{\overrightarrow{uv}} \triangleq \{u, v\}$ and $A_{\overrightarrow{uv}} \triangleq \{\overrightarrow{uv}\}$.
- A *directed graph* G is a graph with a set of directed arcs with no arc-loops, denoted by A_G, i.e. $A_G \subseteq \{\overrightarrow{uv} \mid u, v \in V_G \wedge u \neq v\}$.
- A *mixed graph* G is a graph with a set of undirected edges with no edge-loops, as above, and a set of directed arcs with no arc-loops, as above.
- An *edge-labelled graph* G is a graph with a set of edges, as above, a set of edge labels, denoted by $\mathscr{L}_G$, and an edge-labelling function, denoted by ℓ_G, where $\ell_G : E_G \to \mathscr{L}_G$. The definitions for E_G and $\mathscr{L}_G$ may be omitted, since $E_G \triangleq \mathrm{dom}(\ell_G)$ and $\mathscr{L}_G \triangleq \mathrm{cod}(\ell_G)$.
- An *undirected multigraph* G is a graph with a set of edge names, denoted by $\mathscr{E}_G$, and an edge-connection function, denoted by r_G, where $r_G : \mathscr{E}_G \to \{uv \mid u, v \in V_G\}$. The set of edges E_G is defined as the union of all edges, i.e. $E_G \triangleq \bigcup_{e \in \mathscr{E}_G} r_G(e)$. An undirected multigraph is similar to an undirected edge-labelled graph but differs in that it permits edge-loops and permits multiple edges between any two vertices.
- The *empty graph* with vertices V, written $\bar{K}_V$, is the graph consisting of vertices V with no edges or arcs, i.e, $V_{\bar{K}_V} \triangleq V$, $E_{\bar{K}_V} \triangleq \varnothing$, and $A_{\bar{K}_V} \triangleq \varnothing$.
- The *graph union* of G_1 and G_2, denoted by $G_1 \cup G_2$, is defined by, for each projection, taking the union of the projection of G_1 and G_2, e.g. $V_{G_1 \cup G_2} \triangleq V_{G_1} \cup V_{G_2}$, and $E_{G_1 \cup G_2} \triangleq E_{G_1} \cup E_{G_2}$, etc..
- The *directed rooting* of G in v, written $v \lessdot G$, is the graph formed by adding the vertex v and adding an arc from v to every other vertex of G, i.e. $V_{v \lessdot G} \triangleq \{v\} \cup V_G$ and $A_{v \lessdot G} \triangleq \{\overrightarrow{vu} \mid u \in V_G\} \cup A_G$. Directed rooting preserves the remaining projections, e.g. $E_{v \lessdot G} \triangleq E_G$.
- For any graph G with vertices $u, v \in V_G$, u is *adjacent* to v if and only if there exists some edge $uv \in E_G$ or some arc $\overrightarrow{uv} \in A_G$.
- For any graph G, a *walk* (ranged over by w) is a sequence of pairwise adjacent vertices. Equivalently, a walk is a sequence of edges and arcs that join a sequence of vertices, where all arcs have the same direction.
- A walk w *visits* a vertex v if and only if v occurs in the walk, i.e. if and only if $w_i = v$ for some $i \in \mathbb{N}$.
- A walk is *closed* if and only if its first and last vertex are the same.

- A walk is *undirected* if and only if it contains only edges.
- A walk is *directed* if and only if it contains only arcs.
- A walk is *essentially directed* if and only if it contains at least one arc.
- A *path* (ranged over by p) is a walk without repeated vertices.
- A *cycle* (ranged over by c) is a closed path.
- A graph is *acyclic* if and only if the undirected graph formed by replacing all arcs in with edges contains no cycles.
- A graph is *essentially acyclic* if and only if it contains no essentially directed cycles.
- A graph is *strongly connected* if and only if there exists a path between any two distinct vertices.
- A graph is *connected* if and only if the undirected graph formed by replacing all arcs in with edges is strongly connected. (Commonly, connected is also referred to as *weakly connected*.)
- For any graph G, the *subgraph* of G induced by U, denoted by $G[U]$, where $U \subseteq V_G$, is the graph constructed by taking the subset of vertices U and restricting G's projections to vertices in U.
- For any graph G, a *strongly connected component* of G (ranged over by C) is a maximal strongly connected subgraph of G.
- For any graph G, a *component* of G (ranged over by C) is a maximal connected subgraph of G.
- A *tree* (ranged over by T) is a graph that is connected and acyclic.
- A *forest* (ranged over by F) is a graph whose components are trees.
- For any graph G, *undirected reachability*, denoted by $\sim_G$, is the equivalence closure over E_G, i.e. $u \sim_G v$ holds if $u = v$ or there exists an undirected path from u to v. (If E_G is undefined, undirected reachability is the smallest reflexive relation over V_G.)
- For any graph G, *essentially directed reachability*, denoted by $\prec_G$, is the transitive closure over A_G quotiented by $\sim_G$, i.e. $u \prec_G v$ holds if and only if there exists an essentially directed path from u to v. (If A_G is undefined, essentially directed reachability is the empty relation.)
- For any graph G, *reachability*, denoted by $\preccurlyeq_G$, is the union of undirected and essentially directed reachability, i.e. $u \preccurlyeq_G v$ holds if and only if there exists an undirected or essentially directed path from u to v.
- For any graph G, its *sources* and *sinks*, written sources(G) and sinks(G), respectively, are the sets of minimals and maximals of

