Adjoint Logic with Applications

Klaas Pruiksma

CMU-CS-24-103

May 7, 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Chair
Stephanie Balzer
Henry DeYoung
Robert Harper
Andrew Pitts

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was partially funded by the National Science Foundation under grant number CCF1718267.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Proof Theory, Programming Languages, Adjoint Logic, Program Equivalence

To my partner Al and cat Ackermann.
Most of all, to those who will read this in the future — may you find the information within helpful.

Abstract

Many different systems of logic, such as linear logic, lax logic, and various modal logics have been studied extensively and find applications in diverse domains. Likewise, in the context of programming languages, many different language features have been explored fruitfully. In both cases, however, these different features are often studied in isolation, or in the context of some simple base language. As such, it can be unclear how these features interact, if we want to work with a programming language combining different features, or a logic that allows us to model behavior from several different base logics.

Adjoint logic is a framework or schema for defining logics based on a set of *modes*, each of which represents a single base logic. These base logics are then combined uniformly and coherently into a single instance of adjoint logic. In this document, we will develop a form of adjoint logic that forms a suitable basis for concurrent programming languages. We first develop the proof theory of adjoint logic, proving generically useful logical results such as cut elimination, identity expansion, and focusing. This ensures that our approach to adjoint logic yields a sensible proof system, and means that we can get these results for free for a given logic by showing that it is an instance of adjoint logic.

Interpreting proof reduction as communication between concurrent processes, particularly using a *semi-axiomatic sequent calculus* formulation to model asynchronous communication, we can convert the framework of adjoint logic into a similar adjoint framework for programming languages. By making use of different modes, we can model a range of communication behavior, notably including multicast, where a message is sent to multiple recipients. We also see that with a different interpretation, we can model communication via shared memory, which then also lends itself well to reconstructing sequential computation within this concurrent language. Additionally, the uniformity of this framework means that as we add features to (or encode features into) these languages, we will naturally also be able to work with multi-featural languages, perhaps restricted by mode.

Further building on these languages, we then explore notions of program equivalence in the adjoint setting, which may be useful both for the development of dependent adjoint types, as well as for reasoning about programs, particularly in the context of optimization. A first handling of a uniformly defined equivalence across all modes provides a blueprint for how to work with equivalence in the adjoint setting, particularly handling situations where some, but not all data may be reused, and addressing the intuitive idea of equivalence when communication is limited to a specific interface, which may consist of multiple channels/memory addresses, each with their own specification for communication behavior. Using this, we can then examine several examples of *mode-dependent equivalence*, where we combine multiple different notions of equivalence coherently into one.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Frank Pfenning, who has been an influence on my research since my very first semester of undergrad, when I was directed to him with naive questions about garbage collection. The work presented in this document would not have been possible without all of his guidance, feedback, and support over the past seven years or more. I also want to thank my committee, Stephanie Balzer, Henry DeYoung, Bob Harper, and Andy Pitts, for their feedback and willingness to take the time to serve on a committee, with all that entails. I would particularly like to thank Stephanie for all of her guidance in how to navigate an academic conference — without that advice, I would likely have never met Prof. Pitts, and would have had a very different committee.

Beyond my committee, there are several other people with whom I have had fruitful discussions about this work, or gotten feedback on some portion of it from. Three of these people have had an especially large impact — Wiliam Chargin, with whom I collaborated on some of the earliest parts of this work, developing several earlier drafts of the proof theory of adjoint logic; and Ryan Kavanagh and Siva Somayyajula, both of whom, I believe, have read and commented on more versions of my papers, drafts, and working notes than anyone else. Though less extensive, I appreciate no less the discussions I have had about my work with others, including Joelle Dionne, Hannah Gommerstadt, Danny Gratzer, Jon Sterling, Yong Kiam Tan, and Jennifer Wang.

A few sources of reference material have been especially helpful. I greatly appreciate the aid of the Bibliothèque Nationale de France in making it possible to find old papers that seemingly appear nowhere else. Bob Harper's excellent book on programming language theory was also incredibly useful throughout this work, serving as an excellent reference on many different aspects of the field.

Several of my undergraduate professors have also had a large impact on this work, if more indirectly. Po-Shen Loh and John Mackey were both extremely influential in reinforcing my decision to pursue research in mathematics to begin with. Rami Grossberg helped a great deal in building my interest in and understanding of logic, and also was heavily influential in my decision to study mathematics in a computer science department, leading me to the line of research I have followed. I credit Dave Eckhardt's mentorship for (among other things) a substantial amount of my understanding of and opinions on concurrent computation.

Beyond the academic side of things, my friends and family members have provided a great deal of support, distractions when things are difficult, and just generally company throughout the process. I want to particularly thank my partner, Al, for putting up with all of the time that research, papers, and a thesis take, and all the associated long nights of writing, and my parents, for encouraging me both to get to the point of starting my graduate studies, and throughout those studies. I also appreciate all of the non-working time spent with my fellow students, including (but surely not limited to) Sol Boucher, Kristy Gardner, Isaac Grosof, and Ziv Scully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Linear Logic [35], in both its intuitionistic and classical forms, has seen use in various parts of the computer science literature. Other substructural logics, [1] particularly affine logic, likewise have broad applications. In programming language design, Rust [2] makes use of affine types in its borrow checker for memory, to ensure that at most one piece of code at a time has permission to modify a given piece of memory. Linear types also appear in models of quantum computation such as the quantum lambda calculus [97, 106], where they prevent quantum bits from being reused in a manner inconsistent with physical reality. For reasoning about program behavior, there is the example of *separation logic* [11, 68, 69, 90], which is used for reasoning about programs with a pointer-based memory heap, and is partially substructural. [3] Reasoning about logic and deductive systems likewise can be done in a substructural setting, as with the linear logical framework LLF [15], or the related concurrent logical framework CLF [17, 96, 108].

In order to retain the expressive power of standard structural logic, and to work with both linear and nonlinear (or non-affine, etc) portions of a system, linear logic as initially presented by Girard [35] makes use of the exponential modality ! to mark propositions as non-linear, and therefore freely reusable. This approach to mixing linear and non-linear propositions has two main disadvantages. Firstly, in such systems, linearity (or affinity, etc) is the default, and so working nonlinearly can require a fair amount of indirection and working through encodings. Second, and relatedly, while this approach allows for use of both linear and structural/non-linear components of a system fairly simply, if we want to mix more systems, for instance combining linear, affine, and structural components, a more complex system is needed, with more complex encoding work.

Several different systems for combining substructural (and structural) logics have also been developed, addressing these disadvantages in varied ways. A first of these is Benton's mixed linear and non-linear logic LNL [7], which combines linear and structural logic in a more uniform way, not biasing towards linearity as Girard's linear logic does. There are also several systems of subexponential logic [18, 24, 49, 66, 67], which provide more complex ways to combine sub-

[1] The details of linear and substructural logics are not necessary to understand this section, but the curious reader may jump ahead to Section 1.2 for a brief overview of these systems

[2] As formalized after the fact in several different ways, see, e.g., [48, 109]

[3] The separating conjunction $*$ is a form of linear conjunction, requiring that its conjuncts hold in disjoint parts of the heap

structural logics, using differently tagged exponentials $!_c$, which behave largely like the standard exponential !, but are not inherently related to one another. Adjoint logic, which has been explored in some previous forms [58, 59, 88], builds on both of these lines of work, attempting to provide both a uniform, unbiased approach to combining logics (as in LNL), while also allowing for more general and complex combinations of logics as in systems of subexponential linear logic. For instance, adjoint logic can be used to model a variety of modal logics that do not inherently relate to its substructural origins, such as Lax Logic [31], or the $\diamond$-free fragment of modal S4 [75].

Beyond pure proof theory, adjoint logic also has several applications, both in the more theoretical direction of logical frameworks, and the more applied direction of programming language semantics. Our contributions in the direction of logical frameworks are modest — a first step towards providing a logical framework is to simplify proof search via a focused calculus, which we explore in a few steps. Caires and Pfenning observed that the purely linear (i.e. without the exponential !) sequent calculus[4] can be used as the basis for a concurrent (session-typed) programming language via a proofs-as-programs interpretation [12]. In the purely linear case, each message has exactly one recipient (and dually, every provider of a service has exactly one client). A process with an open communication channel also cannot exit without first concluding the protocol used on that channel, because otherwise some messages might be sent along that channel with no receiver. With the addition of !, some of this expressive power can be regained, but with limited precision, as any channel with session type $!A$ can be used for any number of repetitions of the protocol specified by A. However, while this allows for replicable services, it still does not provide *multicast* functionality, where one message can be sent to multiple recipients. Recursive types also allow us to achieve similar results to !, with a channel being able to produce multiple copies of an A (either arbitrarily many or some bounded number), but do not allow for channels to be used by multiple processes. By making use of recursion, it is also possible to copy some values (e.g. natural numbers represented in unary, or bit streams) to be used multiple times, but again, this reuse is within a single process. As such this approach also does not provide support for multicast, nor for sharing of a service between multiple processes, as ! does. Similarly, while both approaches allow for services to be discarded as necessary, neither allows for *messages* to be discarded.

Building on this work in a different direction, SILL [77, 104] combines a fully-featured functional language with a concurrent linear language, embedding linear computations into the functional language via a (contextual) monad. This approach allows both the linear and non-linear portions of the language to be handled "naturally", resembling a form of π-calculus and λ-calculus, respectively. However, the concurrent portion of SILL is still limited in similar ways to Caires and Pfenning's $\pi\mathsf{DILL}$, as described above. The addition of the functional layer (which can be called into from the concurrent layer) makes it possible to invoke computations which use data non-linearly from within the concurrent layer, but then these computations, living in the functional layer, are not themselves concurrent. As SILL's approach is non-uniform, with the two layers using different syntax (and different logical underpinnings, as the functional layer is based on natural deduction, while the concurrent layer is based on the sequent calculus), it is also not entirely obvious how to extend this approach to additionally handle, for instance, an affine

[4] Often called *MALL*, or *Multiplicative-Additive Linear Logic*

portion of the language.

Adjoint logic, with its uniform handling of logics with varied structural properties, including linear logic, can also serve as a basis for a more uniform and more expressive framework for programming language that make use of substructural features. By extending the work of Caires and Pfenning [12] to adjoint logic, we can provide a concurrent programming language with more natural, uniform support for concurrency patterns involving reuse, sharing, or non-use of channels, including multicast and termination of channels which may already have messages sent along them.

An adjoint programming language makes it easier to write more complex concurrent programs, but a major difficulty in programming, particularly in concurrent programming, is with reasoning about the behavior of programs. Program behavior is, in general, quite complex, and there are many different aspects to consider, but we will focus in particular on program equivalence. A key application of this is in program transformations, particularly when it comes to optimization. To optimize a program, we generally want to apply some transformation that improves efficiency (by some measure, such as time, space, energy consumption, and so on), while ensuring that the transformation does not change the meaning of the program. A suitable notion of program equivalence then lets us evaluate whether a given transformation affects meaning, either in general, or in some restricted cases (for instance, if we know that some piece of memory is used linearly, after it is read from, it can safely be reallocated, but this optimization is not sound for memory that might be accessed more than once. This also means, interestingly, that some optimizations may be sound (in the sense of preserving program meaning) in some parts (e.g. the linear portion) of an adjoint programming language, but not on the language as a whole. By considering how equivalence interacts between different layers of an adjoint language (or how it does not), we can justify applying this type of *locally sound* optimization, without needing to consider what impact it may have outside of the layer or portion of the language where it is sound, or where its necessary constraints are met.

With this thesis, we seek to develop a more full theory of adjoint logic, build on this to develop adjoint programming languages, and explore equivalence for programs in these languages. In doing so, we will support the following:

> **Thesis Statement:** *Adjoint logic provides a suitably general framework for combining diverse components of deductive systems, not just in pure logic, but also when applied to both the specification of and reasoning about programming languages.*

1.1 Contributions

There are several major components to the contributions of this thesis. Firstly (Chapter 2), we present a sequent calculus for adjoint logic in three different formulations — one with explicit structural rules, one with implicit structural rules, and a focused system. We also develop some of the proof theory of adjoint logic, proving cut elimination for each of these systems, and proving that the three systems are equivalent in terms of what they can prove, so they do indeed describe the same logic. Our work on the proof theory of adjoint logic is not yet published, though it is referenced as an unpublished manuscript ([86]) in our later published work that builds on it. While investigating this logic, we developed a new way to present logics in general, the *Semi-*

Axiomatic Sequent Calculus (Chapter 3), which is not inherently related to adjoint logic, but was useful in much of the following work. The adjoint form of the semi-axiomatic sequent calculus presented in this thesis extends our prior published work ([26]) on semi-axiomatic sequent calculus for standard, structural logic, as well as an earlier semi-axiomatic system of adjoint logic used as the basis for programming languages [83], prior to a full theoretical understanding of the semi-axiomatic sequent calculus in general. Building on a proofs-as-programs interpretation of adjoint logic, we then reinterpret the proof theory of adjoint logic (specifically, a semi-axiomatic presentation with implicit structural rules) as the basis for the semantics of several related programming languages. These languages (Chapter 4) interpret proofs as concurrently running processes, which may communicate either via asynchronous message passing, or via a form of shared memory which serves as an implementation of typed substructural futures [39]. For both languages, we prove type-safety via progress and preservation theorems, as well as a confluence result, since the semantics are naturally nondeterministic. The presentation of adjoint programming languages given here builds on both our prior work on message-passing [83, 84] and that on shared memory [85]. The latter appears almost unchanged here, while the message-passing system we present in this thesis, while related to our earlier system, has been revised to fit better into the same framework as the shared-memory language we work with more extensively. We then explore notions of equivalence for these programs (Chapter 5), presenting both a general notion of observational equivalence, which is parameterized by what is observable and a notion of logical equivalence which agrees with an extensional form of observational equivalence. Finally, we seek to lift this notion of logical equivalence into the adjoint setting, allowing for different modes to have different equalities, which can then be combined into a coherent whole. As with the proof theory of adjoint logic, the development of equivalence in the adjoint setting is not yet published.

1.2 Background on Substructural Logic

We present here a brief overview of intuitionistic purely linear logic (i.e linear logic without !, often called IMALL), as well as other substructural logics. The key distinguishing feature of linear logic is that hypotheses must be used exactly once, as opposed to the more familiar structural logic, where hypotheses may be reused (e.g. $(A \supset (B \supset C)) \supset (A \supset B) \supset A \supset C$[5] is provable, using A twice) or discarded (e.g. $A \supset (B \supset A)$ is provable, discarding B).