essentially directed reachability, respectively, i.e. sources(G) $\triangleq$ $\{u \in V_G | \nexists v.v <_G u\}$ and sinks(G) $\triangleq$ $\{u \in V_G | \nexists v.u <_G v\}$.

Lemma A.1. *If T_1 and T_2 are disjoint trees and u and v are vertices in T_1 and T_2, respectively, then the graph G formed by connecting T_1 and T_2 with the edge uv is a tree.*

Lemma A.2. *If G_1 and G_2 are disjoint essentially acyclic mixed graphs, $(u_1, \ldots, u_n)$ and $(v_1, \ldots, v_n)$ are vertices in G_1 and G_2, respectively, such that $u_1 < \ldots < u_n$ and $v_1 < \ldots < v_n$, respectively, then the mixed graph G formed by connecting G_1 and G_2 with the edges $u_1v_1, \ldots, u_nv_n$ is essentially acyclic.*

Proof. By contradiction. Assume c is an essentially directed cycle in G.

- If c visits no vertex in G_2, then c is an essentially directed cycle in G_1.
- If c visits no vertex in G_1, then c is an essentially directed cycle in G_2.
- Otherwise, c must visit some vertex u in G_1 and some vertex v in G_2.

  As c is an essentially directed cycle, there must be distinct paths $p_{uv} = (u, \ldots, v)$ and $p_{vu} = (v, \ldots, u)$. At least one of p_{uv} and p_{vu} must be essentially directed.

  As G_1 and G_2 are disjoint, p_{uv} and p_{vu} must each contain at least one of the edges $u_1v_1, \ldots, u_nv_n$. Hence, there must be paths $p_{uu_i} = (u, \ldots, u_i)$, $p_{v_iv} = (v_i, \ldots, v)$, $p_{vv_j} = (v, \ldots, v_j)$, and $p_{u_ju} = (u_j, \ldots, u)$ for some $1 \leq i, j \leq n$. At least one of p_{uu_i}, p_{u_ju}, p_{vv_i}, and p_{v_jv} must be essentially directed.

  - If $i = j$, then either c contains an arc in G_1 and $p_{uu_i}p_{u_ju}$ is an essentially directed cycle in G_1, or c contains an arc in G_2 and $p_{v_iv}p_{vv_j}$ is an essentially directed cycle in G_2.
  - If $i < j$, then $u_i < u_j$ and $v_i < v_j$. There must be essentially directed paths $p_{u_iu_j} = (u_i, \ldots, u_j)$ in G_1 and $p_{v_iv_j} = (v_i, \ldots, v_j)$ in G_2, and $p_{uu_i}p_{u_iu_j}p_{u_ju}$ is an essentially directed cycle in G_1.
  - If $i > j$, then $u_j < u_i$ and $v_j < v_i$. There must be essentially directed paths $p_{u_ju_i} = (u_j, \ldots, u_i)$ in G_1 and $p_{v_jv_i} = (v_j, \ldots, v_i)$ in G_2, and $p_{v_iv}p_{vv_j}p_{v_jv_i}$ is an essentially directed cycle in G_2.