This restriction means that the familiar $\wedge$ of structural logic splits into two different connectives in linear logic — $\otimes$, from which both conjuncts can be extracted, and $\&$, from which only a single conjunct can be extracted (but either may be chosen, unlike with $\vee$ or $\oplus$, which only contain a single disjunct). A proof of the multiplicative conjunction $A \otimes B$ consists of separate proofs of A and B, which use disjoint sets of hypotheses, while a proof of $A \& B$ consists of a proof of A and a proof of B, which use the same set of hypotheses. The other connectives of linear logic correspond more clearly to their structural counterparts. The linear implication $\multimap$ is much the same as a structural implication, in that a proof of $A \multimap B$ is given by a proof of B which depends on A (or, equivalently, a transformation from proofs of A to proofs of B). The

[5] The type of the S combinator $Sxyz = xz(yz)$

$$\frac{}{A \vdash A}\ \mathsf{id} \qquad \frac{\Delta_1 \vdash A \quad \Delta_2, A \vdash C}{\Delta_1, \Delta_2 \vdash C}\ \mathsf{cut}$$

$$\frac{i \in J \quad \Delta \vdash A^i}{\Delta \vdash \oplus_{j \in J} A^j}\ \oplus R^i \qquad \frac{\Delta, A^j \vdash C \quad \text{for all } j \in J}{\Delta, \oplus_{j \in J} A^j \vdash C}\ \oplus L$$

$$\frac{\Delta \vdash A^j \quad \text{for all } j \in J}{\Delta \vdash \&_{j \in J} A^j}\ \& R \qquad \frac{i \in J \quad \Delta, A^i \vdash C}{\Delta, \&_{j \in J} A^j \vdash C}\ \& L^i$$

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R \qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ \otimes L$$

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R \qquad \frac{\Delta_1 \vdash A \quad \Delta_2, B \vdash C}{\Delta_1, \Delta_2, A \multimap B \vdash C}\ \multimap L$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}L$$

Figure 1.1: A sequent calculus presentation of intuitionistic purely linear logic (IMALL)

difference here is that the proof of B must use A exactly once, whereas a proof of the structural implication $A \supset B$ may discard or reuse A in proving B. Finally, while there are in principle two distinct linear disjunctions, in the intuitionistic setting, only one of these, $\oplus$, is present. A proof of $A \oplus B$ consists of either a proof of A or a proof of B, much like in the structural setting.

In Figure 1.1, we present a sequent calculus for linear logic. Here, the objects we work with and manipulate are *sequents*, written $\Delta \vdash A$, where Δ, the context, stands for a multiset or unordered list of propositions (so, for instance, a context might be something like the list A, A, B, C, containing two copies of the proposition A and one each of B and C). When we write a comma between two contexts (e.g., Δ_1, Δ_2), we mean the multiset union of these contexts, or, more simply, the list formed by first writing out all of the entries of Δ_1, followed by all of the entries of Δ_2. Note that as multisets are unordered, Δ_1, Δ_2 and Δ_2, Δ_1 both represent the same multiset, something which we will often use without further comment. Because it will be convenient through the rest of this thesis to work with n-ary $\&$ and $\oplus$ rather than just the binary form, we will do so here as well, but of course the two are equivalent (as long as we provide a nullary form along with the binary one). The index set J here is an arbitrary finite set, as we are not interested in infinitary connectives in this work.

The key point that ensures that hypotheses are used exactly once from these rules is that the identity rule only allows us to prove A if the context is *exactly* A — no further hypotheses can exist in the context to be ignored. Secondarily, we ensure that hypotheses are never reused in a proof by consuming them in each of the left rules, and by splitting up the context into disjoint pieces in the cut, $\otimes R$, and $\multimap L$ rules, where we need to prove multiple subgoals, all of which are used in the overall proof ($\& R$ and $\oplus L$ provide all hypotheses in each subgoal rather than splitting them up because only a single subgoal can actually be extracted from the $\&/\oplus$, and so there is no actual reuse of hypotheses).

We can then recover structural logic by adding explicit rules for reuse or non-use of hypotheses, called *contraction* and *weakening*, respectively:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\ \textsf{contract} \qquad\qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \textsf{weaken}$$

The other substructural logics that we will consider then arise by taking one, but not both of these rules — with only weakening, we get *affine logic*, in which hypotheses must be used *at most once* — reuse is still not possible, but not all hypotheses need to be used. Taking instead only contraction, we get *strict logic*, in which every hypothesis must be used *at least once* — hypotheses may be freely reused, but they must be relevant to the goal that is proven, as they cannot just be discarded. Other substructural logics exist, particularly *ordered logics*, which reject a rule called *exchange*, which allows for reordering the context of hypotheses, and which we have left implicit here, as we will always assume it throughout the work that follows. However, we will restrict our focus here to linear, affine, strict, and structural logic.

1.3 Outline of Structure

The structure of this thesis largely follows the breakdown of the contributions of the thesis into distinct themes.

We will begin in Chapter 2 by exploring the prior work that adjoint logic builds on, giving context to define the core concepts of adjoint logic itself. With a first presentation of adjoint logic, we develop some results (in particular, cut elimination and identity expansion), ensuring that adjoint logic is a sensible logic, in a technical sense. Sections 2.2 and 2.3 cover our development (in two steps) of a focused system of adjoint logic, in which proof search is more deterministic, with many of the "non-essential" choices to be made when deciding how to prove a given statement taken away, along with results that ensure that all of our systems of adjoint logic have the same expressive power, in that they can prove the same statements. Finally, in Section 2.4, we show how some examples of logics from the computer science literature can be represented as instances of adjoint logic.

The proof theory in Chapter 2 is presented using the sequent calculus, but the programming languages that we will build in Chapter 4 will make use of the semi-axiomatic sequent calculus, and so we bridge the gap with Chapter 3, where we give a brief overview of what the semi-axiomatic sequent calculus is, and what it looks like in the adjoint case. We also, for completeness, extend the cut elimination result from our prior work [26], which applied to a semi-axiomatic sequent calculus presentation of standard structural intuitionistic logic, to one of our calculi for adjoint logic.

In Chapter 4, we present two programming languages, capturing two different notions of asynchronous communication between concurrently running processes. Since the two languages we work with share much of their theory, and, indeed, their syntax, we first present the shared aspects of these languages (Section 4.2). In Sections 4.3 and 4.4, we first provide a way of understanding this syntax as a language where communication occurs via *asynchronous message passing*, and then a second interpretation where communication instead occurs via writing to and reading from memory, each of which is presented with associated type safety and confluence results. The latter interpretation, via memory, also allows us to make some interesting observations about how these naturally parallel computations can be sequentialized, as well as how they relate to futures, which make up the remainder of this chapter (Section 4.4.2).

Chapter 5 is the final main chapter of this work, and covers notions of program equivalence, focusing on the shared-memory language from Section 4.4. We first examine some background, both on the use cases for and some technical details of handling equivalence, before moving on to define notions of both observational (Section 5.4) and logical (Section 5.5) equivalence, which we show agree for well-typed programs. As a final technical portion of this work, in Section 5.6 we examine a few possible ways to define equivalences that make more use of the adjoint framework, combining multiple different notions of equivalence into one.

In Chapter 6, we conclude, summarizing again the results of the thesis as a whole, and providing an outlook on possible directions for future work.

Chapter 2

Proof Theory of Adjoint Logic

Adjoint logic is a schema for describing logics with a wide variety of features, providing a unifying framework for several common modal and substructural logics, including lax logic [30], the $\Box$ fragment of S4 [75], and linear logic (including !) [35].

The two lines of work that most directly feed into adjoint logic are Benton's mixed linear and non-linear logic LNL [7], and various systems of work on subexponential logics [18, 49, 66, 67]. Both systems generalize linear logic in different ways — LNL is a logic with fully-featured linear and structural layers, connected by an adjoint pair of operators F and G which transport propositions between the layers. Its key generalization from linear logic lies in allowing connectives to be applied directly to propositions in the structural layer, where standard linear logic encodes structural connectives (e.g. the structural implication $\supset$ or $\rightarrow$) as combinations of linear connectives with the exponential !. In this way, LNL treats the linear and non-linear parts of the logic on an equal footing, rather than privileging one or the other as the primary "working area" of the logic. Subexponential logics generalize linear logic in a different direction, retaining the idea of a privileged layer (or zone, or mode, depending on the terminology used) of the logic, which all connectives operate on, but extending beyond linear logic's single exponential ![1] to a family of subexponentials $!_z$, labelled with what layer (zone, mode, ...) they correspond to, the idea being that two propositions $!_a A$ and $!_b A$ may be distinct, and not provably equivalent, even with the same underlying proposition A. Some examples of subexponential logic (e.g. [66]) also allow different subexponentials to provide different structural properties (e.g., one may have an affine $!_a$ that allows propositions of the form $!_a A$ to go unused, but not to be duplicated, while also having a structural $!_b$, which allows propositions to be both duplicated and to go unused).

Our system of adjoint logic attempts to build on both of these generalizations of linear logic, giving a general system for working with more than two distinct layers of logic, which may have a variety of different structural properties and interpretations, as with subexponential logics, but treating these different layers in a uniform fashion, as in LNL. We also find that this generality allows for layers to be related to each other in more varied ways than either LNL (with its fixed two layers) or subexponentials (with a fixed "working zone", through which a proposition must transit to get from one layer to another) provide. However, many of the core concepts and ideas that underlie adjoint logic arise as generalizations of concepts in LNL, subexponentials, or modal

[1] Or, in the more commonly presented classical case, pair of exponentials ! and ?.

logics in general.

The first key idea of adjoint logic, based on the multi-context presentations commonly used in modal logic, is to treat propositions non-uniformly, giving each proposition a *mode of truth* (or just *mode*) m. We think of each mode as representing a separate "base" logic, which may have different ways of proving or using propositions than other modes. Adjoint logic allows us to combine the base logics into a coherent larger logic in which different portions may behave differently. For instance, to model linear logic's ! in adjoint logic, we work with two modes, one of which behaves just as linear logic (without !), and the other of which (representing propositions prefixed with !) is treated structurally (albeit restricted to only allow certain forms of propositions, to ensure that the structural layer only represents propositions of the form $!A$), allowing propositions to be reused freely.

To enable us to join different modes together, we need to describe how the modes are allowed to interact. Letting propositions freely move between modes immediately leads to chaos — for instance, in the example of linear logic with !, it would be possible to duplicate or delete linear propositions by simply lifting them to the mode of !-prefixed propositions. The natural idea that comes from modal and substructural logics is to restrict what modes m hypotheses in a proof of a proposition at mode k can have. For each instance of adjoint logic, we will treat the modes as being drawn from a preordered set, i.e., a set equipped with a transitive and reflexive relation $\leq$. We think of $k \leq m$ as expressing exactly the idea that a proof of a proposition at mode k may depend on hypotheses at mode m (or, dually, $k \not\leq m$ means that a proof of a proposition at mode m *may not* depend on hypotheses at mode k). There does not appear to be any fundamental obstacle to working with some more general structure (and indeed, other approaches to adjoint logic have used more complex structures such as 2-categories [58, 59]), but a preorder is sufficient for the applications that we work with.

The restriction on hypotheses can then be simply described by the following principle, which we call the *declaration of independence*:

A proof of a proposition A_k may only depend on hypotheses B_m for which $m \geq k$.

We will enforce this condition globally, and this means that a well-formed sequent of adjoint logic has the form

$$\Psi \vdash A_k \qquad\qquad \text{where } \Psi \geq k.$$

Here, Ψ is a context, treated for now in the linear manner, so that it represents a multiset of propositions (rather than just a set), and we write $\Psi \geq k$ to mean that each antecedent B_m in Ψ satisfies $m \geq k$. This type of sequent generalizes a common dyadic [2] or two-zone presentation of logics with modal operators [6], where proofs of one type of judgment may be restricted to only depend on a certain type of hypotheses. For instance, proofs of validity may be forbidden to depend on hypotheses dealing with truth [75], or proofs of unrestricted (structural) propositions may be forbidden to depend on linear hypotheses [7].

Just as in these more specific examples, this restriction is not arbitrary, but rather is necessary to ensure that our logic is properly constructed — for instance, to prove cut elimination. This is most simply illustrated with an example with a mode L whose propositions are treated linearly and a mode $\mathsf{U} > \mathsf{L}$ whose propositions are treated structurally, able to be reused.

Example 1 (Necessity of Independence)**.** *Suppose we have propositions A_{L} and B_{U}, and that $A_{\mathsf{L}} \vdash B_{\mathsf{U}}$. We can then construct the following proof (using natural rules for cut and identity — see Section 2.1 for some of the rules that we will actually work with in adjoint logic):*

$$\dfrac{A_{\mathsf{L}} \vdash B_{\mathsf{U}} \qquad \dfrac{}{A_{\mathsf{L}}, B_{\mathsf{U}} \vdash A_{\mathsf{L}}}\,\mathsf{id}}{A_{\mathsf{L}}, A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{cut}$$

With cut, this sequent is easily provable, but it has no cut-free proof unless $A_{\mathsf{L}} \vdash \mathbf{1}_{\mathsf{L}}$.

For a second example, albeit one which relies on rules for at least a few connectives, consider the following (invalid) proof:

$$\dfrac{\dfrac{\dfrac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{id}}{A_{\mathsf{L}} \vdash {\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}}\,{\uparrow}R \qquad \dfrac{\mathsf{C} \in \sigma(\mathsf{U}) \qquad \dfrac{\dfrac{\dfrac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{id}}{{\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,{\uparrow}L \qquad \dfrac{\dfrac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{id}}{{\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,{\uparrow}L}{{\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}, {\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \vdash A_{\mathsf{L}} \otimes A_{\mathsf{L}}}\,{\otimes}R}{{\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \vdash A_{\mathsf{L}} \otimes A_{\mathsf{L}}}\,\mathsf{contract}}{A_{\mathsf{L}} \vdash A_{\mathsf{L}} \otimes A_{\mathsf{L}}}\,\mathsf{cut}$$

Of course, we expect that we should not be able to prove the sequent $A_{\mathsf{L}} \vdash A_{\mathsf{L}} \otimes A_{\mathsf{L}}$ if the mode L *is to reasonably be considered linear. The problem here arises with the first (left) premise of cut, $A_{\mathsf{L}} \vdash {\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}$, highlighted in red — this sequent does not satisfy independence. Reading the proof bottom-up, this is the fault of the cut rule, for not enforcing that its premises are well-formed, even though its conclusion is, and we will see later on that our eventual cut rule will have a constraint to enforce this property, ruling out this example proof.*

Now, to bring propositions from one mode to another, we introduce new connectives, called *shifts* ${\uparrow}^m_k$ and ${\downarrow}^\ell_m$, pronounced *up from k to m* and *down from ℓ to m*, or just *up* and *down* when the modes are either clear from context or unimportant. These are unary connectives, requiring $k \leq m$ and $m \leq \ell$, respectively, and allow us to embed propositions from a lower mode k or a higher mode ℓ into the mode m. The shifts are a generalization of Benton's operators F and G. In several closely related systems, including LNL [7] itself as well as other systems of adjoint logic, [58, 59, 88] these operators (or their equivalents) form an adjunction, with $F \dashv G$ (or, in our notation, ${\downarrow}^k_m \dashv {\uparrow}^k_m$), which is the basis for the name *adjoint logic*. These results suggest that our formulation of adjoint logic, too, can be given a categorical semantics for which we can prove (possibly immediately from the definition) that ${\downarrow} \dashv {\uparrow}$, but such a categorical semantics is out of scope of this work.

As a final piece of our definition of adjoint logic, we restrict what structural rules are available at certain modes. To do this, we make use of a monotone map σ that takes modes to subsets of the two-element set $\{\mathsf{W}, \mathsf{C}\}$, with W representing weakening and C contraction. If $\mathsf{W} \in \sigma(m)$, we say that m admits weakening, and allow weakening of propositions at mode m. Likewise, if $\mathsf{C} \in \sigma(m)$, then m admits contraction. While we always allow the structural rule of exchange, we see no inherent obstacle to a system that restricts exchange as well, as in the system of Licata et al. [59], or that of Kanovich et al. [49].

The propositions at each mode are constructed uniformly, using the syntax of linear logic for connectives, other than the newly added shifts ${\uparrow}^m_k A_k$ and ${\downarrow}^\ell_m A_\ell$. We then have the following syntax for propositions of adjoint logic:

$$A_m, B_m \quad ::= \quad p_m \mid A_m \multimap B_m \mid A_m \otimes B_m \mid \mathbf{1}_m \mid \oplus_{j \in J} A_m^j \mid \&_{j \in J} A_m^j \mid {\uparrow}_k^m A_k \mid {\downarrow}_m^\ell A_\ell$$

Here, p_m stands for atomic m-propositions. We also generalize internal and external choice ($\oplus$ and $\&$) from their usual binary forms to n-ary forms, parameterized by a finite index set J, as this will be more practical from an operational perspective (Chapter 4). With $J = \emptyset$ we recover $\top = \&_{j \in \emptyset}(\,)$ and $\mathbf{0} = \oplus_{j \in \emptyset}(\,)$ in any mode, and it is similarly clear that with a two-element index set, we recover the standard binary $\oplus$ and $\&$. Each mode has access to the same connectives (though we may, in some examples, restrict which connectives are available at specific modes), and the left and right rules for these connectives are uniform across different modes. The differences between modes arise only in which structural rules are permissible [2] and in the rules for the shifts, which have side conditions dealing with the modes. Also note that while technically, we have distinct (but related!) connectives $\multimap_m$ and $\multimap_k$ at each mode, with $\multimap_m$ only operating on m-propositions, for instance, which version of a connective is being used can always be determined from the modes of the propositions to which it is applied, and so we do not explicitly label the connectives.

At this point, we are equipped to present our calculi for adjoint logic.

2.1 ADJ^E: Adjoint Logic with Explicit Structural Rules

The first calculus we examine has explicit structural rules of weakening and contraction. This calculus closely matches the calculi used for intuitionistic linear logic with the exponential ![3], where the rules for weakening and contraction of replicated formulae are explicit, and makes it easy to see what the rules look like at any given mode, and so what rules we have for the restriction of the instance of adjoint logic to a specific mode. For instance, if we have modes $\mathsf{L} \leq \mathsf{U}$, with L being linear ($\sigma(\mathsf{L}) = \emptyset$) and U being structural or unrestricted ($\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$), the rules, when restricted to L, are exactly the rules of purely linear logic (augmented with shifts, which, when restricted to a single mode, are logical no-op connectives). Similarly, when restricting to U, we get the rules of structural logic (as both weakening and contraction will always be allowable at U).

This calculus, which we call ADJ^E, can be found in Figure 2.1. As is common for the sequent calculus, we read the rules in the direction of bottom-up proof construction, and so for each rule, we assume that the conclusion is well-formed (satisfies independence), and add side conditions to enforce that the premises are well-formed as well.

We begin with the judgmental rules of identity and cut, which express the connection between antecedents and succedents. Identity says that given A_m as a hypothesis, we may conclude A_m. Cut says the opposite: if we can conclude A_m, then we are entitled to assume A_m as a hypothesis.

In the cut rule, independence comes into play: if we only assume that the conclusion satisfies independence, we get that $\Psi_1\ \Psi_2 \geq k$, but in order for the premise $\Psi_1 \vdash A_m$ to be well-formed,

[2] When we develop a system in which structural rules are implicit, this will lead to some mode-related side conditions on some of the left and right rules, but these conditions are still stated in a uniform fashion.

[3] Often written as ILL

$$\frac{}{A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Psi_1 \geq m \geq k \quad \Psi_1 \vdash A_m \quad \Psi_2, A_m \vdash C_k}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}$$

$$\frac{\mathsf{W} \in \sigma(m) \quad \Psi \vdash C_k}{\Psi, A_m \vdash C_k}\ \mathsf{weaken} \qquad \frac{\mathsf{C} \in \sigma(m) \quad \Psi, A_m, A_m \vdash C_k}{\Psi, A_m \vdash C_k}\ \mathsf{contract}$$

$$\frac{i \in J \quad \Psi \vdash A_m^i}{\Psi \vdash \oplus_{j \in J} A_m^j}\ \oplus R^i \qquad \frac{\Psi, A_m^j \vdash C_k \quad \text{for all } j \in J}{\Psi, \oplus_{j \in J} A_m^j \vdash C_k}\ \oplus L$$

$$\frac{\Psi \vdash A_m^j \quad \text{for all } j \in J}{\Psi \vdash \&_{j \in J} A_m^j}\ \& R \qquad \frac{i \in J \quad \Psi, A_m^i \vdash C_k}{\Psi, \&_{j \in J} A_m^j \vdash C_k}\ \& L^i$$

$$\frac{\Psi_1 \vdash A_m \quad \Psi_2 \vdash B_m}{\Psi_1, \Psi_2 \vdash A_m \otimes B_m}\ \otimes R \qquad \frac{\Psi, A_m, B_m \vdash C_k}{\Psi, A_m \otimes B_m \vdash C_k}\ \otimes L$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \multimap B_m}\ \multimap R \qquad \frac{\Psi_1 \geq m \quad \Psi_1 \vdash A_m \quad \Psi_2, B_m \vdash C_k}{\Psi_1, \Psi_2, A_m \multimap B_m \vdash C_k}\ \multimap L$$

$$\frac{}{\cdot \vdash \mathbf{1}_m}\ \mathbf{1}R \qquad \frac{\Psi \vdash C_k}{\Psi, \mathbf{1}_m \vdash C_k}\ \mathbf{1}L$$

$$\frac{\Psi \geq \ell \quad \Psi \vdash A_\ell}{\Psi \vdash {\downarrow}_m^\ell A_\ell}\ {\downarrow}R \qquad \frac{\Psi, A_\ell \vdash C_k}{\Psi, {\downarrow}_m^\ell A_\ell \vdash C_k}\ {\downarrow}L$$

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}_k^m A_k}\ {\uparrow}R \qquad \frac{k \geq \ell \quad \Psi, A_k \vdash C_\ell}{\Psi, {\uparrow}_k^m A_k \vdash C_\ell}\ {\uparrow}L$$

Figure 2.1: Adjoint Logic with Explicit Structural Rules (ADJ^E).
We presuppose that the conclusion of each rule satisfies the declaration of independence and ensure, with conditions on modes, that the premises will, too.

we need $\Psi_1 \geq m$, and likewise, for $\Psi_2, A_m \vdash C_k$ to be well-formed, we need $m \geq k$. We combine these into the single premise $\Psi_1 \geq m \geq k$.

The structural rules of weakening and contraction are straightforward — they simply need to check that the mode of the principal formula allows the rule to be used.

The logical rules defining the standard multiplicative and additive connectives are the linear rules for those connectives, regardless of what mode they are at, since we have separated out the structural rules. In all but one case — that of $\multimap L$ — the well-formedness of the conclusion implies the well-formedness of all premises. As for $\multimap L$, we know from the well-formedness of the conclusion that $\Psi_1 \geq k$, $\Psi_2 \geq k$, and $m \geq k$. These facts by themselves already imply the well-formedness of the second premise, but we need to check that $\Psi_1 \geq m$ in order for the first premise ($\Psi_1 \vdash A_m$) to be well-formed.

Finally, we reach the rules for the new shift connectives. Recall that in ${\uparrow}_k^m A_k$ and ${\downarrow}_m^\ell A_\ell$ we require that $k \leq m$ and $m \leq \ell$, which provides additional information for well-formedness. We first consider the two rules for $\uparrow$. We know from the conclusion of ${\uparrow}R$ that $\Psi \geq m$ and from the requirement of the shift that $m \geq k$. Therefore, as $\geq$ is transitive, $\Psi \geq k$ and the premise is always well-formed. This also means that this rule is *invertible*, an observation integrated into the focusing rules for the system ADJ^F presented in Section 2.3.

From the conclusion of ${\uparrow}L$, we know $\Psi \geq \ell$, $m \geq \ell$, and $m \geq k$. This does not imply that $k \geq \ell$, which we need for the premise $\Psi, A_k \vdash C_\ell$ to be well-formed. As such, we need to add $k \geq \ell$ as a premise to the rule, and the rule is non-invertible.

The downshift rules are constructed analogously, taking only the declaration of independence and properties of the preorder $\leq$ as guidance. Note that in this case the left rule is invertible, while the right rule is non-invertible.

At this point we can prove some simple properties of the shifts as illustrative examples. For instance, shifts distribute over implication — given modes $k \leq m$, we can construct the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{A_m \multimap_m B_m, A_m \geq m \qquad \dfrac{A_m \geq m \quad \dfrac{}{A_m \vdash A_m}\,\mathsf{id} \quad \dfrac{}{B_m \vdash B_m}\,\mathsf{id}}{A_m \multimap_m B_m, A_m \vdash B_m}\,{\multimap}\,L}{A_m \multimap_m B_m, A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}R}{A_m \multimap_m B_m, {\downarrow}^m_k A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}L}{{\downarrow}^m_k (A_m \multimap_m B_m), {\downarrow}^m_k A_m \vdash {\downarrow}^m_k B_m}\,{\downarrow}L}{{\downarrow}^m_k (A_m \multimap_m B_m) \vdash {\downarrow}^m_k A_m \multimap_k {\downarrow}^m_k B_m}\,{\multimap}\,R$$

A similar proof shows that the upshift also distributes over implication. We can likewise show that the shifts distribute over the other connectives — for instance, the following proof shows that ${\uparrow}^m_k$ distributes over $\&_{j \in J}$. Note that the premises of $\& R$ (and likewise $\oplus L$) need not all have proofs of the same form, but in this example they do (and so we show a generic case):

$$\dfrac{\dfrac{\dfrac{k \geq k \quad \dfrac{\ell \in J \quad \dfrac{}{A^\ell_k \vdash A^\ell_k}\,\mathsf{id}}{\&_{j \in J} A^j_k \vdash A^\ell_k}\,\&L^\ell}{{\uparrow}^m_k \ \&_{j \in J}\ A^j_k \vdash A^\ell_k}\,{\uparrow}L}{{\uparrow}^m_k \ \&_{j \in J}\ A^j_k \vdash {\uparrow}^m_k A^\ell_k \quad \text{for all } \ell \in J}\,{\uparrow}R}{{\uparrow}^m_k \ \&_{j \in J}\ A^j_k \vdash \&_{j \in J} {\uparrow}^m_k A^j_k}\,\&R$$

Another useful property of shifts (which we make heavy use of in Section 2.3 when working with focusing) is that shifts within a single mode do not affect provability — that is, ${\downarrow}^m_m A_m \dashv\vdash A_m \dashv\vdash {\uparrow}^m_m A_m$. This can be seen by the following four proofs:

$$\dfrac{\dfrac{}{A_m \vdash A_m}\,\mathsf{id}}{{\downarrow}^m_m A_m \vdash A_m}\,{\downarrow}L \qquad\qquad \dfrac{A_m \geq m \quad \dfrac{}{A_m \vdash A_m}\,\mathsf{id}}{A_m \vdash {\downarrow}^m_m A_m}\,{\downarrow}R$$

$$\dfrac{m \geq m \quad \dfrac{}{A_m \vdash A_m}\,\mathsf{id}}{{\uparrow}^m_m A_m \vdash A_m}\,{\uparrow}L \qquad\qquad \dfrac{\dfrac{}{A_m \vdash A_m}\,\mathsf{id}}{A_m \vdash {\uparrow}^m_m A_m}\,{\uparrow}R$$

Two of these proofs require no checking of mode conditions, while in the other two, the mode conditions are trivially satisfied, because the only mode involved is m.

For a final simple example, we show that shifts compose, as long as they are all in the same direction. For instance, if $k \leq m \leq \ell$, then ${\uparrow}_m^\ell {\uparrow}_k^m A_k \dashv\vdash {\uparrow}_k^\ell A_k$ — an upshift may make stops in between its start and end point without affecting the provability of the proposition. This also means that in a mode structure where there are multiple routes upwards from k to ℓ, following any such route yields an equivalent proposition. Of course, the same applies for downshifts as well. The following two proofs demonstrate this for upshifts (and those for downshifts are similar):

$$\dfrac{\dfrac{m \geq k \qquad \dfrac{k \geq k \qquad \dfrac{}{A_k \vdash A_k}\,\mathsf{id}}{{\uparrow}_k^m A_k \vdash A_k}\,{\uparrow}L}{{\uparrow}_m^\ell {\uparrow}_k^m A_k \vdash A_k}\,{\uparrow}L}{{\uparrow}_m^\ell {\uparrow}_k^m A_k \vdash {\uparrow}_k^\ell A_k}\,{\uparrow}R \qquad \dfrac{\dfrac{\dfrac{k \geq k \qquad \dfrac{}{A_k \vdash A_k}\,\mathsf{id}}{{\uparrow}_k^\ell A_k \vdash A_k}\,{\uparrow}L}{{\uparrow}_k^\ell A_k \vdash {\uparrow}_k^m A_k}\,{\uparrow}R}{{\uparrow}_k^\ell A_k \vdash {\uparrow}_m^\ell {\uparrow}_k^m A_k}\,{\uparrow}R$$

2.1.1 Cut Elimination

We now set out to prove that this calculus satisfies cut elimination — that is, that any sequent provable in the calculus also has a proof that does not use the cut rule. This has several useful consequences — in the pure proof theory, it gives that ADJ^E has the subformula property (any provable sequent has a proof mentioning only subformulae of the formulae in the original sequent) and that ADJ^E is consistent (that is, that we cannot prove falsehood $\oplus_{j \in \{\}} A_m^j$, as we can easily see that no cut-free proof of this formula exists). Operationally, we can also take the steps of the cut elimination algorithm as computation steps in a programming language based on adjoint logic (chapter 4).