□

A.2 Multisets

Definition A.3 (Multiset)**.** *A* multiset *is a variant of a set that allows multiple occurrences of each element. Formally, a multiset $\mathscr{X}$ is a tuple (X, μ_X) where X is the support set of the multiset, and μ_X is a function, giving each element its* multiplicity*, from X to the class of nonzero cardinal*

numbers. I write multisets as lists of elements between "⟅" and "⟆", e.g. $⟅a, b, b, b⟆$.

I define the following operations on multisets:

Membership	$a \in^k \mathscr{X}$	$\triangleq$	$a \in X \wedge \mu_X(a) = k$
Union	$\mathscr{X} \cup \mathscr{Y}$	$\triangleq$	$(X \cup Y, \max(\mu_X, \mu_Y))$
Intersection	$\mathscr{X} \cap \mathscr{Y}$	$\triangleq$	$(X \cap Y, \min(\mu_X, \mu_Y))$
Deletion	$\mathscr{X} \setminus Y$	$\triangleq$	$(X \setminus Y, \mu_X \lhd Y)$
Subtraction	$\mathscr{X} - \mathscr{Y}$	$\triangleq$	$(\{a \in X \cup Y \mid \mu_X(a) > \mu_Y(a)\}, \mu_X - \mu_Y)$
Sum	$\mathscr{X} + \mathscr{Y}$	$\triangleq$	$(\mathscr{X} \cup \mathscr{Y}, \mu_X + \mu_Y)$
Product	$\mathscr{X}\mathscr{Y}$	$\triangleq$	$(\mathscr{X} \cap \mathscr{Y}, \mu_X \mu_Y)$
Symmetric Difference	$\mathscr{X} \triangle \mathscr{Y}$	$\triangleq$	$(\mathscr{X} - \mathscr{Y}) \cup (\mathscr{Y} - \mathscr{X})$

Definition A.4 (Domain Subtraction)**.** Domain subtraction*,* $f \lhd X$*, is the operation that removes all elements in* X *from the domain of the function* f*, i.e.* $f \lhd X = a \mapsto f(a)$ *if* $a \notin X$*.*

Bibliography

Samson Abramsky. Proofs as processes. *Theoretical Computer Science*, 135 (1):5–9, April 1994. ISSN 0304-3975. doi: 10.1016/0304-3975(94)00103-0.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 6 1992. ISSN 0955-792X. doi: 10.1093/logcom/2.3.297.

Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3 & 4):335–376, 2009. doi: 10.1017/S095679680900728X.

Robert Atkey. Observed communication semantics for classical processes. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 56–82, 2017. doi: 10.1007/978-3-662-54434-1_3.

Robert Atkey, Sam Lindley, and J. Garrett Morris. *Conflation Confers Concurrency*, pages 32–55. Springer International Publishing, Cham, 2016. doi: 10.1007/978-3-319-30936-1_2.

Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991. doi: 10.1007/BF01531058. Arnon Avron credits Garrel Pottinger [Pottinger, 1983] with the first use of hypersequents (Footnote 3, p. 227).

Arnon Avron. The Method of Hypersequents in the Proof Theory of Propositional Non-classical Logics. In *Logic: from Foundations to Applications: European logic colloquium*. Oxford University Press, June 1996. ISBN 9780198538622. doi: 10.1093/oso/9780198538622.003.0001.

Hendrik Pieter Barendregt. The lambda calculus - its syntax and semantics. In *Studies in Logic and the Foundations of Mathematics*, 1985. URL https://api.semanticscholar.org/CorpusID:263892017.

Gianluigi Bellin and Philip J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, April 1994. ISSN 0304-3975. doi: 10.1016/0304-3975(94)00104-9.

Nuel Belnap. Linear logic displayed. *Notre Dame Journal of Formal Logic*, 31(1), December 1989. ISSN 0029-4527. doi: 10.1305/ndjfl/1093635329.

Nuel D. Belnap. Display logic. *Journal of Philosophical Logic*, 11(4):375–417, 1982. ISSN 00223611, 15730433. URL http://www.jstor.org/stable/30226258.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. of POPL*, 2: 1–29, 2018. doi: 10.1145/3158093.

Michele Boreale. *On the expressiveness of internal mobility in name-passing calculi*, page 163–178. Springer Berlin Heidelberg, 1996. ISBN 9783540706250. doi: 10.1007/3-540-61604-7_54. URL http://dx.doi.org/10.1007/3-540-61604-7_54.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010 - Concurrency Theory*, pages 222–236. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-15375-4_16.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. doi: 10.1017/S0960129514000218. Journal version of Caires and Pfenning [2010].