Because we have an explicit rule of contraction, cut elimination does not follow by a simple structural induction. However, we can follow Gentzen's approach [34] and allow multiple copies of the same proposition to be removed by the cut, which then allows a structural induction argument. To do this, we generalize the rule of cut to a *multicut*,[4] which can remove $n \geq 0$ copies of a proposition *provided that the structural properties of the mode of that proposition allow it*.

To simplify the presentation of this rule, we define the *multiplicities of a mode* m ($\mu(m) \subseteq \mathbb{N}$), specifying what numbers of copies of a proposition at mode m can be cut out by a multicut. This is defined as:

$$\mu(m) = \{n \mid (n = 0 \land \mathsf{W} \in \sigma(m)) \lor n = 1 \lor (n \geq 2 \land \mathsf{C} \in \sigma(m))\}$$

With this notation, we can write down a simple rule of multicut (where A_m^n denotes n copies of

[4] The term "multicut" has been used in the literature for several different rules We follow here the proof theory literature [65, Section 5.1], where it refers to a rule that cuts out some number of copies of the *same* proposition A, as in Gentzen's original proof of cut elimination [34], where he calls it "Mischung", rather than one of the variants that cuts out several propositions together, for instance.

A_m — that is, the context $\underbrace{A_m, \ldots, A_m}_{n \text{ times}}$):

$$\frac{\Psi_1 \geq m \geq k \quad n \in \mu(m) \quad \Psi_1 \vdash A_m \quad \Psi_2, A_m^n \vdash C_k}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}(n)$$

As $1 \in \mu(m)$ always, the cut rule that we presented earlier is simply the $n = 1$ case of the $\mathsf{cut}(n)$ rule. As such, proving that the multicut rule is admissible from ADJ^E (without using cut) also proves that cut is admissible, and likewise, if we can eliminate all multicuts from ADJ^E proofs, we can also eliminate standard cuts.

Beyond this, we also observe that weakening and contraction can be derived as instances of multicut where the first premise always holds (by well-formedness of the conclusion of the multicut), the second premise is exactly the check that weakening or contraction is admissible, as in the weakening or contraction rule, the third premise is proven by an identity, and the fourth premise is the main premise of the weakening (or contraction) rule.

$$\frac{A_m \geq m \geq k \quad \mathsf{W} \in \sigma(m) \quad \dfrac{}{A_m \vdash A_m}\ \mathsf{id} \quad \Psi \vdash C_k}{A_m, \Psi \vdash C_k}\ \mathsf{cut}(0)$$

$$\frac{A_m \geq m \geq k \quad \mathsf{C} \in \sigma(m) \quad \dfrac{}{A_m \vdash A_m}\ \mathsf{id} \quad \Psi, A_m, A_m \vdash C_k}{\Psi, A_m \vdash C_k}\ \mathsf{cut}(2)$$

Moreover, each instance of multicut can be derived from cut, along with weakening or contraction, depending on the choice of n — we show here the $n = 0$ and $n = 2$ examples. If $n = 1$, multicut is just the usual cut, and for $n > 2$, we need to apply contraction multiple ($n - 1$) times, rather than the single application we see when $n = 2$.

$$\frac{\Psi_1 \geq m \geq k \quad \Psi_1 \vdash A_m \quad \dfrac{\mathsf{W} \in \sigma(m) \quad \Psi_2 \vdash C_k}{\Psi_2, A_m \vdash C_k}\ \mathsf{weaken}}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}$$

$$\frac{\Psi_1 \geq m \geq k \quad \Psi_1 \vdash A_m \quad \dfrac{\mathsf{C} \in \sigma(m) \quad \Psi_2, A_m, A_m \vdash C_k}{\Psi_2, A_m \vdash C_k}\ \mathsf{contract}}{\Psi_1, \Psi_2 \vdash C_k}\ \mathsf{cut}$$

Exchanging cut for multicut therefore does not affect provability (in the presence of weakening and contraction). Moreover, we can replace cut, weakening, and contraction all with multicut, giving a system which can prove the same sequents as the original (albeit which no longer satisfies cut elimination — a multicut implementing weakening, for instance, cannot necessarily be eliminated).

We now move on to prove admissibility of multicut, writing $\Psi \Vdash_E A_m$ to mean that there is an ADJ^E proof of $\Psi \vdash A_m$ that uses neither cut nor multicut (but which may use weakening or contraction):

Theorem 1 (Admissibility of multicut in ADJ^E). *If* $\Psi_1 \geq m \geq k$, $n \in \mu(m)$, $\Psi_1 \Vdash_E A_m$, *and* $\Psi_2, A_m^n \Vdash_E C_k$, *then* $\Psi_1, \Psi_2 \Vdash_E C_k$.

Proof. This follows by induction on the (lexicographically ordered) triple $(A_m, \mathcal{D}, \mathcal{E})$, where $\mathcal{D}$ is the proof that $\Psi_1 \Vdash_E A_m$ and $\mathcal{E}$ is the proof that $\Psi_2, A_m^n \Vdash_E C_k$. The cases of this induction fall into 4 main groups, depending on the last rule used in each of $\mathcal{D}$ and $\mathcal{E}$, and whether A_m is the principal formula (where we view the proposition being dropped or duplicated as the principal formula of weakening and contraction, respectively) of these last rules.

Identity Cases The simplest cases are the identity cases, where the last rule used in one of $\mathcal{D}$ or $\mathcal{E}$ is an identity. In these cases, the desired proof is simply the non-identity one of $\mathcal{D}$ and $\mathcal{E}$. For example, if $\mathcal{E}$ ends in an identity, then, because the context must contain only the principal formula of that identity, $\mathcal{E}$ must be the following proof:

$$\frac{}{A_m \vdash_E A_m}\ \mathsf{id}$$

with Ψ_2 being empty, $n = 1$, and $C_k = A_m$. This then means that $\mathcal{D}$, being a proof that $\Psi_1 \Vdash_E A_m$, is already exactly a proof that $\Psi_1, \Psi_2 \Vdash_E C_k$. The case where $\mathcal{D}$ ends in an identity is similarly simple.

Commuting Cases The next group of cases are the commuting cases, where A_m is not the principal formula of at least one of the last rules of $\mathcal{D}$ and $\mathcal{E}$. In these cases, we can, intuitively, "commute" the multicut we are trying to prove admissible past this rule for which A_m is not principal, and our proof that $\Psi_1, \Psi_2 \Vdash_E C_k$ will consist of applying this rule to reach a state where we can apply our inductive hypothesis (potentially multiple times) with the same A_m, but one of $\mathcal{D}$ and $\mathcal{E}$ being smaller. Note that these cases include several where either $\mathcal{D}$ or $\mathcal{E}$ end in a structural rule (weakening or contraction), but A_m is not the principal formula of that rule. We consider two examples of these cases, one in which the rule we commute past is for a connective, and one in which the rule we commute past is a structural rule.

First, consider the case where $\mathcal{D}$ is arbitrary, and

$$\mathcal{E} = \frac{\Psi_3, A_m^i \geq \ell \quad \overset{\mathcal{E}_1}{\Psi_3, A_m^i \Vdash_E B_\ell} \quad \overset{\mathcal{E}_2}{\Psi_4, A_m^j, D_\ell \Vdash_E C_k}}{\Psi_3, \Psi_4, A_m^n, B_\ell \multimap D_\ell \Vdash_E C_k}\ {\multimap}L$$

In this case, $\Psi_2 = \Psi_3, \Psi_4, B_\ell \multimap D_\ell$, and i, j are some natural numbers such that $i + j = n$.

We can then construct the following proof (using our inductive hypothesis once for each of i, j that is non-zero):

$$\frac{(\Psi_1 \text{ if } i > 0), \Psi_3 \geq \ell \quad \overset{\text{i.h.}(A_m, \mathcal{D}, \mathcal{E}_2)}{(\Psi_1 \text{ if } i > 0), \Psi_3 \Vdash_E B_\ell} \quad \overset{\text{i.h.}(A_m, \mathcal{D}, \mathcal{E}_2)}{(\Psi_1 \text{ if } j > 0), \Psi_4, D_\ell \Vdash_E C_k}}{\Psi_1, \Psi_3, \Psi_4, B_\ell \multimap D_\ell \Vdash_E C_k}\ {\multimap}L$$

That $\Psi_3 \geq \ell$ follows immediately from $\Psi_3, A_m^i \geq \ell$ in $\mathcal{E}$. If $i > 0$, we can also conclude from this that $m \geq \ell$, and since $\Psi_1 \geq m$, we then also have that $\Psi_1 \geq \ell$ in this case. The other two premises of this $\multimap L$ rule are either exactly $\mathcal{E}_1$ or $\mathcal{E}_2$ if i or j are zero, respectively, or the result of applying the inductive hypothesis to these smaller proofs (as shown above). In either case, we have the desired proof.

Now, consider the case where $\mathcal{E}$ is arbitrary, and

$$\mathcal{D} = \dfrac{\mathsf{W} \in \sigma(\ell) \quad \overset{\mathcal{D}_1}{\Psi_3 \Vdash_E A_m}}{\Psi_3, B_\ell \Vdash_E A_m}\ \textsf{weaken}$$

In this case, $\Psi_1 = \Psi_3, B_\ell$, and we can construct the following proof:

$$\dfrac{\mathsf{W} \in \sigma(\ell) \qquad \overset{\text{i.h.}(A_m, \mathcal{D}_1, \mathcal{E})}{\Psi_3 \Vdash_E C_k}}{\Psi_3, B_\ell, \Psi_2 \Vdash_E C_k}\ \textsf{weaken}$$

Despite the fact that the rule we are commuting our cut past is a structural rule, we can treat it identically to one of the rules for a connective.

The other commuting cases, which make up the bulk of the cases of this induction, are similar to those shown, without any substantial proof needed.

Principal Structural Cases The first set of cases we consider where A_m is the principal formula of the rule we are examining are those where $\mathcal{E}$ ends in a structural rule which has A_m as its principal formula. In these cases, we are able to combine this structural rule into the multicut we are attempting to show admissible, changing the number of copies of A_m that the cut creates, and allowing us to conclude by our induction hypothesis, applied to $(A_m, \mathcal{D}, \mathcal{E}_1)$, where $\mathcal{E}_1$ is a subproof of $\mathcal{E}$.

First, we examine the case of weakening, where $\mathcal{D}$ is arbitrary and

$$\mathcal{E} = \dfrac{\mathsf{W} \in \sigma(m) \quad \overset{\mathcal{E}_1}{\Psi_2 \Vdash_E C_k}}{\Psi_2, A_m \Vdash_E C_k}\ \textsf{weaken}$$

In this case, we know that $\Psi_1 \geq m \geq k$ (by assumption), that $0 \in \mu(m)$ (since $\mathsf{W} \in \sigma(m)$), that $\Psi \Vdash_E A_m$ (from $\mathcal{D}$), and that $\Psi_2 \Vdash_E C_k$ (from $\mathcal{E}_1$), and can therefore apply our inductive hypothesis with $n = 0$ to $(A_m, \mathcal{D}, \mathcal{E}_1)$ to get that $\Psi_1, \Psi_2 \Vdash_E C_k$, as desired.

Now, consider the case of contraction, where $\mathcal{D}$ is again arbitrary, and

$$\mathcal{E} = \dfrac{\mathsf{C} \in \sigma(m) \quad \overset{\mathcal{E}_1}{\Psi_2, A_m, A_m \Vdash_E C_k}}{\Psi_2, A_m \Vdash_E C_k}\ \textsf{contract}$$

As in the previous case, our assumptions, $\mathcal{D}$, and $\mathcal{E}_1$ give us all that we need to apply our inductive hypothesis with the same n, but $\mathcal{E}_1$ uses one more copy of A_m than $\mathcal{E}$ did, and so we need to know that $n + 1 \in \mu(m)$. However, since $\mathsf{C} \in \sigma(m)$, we know that $\mu(m)$ contains all natural numbers greater than 0 — in particular, it contains $n + 1$, which cannot be 0. We again reach the desired result immediately from applying the inductive hypothesis.

While these cases would present difficulty were we working only with a normal cut, our use of multicut makes them not only provable, but almost immediate.

Principal Connective Cases The final cases we need to consider are those where both $\mathcal{D}$ and $\mathcal{E}$ end in connective rules for which A_m is a principal formula. If either $\mathcal{D}$ or $\mathcal{E}$ end in an identity or structural rule, one of the previous three cases applies, and if both end in connective rules, but A_m is not principal in one of these rules, then a commuting case applies. These cases are therefore exhaustive, covering all possible A_m, $\mathcal{D}$ and $\mathcal{E}$.

The principal connective cases contain the bulk of the actual content of the cut admissibility proof. Here, rather than simply moving upwards in the structure of $\mathcal{D}$ or $\mathcal{E}$ and applying the inductive hypothesis at the same A_m but with smaller proofs, we will break down A_m into some number of smaller propositions, at which we may then apply the inductive hypothesis, even with proofs that are not subproofs of or structurally smaller than $\mathcal{D}$ or $\mathcal{E}$.

We will examine here several cases — first $\oplus$ as a simple example to illustrate the general structure of these proofs, then both shifts, to examine what impact the modes have on the proof, and finally $\multimap$, whose left rule also has a condition on modes, and which also illustrates how the proof structure works when rules have multiple (non-uniform, unlike for $\oplus$ and $\&$) premises.

We begin with the case of $\oplus$, where

$$\mathcal{D} = \frac{i \in J \quad \overset{\mathcal{D}_1}{\Psi_1 \Vdash_E A^i_m}}{\Psi_1 \Vdash_E \oplus_{j\in J} A^j_m}\oplus R^i \qquad \text{and} \qquad \mathcal{E} = \frac{\overset{\mathcal{E}_j}{\Psi_2, \left(\oplus_{j\in J}A^j_m\right)^{n-1}, A^j_m \Vdash_E C_k} \quad \text{for all } j \in J}{\Psi_2, \left(\oplus_{j\in J}A^j_m\right)^{n-1}, \oplus_{j\in J}A^j_m \Vdash_E C_k}\oplus L$$

Note that we cannot have $n = 0$ in this case, as then the principal formula of $\mathcal{E}$ would (even if it were also an instance of $\oplus_{j\in J}A^j_m$) not be the cut formula, and this can be handled as a commuting case. We therefore only consider $n \geq 1$ here.

If $n = 1$, then we can apply the inductive hypothesis to $(A^i_m, \mathcal{D}_1, \mathcal{E}_i)$, and this immediately gives our desired result.

If $n > 1$, since we know by assumption that $n \in \mu(m)$, it must be the case that $\mathsf{C} \in \sigma(m)$, and also that $n - 1 \in \mu(m)$. We may therefore apply the inductive hypothesis to $(\oplus_{j\in J}A^j_m, \mathcal{D}, \mathcal{E}_i)$, giving us the following proof, which we call $\mathcal{E}'$:

$$\frac{\Psi_1 \geq m \geq k \quad n-1 \in \mu(m) \quad \overset{\mathcal{D}}{\Psi_1 \Vdash_E \oplus_{j\in J}A^j_m} \quad \overset{\mathcal{E}_i}{\Psi_2, \left(\oplus_{j\in J}A^j_m\right)^{n-1}, A^i_m \Vdash_E C_k}}{\Psi_1, \Psi_2, A^i_m \Vdash_E C_k}\ \text{i.h.}(\left(\oplus_{j\in J}A^j_m\right), \mathcal{D}, \mathcal{E}_i)$$

Now, applying the inductive hypothesis again, at $(A^i_m, \mathcal{D}_1, \mathcal{E}')$, we get the following proof:

$$\frac{\Psi_1 \geq m \geq k \quad 1 \in \mu(m) \quad \overset{\mathcal{D}_1}{\Psi_1 \Vdash_E A^i_m} \quad \overset{\mathcal{E}'}{\Psi_1, \Psi_2, A^i_m \Vdash_E C_k}}{\Psi_1, \Psi_1, \Psi_2 \Vdash_E C_k}\ \text{i.h.}(A^i_m, \mathcal{D}_1, \mathcal{E}')$$

This is almost the desired result, but has two copies of Ψ_1, as a result of our two uses of the inductive hypothesis. To resolve this, we note that $\Psi_1 \geq m$ and $\mathsf{C} \in \sigma(m)$, so it must also be the case that $\mathsf{C} \in \sigma(\Psi_1)$, and so we can apply the contraction rule repeatedly to propositions in Ψ_1 to remove the extra copies, giving the desired proof that $\Psi_1, \Psi_2 \Vdash_E C_k$.

In the remaining cases, we will elide the details of handling of this multiplicity n, but all are similar, involving multiple invocations of the inductive hypothesis, and then contracting to remove excess copies of Ψ_1 from the resulting proof.

We now examine the shifts, first taking $A_m = \downarrow^{\ell}_{m} A_\ell$. In this principal case, we have

$$\mathcal{D} = \dfrac{\Psi_1 \geq \ell \quad \overset{\mathcal{D}_1}{\Psi_1 \Vdash_E A_\ell}}{\Psi_1 \Vdash_E \downarrow^{\ell}_{m} A_\ell} \downarrow R \qquad \text{and} \qquad \mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Psi_2, \left(\downarrow^{\ell}_{m} A_\ell\right)^{n-1}, A_\ell \Vdash_E C_k}}{\Psi_2, \left(\downarrow^{\ell}_{m} A_\ell\right)^{n-1}, \downarrow^{\ell}_{m} A_\ell \Vdash_E C_k} \downarrow L$$

From $\mathcal{D}$, we have $\Psi_1 \geq \ell$, and since $\downarrow^{\ell}_{m} A_\ell$ is well-formed, $\ell \geq m$. Thus, $\Psi_1 \geq \ell \geq k$, and so we may apply the inductive hypothesis to $A_\ell, \mathcal{D}_1, \mathcal{E}_1$ to reach the desired result: [5]

$$\dfrac{\Psi_1 \geq \ell \geq k \quad 1 \in \mu(\ell) \quad \overset{\mathcal{D}_1}{\Psi_1, \Vdash_E A_\ell} \quad \overset{\mathcal{E}_1}{\Psi_2, A_\ell \Vdash_E C_k}}{\Psi_1, \Psi_2, \Vdash_E C_k} \text{ i.h.}(A_\ell, \mathcal{D}_1, \mathcal{E}_1)$$

In the principal case where $A_m = \uparrow^{m}_{\ell} A_\ell$, we have

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Psi_1 \Vdash_E A_\ell}}{\Psi_1 \Vdash_E \uparrow^{m}_{\ell} A_\ell} \uparrow R \qquad \text{and} \qquad \mathcal{E} = \dfrac{\ell \geq k \quad \overset{\mathcal{E}_1}{\Psi_2, (\uparrow^{m}_{\ell} A_\ell)^{n-1}, A_\ell \Vdash_E C_k}}{\Psi_2, (\uparrow^{m}_{\ell} A_\ell)^{n-1}, \uparrow^{m}_{\ell} A_\ell \Vdash_E C_k} \uparrow L$$

Since $\uparrow^{m}_{\ell} A_\ell$ is well-formed, we have $m \geq \ell$, which, combined with the assumption that $\Psi_1 \geq m \geq k$, gives us that $\Psi_1 \geq m \geq \ell$. Adding also that $\ell \geq k$ (from $\mathcal{E}$), we can conclude that $\Psi_1 \geq \ell \geq k$, enabling us to apply the inductive hypothesis to $A_\ell, \mathcal{D}_1, \mathcal{E}_1$ to get the desired result:

$$\dfrac{\Psi_1 \geq \ell \geq k \quad 1 \in \mu(\ell) \quad \overset{\mathcal{D}_1}{\Psi_1 \Vdash_E A_\ell} \quad \overset{\mathcal{E}_1}{\Psi_2, A_\ell \Vdash_E C_k}}{\Psi_1, \Psi_2 \Vdash_E C_k} \text{ i.h.}(A_\ell, \mathcal{D}_1, \mathcal{E}_1)$$

We now examine our final example case, where $A_m = B_m \multimap D_m$. In this case, we have

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Psi_1, B_m \Vdash_E D_m}}{\Psi_1 \Vdash_E B_m \multimap D_m} \multimap R$$

and, writing $\Psi_2 = \Psi_3, \Psi_4$ and taking i, j to be natural numbers with $i + j = n - 1$,

$$\mathcal{E} = \dfrac{\Psi_3, (B_m \multimap D_m)^i \geq m \quad \overset{\mathcal{E}_1}{\Psi_3, (B_m \multimap D_m)^i \Vdash_E B_m} \quad \overset{\mathcal{E}_2}{\Psi_4, (B_m \multimap D_m)^j, D_m \Vdash_E C_k}}{\Psi_3, \Psi_4, (B_m \multimap D_m)^{n-1}, B_m \multimap D_m \Vdash_E C_k} \multimap L$$

If $n = 1$, then i, j must both be zero, and we can build the following proof:

$$\dfrac{\Psi_1, \Psi_3 \geq m \geq k \quad \mathcal{F} \quad \overset{\mathcal{E}_2}{\Psi_4, D_m \Vdash_E C_k}}{\Psi_1, \Psi_3, \Psi_4 \Vdash_E C_k} \text{ i.h.}(D_m, \ldots, \mathcal{E}_2)$$

[5] In the case where $n = 1$ — for $n > 1$, we first need to apply the inductive hypothesis to remove the extra copies of $\downarrow^{\ell}_{m} A_\ell$, and our resulting proof will require us to contract repeatedly to remove excess copies of Ψ_1, as with $\oplus$.

where

$$\mathcal{F} = \dfrac{\Psi_3 \geq m \qquad \overset{\mathcal{E}_1}{\Psi_3 \Vdash_E B_m} \qquad \overset{\mathcal{D}_1}{\Psi_1, B_m \Vdash_E D_m}}{\Psi_1, \Psi_3 \Vdash_E D_m}\ \text{i.h.}(B_m, \mathcal{E}_1, \mathcal{D}_1)$$

Both uses of the inductive hypothesis here are justified, as they are on B_m and D_m, which are both subformulae of $A_m = B_m \multimap D_m$, and so the proofs to which they are applied need not be smaller than $\mathcal{D}$ and $\mathcal{E}$. For space reasons we have omitted several side conditions on these uses of the inductive hypothesis which are trivially true — e.g., that $m \geq m$ in the first usage on B_m, and that $1 \in \mu(m)$ in both cases. We get that $\Psi_3 \geq m$ from $\mathcal{E}$, which is sufficient to give the conditions on modes needed for the inductive hypothesis in this case.

As in the previous cases, if $n > 1$, then we need to do some extra work to deal with the extra copies of $B_m \multimap D_m$. Since $n \in \mu(m)$ by assumption, we know that $\mathsf{C} \in \sigma(m)$, and so $\mu(m)$ contains all positive natural numbers.

If $i > 0$, then we can use the inductive hypothesis at $(B_m \multimap D_m, \mathcal{D}, \mathcal{E}_1)$ to get a proof $\mathcal{E}'_1$ that $\Psi_1, \Psi_3 \Vdash_E B_m$. Likewise, if $j > 0$, we get a proof $\mathcal{E}'_2$ that $\Psi_1, \Psi_4, D_m \Vdash_E C_k$. There are several case distinctions depending on the values of i and j, but regardless, we are able to create a proof with the same structure as in the $n = 1$ case, but with two or three copies of Ψ_1. As with $\oplus$, we can apply contraction repeatedly to remove these excess copies, leaving the desired result.

The cases for the remaining connectives follow the pattern of those shown here. □

Admissibility of multicut then yields cut elimination in its usual form, by tracing through the given proof and using the admissibility of multicut to remove cuts wherever necessary.

Theorem 2 (Cut elimination for ADJ^E)**.** *If* $\Psi \vdash_E A_m$*, then* $\Psi \Vdash_E A_m$*.*

These proofs, like many others involving deductive systems, involves many cases, only a few of which are shown here, and many of which follow similar patterns. While we endeavor to make sure that enough cases of such proofs are shown to be convincing, there is always a risk of omitting a case that readers might find useful or interesting, or, even worse, failing to prove a case and never noticing. Proof assistants, also called interactive theorem provers (e.g., Coq [8, 103], Lean [21], Twelf [78], among many other examples), provide computer support for these problems, by mechanically checking that a given proof is complete. This gives a guarantee both to the proof-writer and the proof-reader that the proof is indeed correct (although both need to ensure that they are careful to check exactly what statement was proven, as the translation from a natural-language theorem to a formalization in one of these systems is often not straightforward).

Formalizing theorems in these tools is, however, often nontrivial, especially as the theorem statements become more complex. Moreover, some pieces of reasoning that can be simple in the pen-and-paper world (or, at least, whose fine details are often ignored), such as α-equivalence and substitution can be challenging to deal with in proof assistants. In particular, trying to explicitly manage a context, particularly a substructural context, in a proof assistant often leads to difficulties proving seemingly basic results about concepts such as substitution. Higher-order abstract syntax (HOAS) [76] provides one solution to this, avoiding explicit handling of variables, binding, and context management by representing each of these in the *object language* being reasoned about (adjoint logic, in the context of the theorem above) with the same concept in the *metalanguage* being used for the proof (for our purposes, a proof assistant's internal language, or a logical framework). However, in its initial form, the ideas of higher-order abstract syntax are insufficient to handle substructural logics or languages, since, for instance, the object language's context is identified with that of the metalanguage, and as such, if the metalanguage's context is treated structurally (i.e., as a set), then so too will that in the object language. There is some existing work [19, 95] addressing this, using alternate methods to track the usage of hypotheses in a context and enforce substructural constraints such as linearity. This last approach seems particularly promising for reasoning about adjoint logic, particularly as Crary [19] notes that affine and strict logics may be handled equally well as the linear logic that his paper focuses on. We believe that a formalization in such a system of the results in this thesis (not only the above theorem, but the many others of similar complexity) would be a valuable piece of future work, but one that is potentially quite large in scope.

2.1.2 Identity expansion

Identity expansion for ADJ^E is standard in its statement and proof.

Theorem 3 (Identity Expansion)**.** *If $\Psi \vdash_E A_m$, then there exists a proof that $\Psi \vdash_E A_m$ using identity rules only at atomic propositions p_m, which is cut-free if the original proof is.*

Proof. We begin by proving that for any formula A_m, there is a cut-free proof that $A_m \vdash_E A_m$ using identity rules only at atomic propositions. This follows easily from an induction on A_m. Now, we arrive at the theorem by induction over the structure of the given proof that $\Psi \vdash_E A_m$, applying the above result to remove any identities on non-atomic propositions. □

2.2 ADJ^I: Making Structural Rules Implicit

In order to move towards an eventual focused system (Section 2.3), we loosely follow the approach of Andreoli [2], eliminating some of the nondeterminism in proof search by removing the structural rules, instead building weakening and contraction into the other rules. A side benefit of this intermediate system ADJ^I, whose rules can be found in Figure 2.2, is that it is well-suited to embedding logics whose standard presentations similarly leave structural rules implicit, as many structural modal logics do.

In several rules, we would like to know that a whole context of propositions satisfies some structural property, to either implicitly weaken or contract several times at once. To describe these conditions, we define for contexts Ψ the set $\sigma(\Psi)$ of structural properties shared by all propositions in Ψ. Intuitively, W is a member of $\sigma(\Psi)$ if and only iff all propositions in Ψ are weakenable, and likewise for contraction.

Definition 1. *We define $\sigma(\Psi)$ inductively as follows:*

$$\begin{array}{rcl}
\sigma(\cdot) & = & \{\mathsf{W}, \mathsf{C}\} \\
\sigma(A_m) & = & \sigma(m) \\
\sigma(\Psi_1, \Psi_2) & = & \sigma(\Psi_1) \cap \sigma(\Psi_2)
\end{array}$$

There are a few key differences between ADJ^E and Andreoli's Σ_1. First, we allow for modes to have weakening without contraction and vice versa, whereas Σ_1 allows either neither (for normal propositions) or both (for propositions of the form $?A$). Second, we have the additional shift connectives to consider. Most of the shift rules are straightforward to turn into this implicit form, but $\downarrow R$ requires some thought because of its restriction on the modes allowed in its context. In ADJ^E, we can weaken away any A_m with $\mathsf{W} \in \sigma(m)$ before applying $\downarrow R$ in order to satisfy that restriction. In order to match that behavior, in the $\downarrow R$ rule of ADJ^I, we split the context into two pieces, Ψ_1 and Ψ_2, and require that $\Psi_1 \geq m$, while $\mathsf{W} \in \sigma(\Psi_2)$. This rule then corresponds to the ADJ^E proof which weakens everything in Ψ_2 and then applies $\downarrow R$. Weakening is otherwise easily handled at the leaves of the proof (id and $\mathbf{1}R$) in a similar manner.