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33:1–33:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-017-0. doi: 10.4230/LIPIcs.CONCUR.2016.33. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2016.33.

B. Jack Copeland, editor. *The Essential Turing*. Clarendon Press, Oxford, England, July 2004. Via Wikipedia [2024] under History.

Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2–3):316–383, 2014. doi: 10.1017/S0956796814000069.

Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, October 1989. ISSN 1432-0665. doi: 10.1007/bf01622878.

Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, volume 10803 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2018. doi: 10.1007/978-3-319-89366-2_5.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi: 10.1016/J.IC.2017.06.002. URL https://doi.org/10.1016/j.ic.2017.06.002.

Martin Davis. Lecture at the Control Systems Laboratory at the University of Illinois, 1952. Via Copeland [2004], Footnote 61, p. 40.

Martin Davis. *Computability & Unsolvability*. McGraw-Hill, New York, USA, 1958.

Thomas Ehrhard. Finiteness spaces. *Mathematical Structures in Computer Science*, 15(4):615–646, 2005. doi: 10.1017/S0960129504004645.

Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28(7):995–1060, 2018. doi: 10.1017/S0960129516000372.

Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Theoretical Computer Science*, 364(2):166–195, November 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.08.003.

Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*. ACM Press, 1994. doi: 10.1145/174675.178047. URL https://doi.org/10.1145/174675.178047.

Simon Fowler. *Typed concurrent functional programming with channels, actors and sessions*. PhD thesis, The University of Edinburgh, 2019.

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi: 10.1145/3290341.

Simon Fowler, Wen Kokke, Ornela Dardha, J. Garrett Morris, and Sam Lindley. Separating Sessions Smoothly. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory (CONCUR 2021)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-203-7. doi: 10.4230/LIPIcs.CONCUR.2021.36.

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating Sessions Smoothly. *Logical Methods in Computer Science*, 19(3), July 2023. doi: 10.46298/lmcs-19(3:3)2023. Journal version of Fowler et al. [2021].

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

J. Y. Girard and Y. Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87*, pages 52–66, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. ISBN 978-3-540-47717-4.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989. ISBN 0521371813.

Alessio Guglielmi. A system of interaction and structure. *ACM Trans. Comput. Logic*, 8(1):1–es, jan 2007. ISSN 1529-3785. doi: 10.1145/1182613.1182614.

Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47968-0.

Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theoretical Computer Science*, 411(22):2223–2238, 2010. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2010.01.028.

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

Stephen Cole Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, Inc., January 1952. Via Wikipedia Wikipedia [2024] under History.

Naoki Kobayashi. *A New Type System for Deadlock-Free Processes*, page 233–247. Springer Berlin Heidelberg, 2006. ISBN 9783540373773. doi: 10.1007/11817949_16. URL http://dx.doi.org/10.1007/11817949_16.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 358–371, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.237804.

Wen Kokke. Races in Classical Linear Logic. Master's thesis, University of Edinburgh, August 2017.

Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in

Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, September 2019. ISSN 2075-2180. doi: 10.4204/eptcs.304.4. Renamed to Sesh.

Wen Kokke and Ornela Dardha. Prioritise the best variation. In Kirstin Peters and Tim A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 100–119, Cham, February 2021a. Springer International Publishing. ISBN 978-3-030-78089-0.

Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, page 1–13, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450386159. doi: 10.1145/3471874.3472979.

Wen Kokke and Ornela Dardha. Prioritise the Best Variation. *Logical Methods in Computer Science*, Volume 19, Issue 4, December 2023. doi: 10.46298/lmcs-19(4:28)2023. Journal version of Kokke and Dardha [2021a].

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better Late Than Never: A fully-abstract semantics for Classical Processes. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019a. doi: 10.1145/3290337.

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking Linear Logic Apart. In Thomas Ehrhard, Maribel Fernández, Valeria de Paiva, and Lorenzo Tortora de Falco, editors, *Proceedings Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Oxford, UK, 7-8 July 2018*, volume 292 of *Electronic Proceedings in Theoretical Computer Science*, pages 90–103. Open Publishing Association, April 2019b. doi: 10.4204/EPTCS.292.5. Version with errata: https://marcoperessotti.com/files/papers/KMP18-revised.pdf.

Sam Lindley and J. Garrett Morris. Sessions as propositions. In Alastair F. Donaldson and Vasco T. Vasconcelos, editors, *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014*, volume 155 of *EPTCS*, pages 9–16, 2014. doi: 10.4204/EPTCS.155.2. URL https://doi.org/10.4204/EPTCS.155.2.

Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 560–584, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46669-8. doi: 10.1007/978-3-662-46669-8_23.

Sam Lindley and J. Garrett Morris. Embedding session types in haskell. In *Proceedings of the 9th International Symposium on Haskell*,

Haskell 2016, page 133–145, New York, NY, USA, 2016a. Association for Computing Machinery. ISBN 9781450344340. doi: 10.1145/2976002.2976018.

Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2016b. doi: 10.1145/2951913.2951921.

Sam Lindley and J. Garrett Morris. *Lightweight Functional Session Types*, pages 265–286. River Publisher, 2017. doi: 10.13052/rp-9788793519817.

Simon Marlow et al. Haskell 2010 language report. Available online https://www.haskell.org/onlinereport/haskell2010/, 2010. Available online https://www.haskell.org/onlinereport/haskell2010/.

Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34 (3):103–104, oct 2014. ISSN 1094-3641. doi: 10.1145/2692956.2663188. URL https://doi.org/10.1145/2692956.2663188.

Damiano Mazza. The true concurrency of differential interaction nets. *Mathematical Structures in Computer Science*, 28(7):1097–1125, 2018. doi: 10.1017/S0960129516000402.

Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 47, 2010.

Paul-André Melliès. *A Topological Correctness Criterion for Multiplicative Non-Commutative Logic*, page 283–322. Cambridge University Press, Cambridge, 2004.

Robin Milner. The polyadic pi-calculus: A tutorial. Technical report, The University of Edinburgh, 1991. URL http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992a. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90008-4.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992b. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90009-5.

Fabrizio Montesi and Marco Peressotti. Classical transitions. Technical report, University of Southern Denmark, Odense, Denmark, 2018.

Fabrizio Montesi and Marco Peressotti. Linear logic, the π-calculus, and their metatheory: A recipe for proofs as processes. *CoRR*, abs/2106.11818, 2021. URL https://arxiv.org/abs/2106.11818.

Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14. ACM, July 2014. doi: 10.1145/2603088.2603116. URL http://dx.doi.org/10.1145/2603088.2603116.

Garrel Pottinger. Uniform, cut-free formulations of t, s4 and s5. *Journal of Symbolic Logic*, 48(3):900, 1983. (In the notes for the Annual Meeting of the Association for Symbolic Logic, Philadelphia, 1981.).

Zesen Qian. *Concurrency And Races In Classical Linear Logic*. PhD thesis, Århus Universitet, January 2023.

Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proceedings of the ACM on Programming Languages*, 5 (ICFP), August 2021. doi: 10.1145/3473567. Expanded on in Qian's Ph.D. thesis [Qian, 2023].

Christian Retoré. Pomset logic: A non-commutative extension of classical linear logic. In *Lecture Notes in Computer Science*, pages 300–318. Springer Berlin Heidelberg, 1997. doi: 10.1007/3-540-62688-3_43.

Davide Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. *Theoretical Computer Science*, 167(1):235–274, 1996. ISSN 0304-3975. doi: 10.1016/0304-3975(96)00075-8.

Davide Sangiorgi and David Walker. *The pi-calculus*. Cambridge University Press, Cambridge, England, October 2003.

Sergey Slavnov. On noncommutative extensions of linear logic. *Logical Methods in Computer Science*, Volume 15, Issue 3, September 2019. doi: 10.23638/LMCS-15(3:30)2019.

Lutz Straßburger. A local system for linear logic. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 388–402, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-36078-0.

Aaron Stump. The calculus of dependent lambda eliminations. *Journal of Functional Programming*, 27:e14, 2017. doi: 10.1017/S0956796817000053.

Alwen Tiu. A System of Interaction and Structure II: The Need for Deep Inference. *Logical Methods in Computer Science*, Volume 2, Issue 2, April 2006. doi: 10.2168/LMCS-2(2:4)2006. URL https://lmcs.episciences.org/2252.

Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 273–286, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310543. doi: 10.1145/2364527.2364568.

Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014. doi: 10.1017/S095679681400001X. Journal version of Wadler [2012].

Wikipedia. Halting problem — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Halting%20problem&oldid=1184812236. [Online; accessed 12-January-2024].

Nobuko Yoshida, Kohei Honda, and Martin Berger. Linearity and bisimulation. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 417–433, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45931-6.