Contraction without weakening leads to most of the complication in this system. Were contraction to always imply weakening, we could simply propagate contractible propositions to all branches of the proof and weaken them if they are not needed, as is done in standard intuitionistic logic, or as with Andreoli's second context Θ. Instead, for each multiplicative rule with two premises ($\otimes R$ and $\multimap L$), as well as for the cut rule, we split the context into three parts, sending Ψ_1 to the first premise only, Ψ_3 to the second premise only, and Ψ_2 to both. Of course, this also requires that all $\mathsf{C} \in \sigma(\Psi_2)$, so all propositions in Ψ_2 can safely be duplicated to both premises. The nondeterminism in how Ψ_2 is chosen allows us to propagate contractible propositions to precisely those premises where they will be needed. Similarly, we allow (but do not require) the principal formula to be preserved in the premises after applying a left rule. To this end, we split each left rule into two versions, labelled with $\alpha \in \{0, 1\}$ (or four versions, labelled with $\alpha, \beta \in \{0, 1\}$ in one case). Each has $(A_m)^\alpha$ in its premise, where $(A_m)^1$ is A_m and $(A_m)^0$ is the empty context. The rule with $\alpha = 1$ thus preserves the principal formula, while the rule with $\alpha = 0$ consumes it. In order for this preservation of the principal formula A_m to be allowed, we must have $\mathsf{C} \in \sigma(m)$, and so a side condition of each $\alpha = 1$ rule variant is that $\mathsf{C} \in \sigma(m)$ (in the case of $\multimap L^{\alpha,\beta}$, we need this side condition for the three cases where either α or β is 1). These changes, along with the removal of the explicit weakening and contraction rules, give us ADJ^I, as shown in Figure 2.2.

2.2.1 Equivalence of ADJ^I and ADJ^E

In order to justify that ADJ^I and ADJ^E are different presentations of the same logic, we want to show that ADJ^I is sound and complete with respect to ADJ^E. That it is sound is nearly immediate, as each rule of ADJ^I is derivable in ADJ^E. Completeness follows from Lemma 1, as all rules of ADJ^E other than the structural rules of weakening and contraction are derivable in ADJ^I. One interesting (though unsurprising) feature of the translations from the proofs of soundness and completeness is that both take cut-free proofs to cut-free proofs, as well as translating identities at A_m to identities at A_m. As such, we get cut elimination and identity expansion for ADJ^I for free via these translations and our results for ADJ^E.

Theorem 4 (Soundness of ADJ^I)**.** *If $\Psi \vdash_I A_m$, then $\Psi \vdash_E A_m$. Moreover, if $\Psi \Vdash_I A_m$, then $\Psi \Vdash_E A_m$, and if the proof in ADJ^I uses identity only at atomic propositions, then so does the resulting ADJ^E proof.*

Proof. By induction over the proof of $\Psi \vdash_I A_m$. At each step, we replace an ADJ^I rule with its ADJ^E equivalent, potentially also using some number of weakenings or contractions. □

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi, A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Psi_1, \Psi_2 \geq m \geq k \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash A_m \quad \Psi_2, \Psi_3, A_m \vdash C_k}{\Psi_1, \Psi_2, \Psi_3 \vdash C_k}\ \mathsf{cut}$$

$$\frac{i \in J \quad \Psi \vdash A_m^i}{\Psi \vdash \oplus_{j \in J} A_m^j}\ \oplus R^i \qquad \frac{\Psi, (\oplus_{j \in J} A_m^j)^\alpha, A_m^j \vdash C_k \quad \text{for all } j \in J}{\Psi, \oplus_{j \in J} A_m^j \vdash C_k}\ \oplus L^\alpha$$

$$\frac{\Psi \vdash A_m^j \quad \text{for all } j \in J}{\Psi \vdash \&_{j \in J} A_m^j}\ \& R \qquad \frac{i \in J \quad \Psi, (\&_{j \in J} A_m^j)^\alpha, A_m^i \vdash C_k}{\Psi, \&_{j \in J} A_m^j \vdash C_k}\ \& L^{i,\alpha}$$

$$\frac{\mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash A_m \quad \Psi_2, \Psi_3 \vdash B_m}{\Psi_1, \Psi_2, \Psi_3 \vdash A_m \otimes B_m}\ \otimes R \qquad \frac{\Psi, (A_m \otimes B_m)^\alpha, A_m, B_m \vdash C_k}{\Psi, A_m \otimes B_m \vdash C_k}\ \otimes L^\alpha$$

$$\frac{\Psi, A_m \vdash B_m}{\Psi \vdash A_m \multimap B_m}\ \multimap R$$

$$\frac{\Psi_1, \Psi_2 \geq m \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2, (A_m \multimap B_m)^\alpha \vdash A_m \quad \Psi_2, \Psi_3, (A_m \multimap B_m)^\beta, B_m \vdash C_k}{\Psi_1, \Psi_2, \Psi_3, A_m \multimap B_m \vdash C_k}\ \multimap L^{\alpha,\beta}$$

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \vdash \mathbf{1}_m}\ \mathbf{1}R \qquad \frac{\Psi, (\mathbf{1}_m)^\alpha \vdash C_k}{\Psi, \mathbf{1}_m \vdash C_k}\ \mathbf{1}L^\alpha$$

$$\frac{\Psi_1 \geq m \quad \mathsf{W} \in \sigma(\Psi_2) \quad \Psi_1 \vdash A_m}{\Psi_1, \Psi_2 \vdash {\downarrow}_k^m A_m}\ {\downarrow}R \qquad \frac{\Psi, ({\downarrow}_k^m A_m)^\alpha, A_m \vdash C_\ell}{\Psi, {\downarrow}_k^m A_m \vdash C_\ell}\ {\downarrow}L^\alpha$$

$$\frac{\Psi \vdash A_k}{\Psi \vdash {\uparrow}_k^m A_k}\ {\uparrow}R \qquad \frac{k \geq \ell \quad \Psi, ({\uparrow}_k^m A_k)^\alpha, A_k \vdash C_\ell}{\Psi, {\uparrow}_k^m A_k \vdash C_\ell}\ {\uparrow}L^\alpha$$

Figure 2.2: Adjoint Logic with Implicit Structural Properties (ADJ^I).
We presuppose that the conclusion of each rule satisfies the declaration of independence and ensure, with conditions on modes, that the premises will, too.
α and β range over $\{0, 1\}$. $(A_m)^0$ always denotes the empty context, while $(A_m)^1$ denotes A_m.
If α or β are 1, then $\mathsf{C} \in \sigma(m)$ should be treated as an additional premise of the rule.

Lemma 1 (Admissibility of weakening and contraction for ADJ^I)**.**

1. If $\Psi \vdash_I C_k$ and $\mathsf{W} \in \sigma(m)$, then $\Psi, A_m \vdash_I C_k$.

2. If $\Psi, A_m, A_m \vdash_I C_k$ and $\mathsf{C} \in \sigma(m)$, then $\Psi, A_m \vdash_I C_k$.

Moreover, the resulting proof is structurally identical to the original proof in both cases — that is, it uses the same sequence of rules, applied to the same propositions at each step.

Proof. For weakening, it nearly suffices to just add A_m to every context in the proof, deferring weakening to the leaves, where it is accomplished by either id or $\mathbf{1}R$ allowing arbitrary weakenable contexts. However, ${\downarrow}R$ complicates this slightly — A_m may not be permitted in Ψ_1. We can solve this problem by placing A_m in the Ψ_2 context of ${\downarrow}R$, or we can defer weakening by placing A_m in Ψ_1 if the mode restrictions allow, and placing it in Ψ_2 otherwise. Either way, we end up with the desired structurally identical proof.

Contraction presents additional challenges — the usual strategy is to simply preserve A_m

whenever it is used (by a left rule, for instance), and to propagate it to all premises of multi-premise rules. However, this leaves an extra copy of A_m at each leaf of the proof, and if $\sigma(m) = \{\mathsf{C}\}$, we are unable to weaken away this extra copy. We therefore must instead be more precise, only propagating A_m to the premises which actually use it. □

Theorem 5 (Completeness of ADJ^I)**.** *If $\Psi \vdash_E A_m$ then $\Psi \vdash_I A_m$. Moreover, if $\Psi \Vdash_E A_m$, then $\Psi \Vdash_I A_m$, and if the proof in ADJ^E uses identity only at atomic propositions, then so does the resulting ADJ^I proof.*

Proof. By induction over the proof of $\Psi \vdash_E A_m$. Each rule of ADJ^E other than weakening and contraction is already an instance of a corresponding rule of ADJ^I, and weakening and contraction can be replaced using lemma 1. □

2.2.2 Cut elimination and identity expansion for ADJ^I, directly

While we get cut elimination and identity expansion for free in ADJ^I from the details of soundness and completeness, these results are not very enlightening, due to their indirectness. We will briefly note here that we can provide direct proofs of both cut elimination and identity expansion for ADJ^I. Both follow standard methods, and so we will not dwell on them.

Theorem 6 (Identity Expansion)**.** *If $\Psi \vdash_I A_m$, then there exists a proof that $\Psi \vdash_I A_m$ using identity rules only at atomic propositions p_m, which is cut-free if the original proof is.*

Proof. This follows in a standard way by induction over A_m, using paired left and right rules to push the identity up to subformulae of A_m. □

Theorem 7 (Admissibility of cut in ADJ^I)**.** *Suppose that $\Psi_1, \Psi_2 \geq m \geq k$, and $\mathsf{C} \in \sigma(\Psi_2)$. If $\Psi_1, \Psi_2 \Vdash_I A_m$ and $\Psi_2, \Psi_3, A_m \Vdash_I C_k$, then also $\Psi_1, \Psi_2, \Psi_3 \Vdash_I C_k$.*

Proof. Let $\mathcal{D}$ be the given proof that $\Psi_1, \Psi_2 \Vdash_I A_m$, and $\mathcal{E}$ the proof that $\Psi_2, \Psi_3, A_m \Vdash_I C_k$.

As with ADJ^E, admissibility of cut for ADJ^I follows from an induction on the (lexicographically ordered) triple $(A_m, \mathcal{D}, \mathcal{E})$, in much the standard manner used to prove admissibility of cut in intuitionistic logic, using cross-cuts to handle implicit contraction if necessary.

We recall the split into identity, commuting, and principal cases (noting that the structural cases of ADJ^E do not appear here, because we do not have explicit structural rules).

Identity Cases We examine one identity case, noting that the three-way split of the context for cut means we have to distinguish some additional subcases, depending on which of Ψ_1, Ψ_2, Ψ_3 the identity rule pulls A_m from, but that these subcases all have a similar structure, relying on admissibility of structural rules for ADJ^I (Lemma 1).

Suppose $\mathcal{E}$ is arbitrary and

$$\mathcal{D} = \cfrac{\mathsf{W} \in \sigma(\Psi_1', \Psi_2')}{\Psi_1', \Psi_2', A_m \Vdash_I A_m}\ \mathsf{id}$$

where either $\Psi_1 = \Psi_1', A_m$ and $\Psi_2 = \Psi_2'$ or $\Psi_1 = \Psi_1'$ and $\Psi_2 = \Psi_2', A_m$.

Then, we wish to prove $\Psi_1, \Psi_2, \Psi_3 \Vdash C_k$. If $\Psi_1 = \Psi_1', A_m$, we may use admissibility of weakening for ADJ^I to weaken away all of Ψ_1', and we are left with $\Psi_2, \Psi_3, A_m \Vdash_I C_k$ as a goal, which is given exactly by $\mathcal{E}$. If, instead, $\Psi_2 = \Psi_2', A_m$, we again use admissibility of weakening to weaken away all of Ψ_1, and are left with $\Psi_2', \Psi_3, A_m \Vdash_I C_k$ as a goal. Applying admissibility of contraction to duplicate A_m (permissible because A_m was in Ψ_2, all of which is contractible), we get a goal of $\Psi_2, \Psi_3, A_m \Vdash_I C_k$, which again is exactly $\mathcal{E}$.

Commuting Cases We also examine a single commuting case for illustration.

Suppose $\mathcal{E}$ is arbitrary and

$$\mathcal{D} = \frac{\ell \geq m \quad \begin{matrix}\mathcal{D}_1 \\ \Psi_1', \Psi_2', (\uparrow^p_\ell B_\ell)^\alpha, B_\ell \Vdash_I A_m\end{matrix}}{\Psi_1', \Psi_2', \uparrow^p_\ell B_\ell \Vdash_I A_m} \uparrow L^\alpha$$

As in the example identity case, either $\Psi_1 = \Psi_1', \uparrow^p_\ell B_\ell$ or $\Psi_2 = \Psi_2', \uparrow^p_\ell B_\ell$. If $\uparrow^p_\ell B_\ell$ was taken from Ψ_1, then $\Psi_2 = \Psi_2'$ and we can construct the following proof:

$$\frac{\ell \geq k \quad \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi_1', \Psi_2, (\uparrow^p_\ell B_\ell)^\alpha, B_\ell \Vdash_I A_m\end{matrix} \quad \begin{matrix}\mathcal{E} \\ \Psi_2, \Psi_3, A_m \Vdash_I C_k\end{matrix}}{\Psi_1', \Psi_2, \Psi_3, (\uparrow^p_\ell B_\ell)^\alpha, B_\ell \Vdash_I C_k} \text{ i.h.}(A_m, \mathcal{D}_1, \mathcal{E})}{\Psi_1', \Psi_2, \Psi_3, \uparrow^p_\ell B_\ell \Vdash_I C_k} \uparrow L^\alpha$$

The use of the inductive hypothesis here also has as side conditions that $\mathsf{C} \in \sigma(\Psi_2)$ and that $\Psi_1', \Psi_2, (\uparrow^p_\ell B_\ell)^\alpha, B_\ell \geq m \geq k$. The former is an assumption, and the latter follows from $\ell \geq m$, which we learn from $\mathcal{D}$, and $\Psi_1, \Psi_2 \geq m \geq k$, which is another assumption of the theorem.

Alternately, if $\uparrow^p_\ell B_\ell$ was taken from Ψ_2, then $\Psi_1 = \Psi_1'$ and we also know that $\mathsf{C} \in \sigma(\ell)$.

We can construct the following (slightly different) proof:

$$\frac{\ell \geq k \quad \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi_1, \Psi_2', (\uparrow^p_\ell B_\ell)^\alpha, B_\ell \Vdash_I A_m\end{matrix} \quad \begin{matrix}\mathcal{E} \\ \Psi_2, \Psi_3, A_m \Vdash_I C_k\end{matrix}}{\Psi_1, \Psi_2', \Psi_3, \uparrow^p_\ell B_\ell, B_\ell \Vdash_I C_k} \text{ i.h.}(A_m, \mathcal{D}_1, \mathcal{E})}{\Psi_1, \Psi_2', \Psi_3, \uparrow^p_\ell B_\ell \Vdash_I C_k} \uparrow L^1$$

Note that here, since $\mathcal{E}$ will need to use $\uparrow^p_\ell B_\ell$, we apply $\uparrow L^1$ regardless of the value of α. We then propagate this formula to $\mathcal{E}$ unconditionally, and to $\mathcal{D}_1$ only if $\alpha = 1$, and so it is actually needed in $\mathcal{D}_1$.

This type of careful management of contraction to ensure that formulae occur exactly in the branches of proofs where they are needed is characteristic of ADJ^I. Since contraction does not always come with weakening, if we are not careful in this way, we run the risk of inadvertently contracting too aggressively, and ending up with excess copies of a proposition that we are unable to use. Other commuting cases require similar care, but are structurally much the same as this one.

Principal Cases As for ADJ^E, we examine both shift cases, with others following similarly.

First, we consider $A_m = {\downarrow}^{\ell}_{m} A_\ell$, with

$$\mathcal{D} = \frac{\Psi_a \geq \ell \quad \mathsf{W} \in \sigma(\Psi_b) \quad \overset{\mathcal{D}_1}{\Psi_a \Vdash_I A_\ell}}{\Psi_a, \Psi_b \Vdash_I {\downarrow}^{\ell}_{m} A_\ell} {\downarrow}R \qquad \text{and} \qquad \mathcal{E} = \frac{\overset{\mathcal{E}_1}{\Psi_2, \Psi_3, ({\downarrow}^{\ell}_{m} A_\ell)^\alpha, A_\ell \Vdash_I C_k}}{\Psi_2, \Psi_3, {\downarrow}^{\ell}_{m} A_\ell \Vdash_I C_k} {\downarrow}L^\alpha$$

Here, $\Psi_1, \Psi_2 = \Psi_a, \Psi_b$, but the split into Ψ_1 and Ψ_2 may be different from the split into Ψ_a and Ψ_b.

If $\alpha = 1$, we can perform a cross-cut, applying the inductive hypothesis to $({\downarrow}^{\ell}_{m} A_\ell, \mathcal{D}, \mathcal{E}_1)$, giving a proof $\mathcal{E}_1'$ of $\Psi_1, \Psi_2, \Psi_3, A_\ell \Vdash_I C_k$.

The inductive hypothesis can then be applied to $(A_\ell, \mathcal{D}_1, \mathcal{E}_1')$ to get a proof witnessing that $\Psi_a, \Psi_1, \Psi_2, \Psi_3 \Vdash_I C_k$. Applying admissibility of contraction to remove the extra copy of Ψ_a then gives the desired result. This is possible because we know that $\Psi_a \geq \ell$, and since $\alpha = 1$, it must be the case that $\mathsf{C} \in \sigma(\ell)$.

Now, if $\alpha = 0$, we avoid the need for this cross-cut, and can instead directly apply the inductive hypothesis to $(A_\ell, \mathcal{D}_1, \mathcal{E}_1)$ to get a proof of $\Psi_a, \Psi_2, \Psi_3 \Vdash_I C_k$. We may then apply admissibility of weakening to Ψ_b, all of which is weakenable, to get $\Psi_a, \Psi_b, \Psi_2, \Psi_3 \Vdash_I C_k$. Since $\Psi_a, \Psi_b = \Psi_1, \Psi_2$, this is equivalent to $\Psi_1, \Psi_2, \Psi_2, \Psi_3 \Vdash_I C_k$, and admissibility of contraction applied to Ψ_2 then gives the desired result.

Next, consider the case of $A_m = {\uparrow}^{m}_{\ell} A_\ell$, with

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Psi_1, \Psi_2 \Vdash_I A_\ell}}{\Psi_1, \Psi_2 \Vdash_I {\uparrow}^{m}_{\ell} A_\ell} {\uparrow}R \qquad \text{and} \qquad \mathcal{E} = \frac{\ell \geq k \quad \overset{\mathcal{E}_1}{\Psi_2, \Psi_3, ({\uparrow}^{m}_{\ell} A_\ell)^\alpha, A_\ell \Vdash_I C_k}}{\Psi_2, \Psi_3, {\uparrow}^{m}_{\ell} A_\ell \Vdash_I C_k} {\uparrow}L^\alpha$$

If $\alpha = 1$, we construct the following proof:

$$\cfrac{\overset{\mathcal{D}_1}{\Psi_1, \Psi_2 \Vdash_I A_\ell} \qquad \cfrac{\overset{\mathcal{D}}{\Psi_1, \Psi_2 \Vdash_I {\uparrow}^{m}_{\ell} A_\ell} \quad \overset{\mathcal{E}_1}{\Psi_2, \Psi_3, {\uparrow}^{m}_{\ell} A_\ell, A_\ell \Vdash_I C_k}}{\Psi_1, \Psi_2, \Psi_3, A_\ell \Vdash_I C_k} \text{ i.h.}({\uparrow}^{m}_{\ell} A_\ell, \mathcal{D}, \mathcal{E}_1)}{\cfrac{\Psi_1, \Psi_1, \Psi_2, \Psi_3 \Vdash_I C_k}{\Psi_1, \Psi_2, \Psi_3 \Vdash_I C_k} \ \vdots\ \mathsf{contract}^*} \text{ i.h.}(A_\ell, \mathcal{D}_1, \ldots)$$

This relies on the fact that if $\alpha = 1$, $\mathsf{C} \in \sigma(\ell)$, so since $\Psi_1, \Psi_2 \geq \ell$ by assumption, both Ψ_1 and Ψ_2 are contractible. For both applications of the inductive hypothesis, the side conditions $\Psi_1, \Psi_2 \geq \ell \geq k$ and $\mathsf{C} \in \sigma(\Psi_2)$ are needed, but these both follow from the assumptions of the theorem, $\ell \geq k$ (from $\mathcal{E}$), and $m \geq \ell$ (from well-formedness of ${\uparrow}^{m}_{\ell} A_\ell$).

Finally, if $\alpha = 0$, we construct the following (simpler) proof:

$$\frac{\Psi_1, \Psi_2 \geq \ell \geq k \quad \mathsf{C} \in \sigma(\Psi_2) \quad \overset{\mathcal{D}_1}{\Psi_1, \Psi_1 \Vdash_I A_\ell} \quad \overset{\mathcal{E}_1}{\Psi_2, \Psi_3, A_\ell \Vdash_I C_k}}{\Psi_1, \Psi_2, \Psi_3 \Vdash_I C_k} \text{ i.h.}(A_\ell, \mathcal{D}_1, \mathcal{E}_1)$$

Again, the side conditions on this use of the induction hypothesis follow from assumptions of the theorem, $\ell \geq k$, and $m \geq k$.

□

As in the case of ADJ^E, cut elimination follows immediately from an induction on the (potentially cut-containing) proof that $\Psi \vdash_I A_m$, using admissibility of cut to remove each cut as it is reached.

Theorem 8 (Cut elimination for ADJ^I)**.** *If* $\Psi \vdash_I A_m$*, then* $\Psi \Vdash_I A_m$.

2.3 ADJ^F: Focused Adjoint Logic

In ADJ^I, we were able to reduce the non-determinism involved in proof search by limiting structural rules to where they are essential, as in the dyadic system Σ_2 of Andreoli [2]. In addition to this source of non-determinism, Andreoli also identifies that certain pairs of inference rules can be permuted past each other without affecting the overall structure of the proof. For example, the following two proof fragments[6] have the same premises and conclusion, and so may be freely substituted for one another, but differ only in the order that we have chosen to apply the rules:

$$\dfrac{\dfrac{A_m, B_m, C_m \vdash D_m}{A_m, B_m, {\downarrow}^m_k C_m \vdash D_m}\,{\downarrow}L^0}{A_m \otimes B_m, {\downarrow}^m_k C_m \vdash D_m}\,{\otimes}L^0 \qquad \text{and} \qquad \dfrac{\dfrac{A_m, B_m, C_m \vdash D_m}{A_m \otimes B_m, C_m \vdash D_m}\,{\otimes}L^0}{A_m \otimes B_m, {\downarrow}^m_k C_m \vdash D_m}\,{\downarrow}L^0$$

In order to eliminate this kind of "don't-care non-determinism", where rules may be applied in either order, Andreoli splits connectives (and thus formulae, by their top-level connective) into "synchronous" and "asynchronous", where asynchronous formulae may be eagerly decomposed without fear of missing a proof. His final, triadic system Σ_3 then proceeds in alternating phases. In the asynchronous phase, all asynchronous formulae are decomposed as far as possible, stopping once a synchronous connective is reached (here, the order does not matter, so it may be treated as if all such formulae are decomposed in parallel, or some ordering may be enforced to make the process truly deterministic). In the synchronous phase, one (synchronous) formula in particular is chosen non-deterministically to "focus" on, and it and its subformulae are then decomposed as far as possible, until all remaining such subformulae are asynchronous, at which point the next asynchronous phase begins. The somewhat remarkable result of [2] is that this structured approach to proof is sound and complete with respect to the much more non-deterministic calculus Σ_1 for linear logic. In Σ_3, the only non-determinism lies in the choice of which synchronous formula to focus on in each synchronous phase.

Since Andreoli's work is in the context of classical linear logic, our approach will differ slightly — while asynchronous formulae may be freely broken down *on the right of a sequent*, on the left, this is reversed, with synchronous formulae being freely broken down, and asynchronous formulae needing to be focused on. Howe [45], in his treatment of focusing for intuitionistic linear logic, labels connectives positive or negative based on whether they occur on the left or on the right of the sequent, and modifies his use of the terms synchronous and asynchronous accordingly so that, for instance, $\otimes^-$, occurring on the right, is asynchronous, but $\otimes^+$, occurring on the left of the sequent, is not. We find this use of terminology confusing, and so will prefer to talk about the *polarity* of connectives, as used by Liang and Miller [56, 57], for instance, both to classify atomic propositions in a similar manner to Andreoli's synchrony, as well as

[6] Note that we write $\vdash_I$ here to emphasize that this is a proof in ADJ^I

to clarify the distinction between the conjunctions $\otimes$ and $\&$, which, in structural intuitionistic logic, both reduce to $\wedge$. This distinction by polarity is also similar to the separation in call-by-push-value [54, 55] between *value types* and *computation types*, with value types corresponding closely to positive formulae, and computation types to negative formulae.

We now begin to set up the calculus ADJ^F as a focused calculus for adjoint logic, following the ideas of Andreoli, although we will use the more recent approach of Simmons [98], using cut elimination and identity expansion for the focused calculus in order to prove focalization (completeness).

We begin by assigning polarities (following prior systems in that positive propositions should be those which are "asynchronous on the right", to use Andreoli's terminology) to the propositions of ADJ, giving us the following syntax:

$$\begin{array}{llcl}
\textit{Negative propositions} & A_m^-, B_m^- & ::= & p_m^- \mid A_m^+ \multimap B_m^- \mid \&_{j\in J} A_m^{j\,-} \mid {\uparrow}_k^m A_k^+ \\
\textit{Positive propositions} & A_m^+, B_m^+ & ::= & p_m^+ \mid A_m^+ \otimes B_m^+ \mid \mathbf{1}_m^+ \mid \oplus_{j\in J} A_m^{j\,+} \mid {\downarrow}_m^\ell A_\ell^-
\end{array}$$

Here, p_m^+ and p_m^- are positive and negative atomic propositions, respectively.

In this polarization, we have chosen to make both $\downarrow$ and $\uparrow$ shifts reverse polarity. We believe that other polarizations are possible, which may streamline some encodings, but as the polarity-reversing shifts are sufficient here, we leave the polarity-preserving shifts to potential future work.

Again closely following Simmons [98], we use the following grammar for the components of our sequents:

$$\begin{array}{llcl}
\textit{Stable antecedents} & \Psi & ::= & \cdot \mid A_m^- \mid \langle A_m^+ \rangle \mid \Psi, \Psi' \\
\textit{Inversion antecedents} & \Omega & ::= & \cdot \mid A_m^+ \bullet \Omega \\
\textit{Succedents} & U_m & ::= & [A_m^+] \mid A_m^+ \mid A_m^- \mid \langle A_m^- \rangle \\
\textit{Ordered antecedents} & L & ::= & \Omega \mid [A_m^-]
\end{array}$$

Note that despite our reuse of Ψ, in the context of ADJ^F, this refers specifically to a *stable* antecedent, following the grammar above, with all positive propositions being suspended (of the form $\langle A_m^+ \rangle$). We use a large centered dot $\bullet$ rather than a comma to separate propositions in the (ordered) inversion context Ω to emphasize that those contexts are to be treated as lists rather than as multisets.

Just as we distinguish stable and inversion antecedents, we will refer to the succedents A_m^+ and $\langle A_m^- \rangle$ as stable succedents.

We use square brackets to denote propositions in focus (e.g. $[A_m^+], [A_m^-]$). Likewise, we use angle brackets to denote suspended propositions ($\langle A_m^+ \rangle, \langle A_m^- \rangle$). As atomic propositions are intended to represent arbitrary formulae, we cannot break them down in the inversion phase (as doing so would require knowing more than just their polarity). Instead, in our focused system (Figure 2.3), we suspend atomic propositions, making it possible for them to appear in stable sequents as antecedents with otherwise positive propositions and in succedents with otherwise negative propositions, respectively. The use of arbitrary suspended propositions is a technical device introduced by Simmons [98] that allows for a structural proof of identity expansion for the focused system, and so we allow for such non-atomic suspended propositions in sequents, although our rules provide no way to introduce such a suspension, as the $\mathsf{susp}^{\pm}$ rules only allow for suspending atomic propositions. Moreover, allowing our sequents to contain arbitrary suspended propositions means that we can substitute an arbitrary proposition $A_m^{\pm}$ for an atomic proposition $p_m^{\pm}$ while staying within the confines of the system.

Using these parts, we have the following four types of sequents:

Right focus	$\Psi \vdash_F [A_m^+]$
Right inversion	$\Psi \; ; \; \Omega \vdash_F A_m^{\pm}$
Left inversion	$\Psi \; ; \; \Omega \vdash_F \langle A_m^- \rangle$
Left focus	$\Psi \; ; \; [A_m^-] \vdash_F U_\ell$ (where U_ℓ is *stable*)

Each of these sequents is a special case of the general form $\Psi \; ; \; L \vdash_F U$, but it is useful to separate these cases for some theorem statements and proofs. We will also often combine the two inversion cases into a single $\Psi \; ; \; \Omega \vdash_F U_m$, where U_m is not $[A_m^+]$, as they can generally be treated together in proofs. The constraints on what form U_m may take in each sequent are standard for intuitionistic focused systems [57, 98], and serve to ensure that at most one formula is in focus at a time, and that if a formula is in focus, then there are no formulae in inversion. We also note that it is sometimes useful to distinguish *stable sequents*, which are those where both the antecedents and succedent are entirely stable — that is, sequents of the form $\Psi \; ; \; \cdot \vdash_F U_m$, with U_m stable. We will see when we come to the rules for ADJ^F that these stable sequents are exactly those which allow us to make a choice of how to proceed by focusing on a proposition, while if we have a proposition in focus or in inversion, we are restricted to have only one applicable rule at a time.

Most of the rules of ADJ^F (Figure 2.3) arise straightfowardly from their ADJ^I counterparts by having the principal formula either in focus or in inversion (depending on its polarity and whether it is on the left or the right — we focus on positive formulae on the right and negative formulae on the left, and likewise, we invert negative formulae on the right, and positive formulae on the left). We also add the $\mathsf{focus}^{\pm}$ rules, which allow us to bring formulae into focus. The $\mathsf{susp}^{\pm}$ rules likewise allow us to suspend atomic formulae that we can no longer break down. Finally, the $\mathsf{id}^{\pm}$ rules bring suspended and focused formulae together, using one to prove the other.

2.3.1 Soundness and Completeness

As before, this calculus is sound and complete with respect to the prior two. In particular, it is easiest to show that it is sound and complete with respect to ADJ^I. In order to state the soundness and completeness theorems (or defocalization and focalization), we must first give a concept of erasure. Informally, we think of $(\cdot)^{\bullet}$ as removing all focusing and suspension brackets, as well as all polarity information from the proposition or context being erased. We can then apply this erasure to each portion of a sequent to convert from an ADJ^F sequent to a corresponding ADJ^E or ADJ^I sequent (as both systems use the same syntax of sequents, other than the labelling of the turnstile). Unlike in many prior focused systems, where new connectives are added to change polarity (without affecting provability), we have reused the existing shift connectives to change polarities. As such, when we are presented with a polarized formula, we cannot just remove all shifts and hope to be left with a sensible unpolarized formula in the end. Instead, we observe that

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi, \langle A_m^+ \rangle \vdash [A_m^+]}\ \mathsf{id}^+$$

$$\frac{\Psi_1 \geq m \quad \mathsf{W} \in \sigma(\Psi_2) \quad \Psi_1 \,;\, \cdot \vdash A_m^-}{\Psi_1, \Psi_2 \vdash [{\downarrow}_k^m A_m^-]}\ {\downarrow}R \qquad \frac{i \in J \quad \Psi \vdash \left[A_m^{i\ +}\right]}{\Psi \vdash \left[\oplus_{j \in J} A_m^{j\ +}\right]}\ \oplus R^i$$

$$\frac{\mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash [A_m^+] \quad \Psi_2, \Psi_3 \vdash [B_m^+]}{\Psi_1, \Psi_2, \Psi_3 \vdash [A_m^+ \otimes B_m^+]}\ \otimes R \qquad \frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \vdash [\mathbf{1}_m^+]}\ \mathbf{1}R$$

$$\frac{\Psi \vdash [A_m^+]}{\Psi \,;\, \cdot \vdash A_m^+}\ \mathsf{focus}^+ \qquad \frac{U_\ell\,\text{stable} \quad \Psi, (A_m^-)^\alpha \,;\, [A_m^-] \vdash U_\ell}{\Psi, A_m^- \,;\, \cdot \vdash U_\ell}\ (\mathsf{focus}^-)^\alpha$$

$$\frac{\Psi, \langle p^+ \rangle \,;\, \Omega \vdash U_\ell}{\Psi \,;\, p^+ \bullet \Omega \vdash U_\ell}\ \mathsf{susp}^+ \qquad \frac{\Psi \,;\, \cdot \vdash \langle p^- \rangle}{\Psi \,;\, \cdot \vdash p^-}\ \mathsf{susp}^-$$

$$\frac{\Psi, A_m^- \,;\, \Omega \vdash U_\ell}{\Psi \,;\, {\downarrow}_k^m A_m^- \bullet \Omega \vdash U_\ell}\ {\downarrow}L \qquad \frac{\Psi \,;\, A_m^{j\ +} \bullet \Omega \vdash U_\ell \quad \text{for all } j \in J}{\Psi \,;\, \oplus_{j \in J} A_m^{j\ +} \bullet \Omega \vdash U_\ell}\ \oplus L$$

$$\frac{\Psi \,;\, A_m^+ \bullet B_m^+ \bullet \Omega \vdash U_\ell}{\Psi \,;\, A_m^+ \otimes B_m^+ \bullet \Omega \vdash U_\ell}\ \otimes L \qquad \frac{\Psi \,;\, \Omega \vdash U_\ell}{\Psi \,;\, \mathbf{1}_m^+ \bullet \Omega \vdash U_\ell}\ \mathbf{1}L$$

$$\frac{\Psi \,;\, \cdot \vdash A_k^+}{\Psi \,;\, \cdot \vdash {\uparrow}_k^m A_k^+}\ {\uparrow}R \qquad \frac{\Psi \,;\, \cdot \vdash A_m^{j\ -} \quad \text{for all } j \in J}{\Psi \,;\, \cdot \vdash \&_{j \in J} A_m^{j\ -}}\ \&R$$

$$\frac{\Psi \,;\, A_m^+ \vdash B_m^-}{\Psi \,;\, \cdot \vdash A_m^+ \multimap B_m^-}\ \multimap R$$

$$\frac{\mathsf{W} \in \sigma(\Psi)}{\Psi \,;\, [A_m^-] \vdash \langle A_m^- \rangle}\ \mathsf{id}^-$$

$$\frac{k \geq \ell \quad \Psi \,;\, A_k^+ \vdash U_\ell}{\Psi \,;\, [{\uparrow}_k^m A_k^+] \vdash U_\ell}\ {\uparrow}L \qquad \frac{i \in J \quad \Psi \,;\, \left[A_m^{i\ -}\right] \vdash U_\ell}{\Psi \,;\, \left[\&_{j \in J} A_m^{j\ -}\right] \vdash U_\ell}\ \&L^i$$

$$\frac{\Psi_1, \Psi_2 \geq m \quad \mathsf{C} \in \sigma(\Psi_2) \quad \Psi_1, \Psi_2 \vdash [A_m^+] \quad \Psi_2, \Psi_3 \,;\, [B_m^-] \vdash U_\ell}{\Psi_1, \Psi_2, \Psi_3 \,;\, [A_m^+ \multimap B_m^-] \vdash U_\ell}\ \multimap L$$

Figure 2.3: Focused Adjoint Logic (ADJ^F).
"U stable" means that U is either A_m^+ or $\langle p_m^- \rangle$.
Note also that each $\vdash$ in this figure should formally be $\vdash_F$, but we omit the subscript for simplicity of presentation.

shifts from a mode to itself (i.e. ${\downarrow}_m^m$ and ${\uparrow}_m^m$) do not affect provability, while shifts that do change mode may. As such, we *allow* our erasure process to remove these shifts, but do not require it to do so, giving us an erasure *relation*, rather than function. One polarized formula may be a valid polarization of several different unpolarized formulae, just as an unpolarized formula may have several different polarizations.

Definition 2 (Focusing Erasure)**.** *For single propositions, we have a straightforward inductive (but non-deterministic, thus defining a relation rather than a function) definition of erasure, removing polarity information throughout the proposition:*

$$\begin{array}{rcl@{\qquad\qquad}rcl}
(p_m^\pm)^\bullet & ::= & p_m & (\mathbf{1}_m^+)^\bullet & ::= & \mathbf{1}_m \\
(A_m^+ \otimes B_m^+)^\bullet & ::= & (A_m^+)^\bullet \otimes (B_m^+)^\bullet & (A_m^+ \multimap B_m^-)^\bullet & ::= & (A_m^+)^\bullet \multimap (B_m^-)^\bullet \\
(\oplus_{j\in J} A_m^{j\,+})^\bullet & ::= & \oplus_{j\in J}(A_m^{j\,+})^\bullet & (\&_{j\in J} A_m^{j\,-})^\bullet & ::= & \&_{j\in J}(A_m^{j\,-})^\bullet \\
({\downarrow}_m^\ell A_\ell^-)^\bullet & ::= & {\downarrow}_m^\ell (A_\ell^-)^\bullet & ({\uparrow}_k^m A_k^+)^\bullet & ::= & {\uparrow}_k^m (A_k^+)^\bullet \\
({\downarrow}_m^m A_m^-)^\bullet & ::= & (A_m^-)^\bullet & ({\uparrow}_m^m A_m^+)^\bullet & ::= & (A_m^+)^\bullet
\end{array}$$

Note that the last four rules overlap, but that they are the only such overlapping rules.

We can then extend this to propositions which are suspended or in focus, stripping away the suspension or focusing brackets:

$$([A_m^\pm])^\bullet \ ::= \ (A_m^\pm)^\bullet \qquad\qquad (\langle A_m^\pm \rangle)^\bullet \ ::= \ (A_m^\pm)^\bullet$$

At this point, we know how to erase all succedents U, but still need to extend erasure to contexts Ψ and Ω, which we do pointwise:

$$\begin{array}{rcl}
(\cdot)^\bullet & ::= & \cdot \\
(\Psi, \Psi')^\bullet & ::= & (\Psi)^\bullet, (\Psi')^\bullet \\
(A_m^+ \bullet \Omega)^\bullet & ::= & (A_m^+)^\bullet, (\Omega)^\bullet
\end{array}$$

With this definition of erasure potentially yielding multiple results, we would like to ensure that it interacts sensibly with provability — in particular, two erasures of the same polarized sequent should be equiprovable (in ADJ^I). Since we have admissibility of cut in ADJ^I, it will suffice to show that if $(A_m^\pm)^\bullet$ yields both B_m and C_m, then $B_m \vdash_I C_m$ and $C_m \vdash_I B_m$.

Lemma 2 (Equiprovability of erasures)**.** *Suppose $A_m^\pm, B_m, C_m$ are such that $(A_m^\pm)^\bullet$ yields both B_m and C_m. Then, $B_m \vdash_I C_m$ and $C_m \vdash_I B_m$.*

Proof. We proceed by induction on the structure of $A_m^\pm$. In all cases but shifts from a mode m to itself, there is only one possible erasure, and so B_m and C_m have the same top-level connective. We can then (using the same approach as for proving identity expansion) apply the left and right rule for this connective (first left, then right for positive connectives, and vice versa for negative connectives), and apply the inductive hypothesis.

If $A_m^\pm = {\downarrow}_m^m D_m^-$, we note that if $(D_m^-)^\bullet$ yields both B_m and C_m, or if ${\downarrow}_m^m (D_m^-)^\bullet$ yields both B_m and C_m, we can follow the same approach as for other connectives. We therefore assume without loss of generality that $B_m = (D_m^-)^\bullet$ while $C_m = {\downarrow}_m^m (D_m^-)^\bullet$, and construct the following two proofs:

$$\dfrac{(D_m^-)^\bullet \geq m \qquad \dfrac{\cdots}{(D_m^-)^\bullet \vdash_I (D_m^-)^\bullet}\,\text{i.h.}(D_m^-)}{(D_m^-)^\bullet \vdash_I {\downarrow}_m^m (D_m^-)^\bullet}\,{\downarrow}R \qquad\qquad \dfrac{\dfrac{\cdots}{(D_m^-)^\bullet \vdash_I (D_m^-)^\bullet}\,\text{i.h.}(D_m^-)}{{\downarrow}_m^m (D_m^-)^\bullet \vdash_I (D_m^-)^\bullet}\,{\downarrow}L^0$$

The cases where $A_m^{\pm} = \uparrow_m^m D_m^+$ are similar. □

With this erasure operation, we are now equipped to give our defocalization (or soundness) theorem. Since all erasures of a given polarized sequent are equiprovable, we do not need to be precise about which one we refer to in the theorem statement.

Theorem 9 (Defocalization). *If* $\Psi \; ; \; L \vdash_F U_m$*, then* $(\Psi)^\bullet, (L)^\bullet \vdash_I (U_m)^\bullet$.

Proof. We prove this by noting that each (erased) rule of the focused system is either a rule of ADJ^I or (in the case of the $\mathsf{focus}^{\pm}$ and $\mathsf{susp}^{\pm}$ rules) a no-op. As such, we translate the ADJ^F proof into an ADJ^I proof rule-by-rule, removing the no-op rules. □

Focalization (or completeness of ADJ^F with respect to ADJ^I) is much more involved, and we begin by proving versions of the cut admissibility and identity expansion theorems for ADJ^F. One technical condition that we will make use of in cut admissibility is that the only suspended propositions in a given sequent are atomic. We refer to such a sequent as *suspension-normal*, following Simmons' terminology, and we will also refer to propositions and contexts as suspension-normal if likewise, all suspended propositions they contain are atomic. Recall that while our syntax of propositions allow for arbitrary suspended propositions, and indeed our $\mathsf{id}^{\pm}$ rules allow us to *use* these arbitrary suspended propositions in proofs, the $\mathsf{susp}^{\pm}$ rules only let us suspend atomic propositions. As such, if we begin with a suspension-normal sequent, any sequence of rules we apply (in a bottom-up direction) will leave us with another suspension-normal sequent.

Theorem 10 (Cut admissibility for ADJ^F). *Assuming* $\mathsf{C} \in \sigma(\Psi_2)$*,* $\Psi_1, \Psi_2 \geq m \geq \ell$*, and that* Ψ_1*,* Ψ_2*,* Ψ_3*, and* U_ℓ *are suspension-normal:*

1. *If* $\Psi_1, \Psi_2 \vdash_F [A_m^+]$ *and* $\Psi_2, \Psi_3 \; ; \; A_m^+ \bullet \Omega \vdash_F U_\ell$*, then* $\Psi_1, \Psi_2, \Psi_3 \; ; \; \Omega \vdash_F U_\ell$.
2. *If* $\Psi_1, \Psi_2 \; ; \; \cdot \vdash_F A_m^-$ *and* $\Psi_2, \Psi_3 \; ; \; [A_m^-] \vdash_F U_\ell$ *and* U_ℓ *stable, then* $\Psi_1, \Psi_2, \Psi_3 \; ; \; \cdot \vdash_F U_\ell$.
3. *If* $\Psi_1, \Psi_2 \; ; \; L \vdash_F A_m^+$ *and* $\Psi_2, \Psi_3 \; ; \; A_m^+ \vdash_F U_\ell$ *and* U_ℓ *stable, then* $\Psi_1, \Psi_2, \Psi_3 \; ; \; L \vdash_F U_\ell$.
4. *If* $\Psi_1, \Psi_2 \; ; \; \cdot \vdash_F A_m^-$ *and* $\Psi_2, \Psi_3, A_m^- \; ; \; L \vdash_F U_\ell$*, then* $\Psi_1, \Psi_2, \Psi_3 \; ; \; L \vdash_F U_\ell$.

Proof. This proceeds in a relatively standard nested induction, except that cases (3) and (4) depend on cases (1) and (2), respectively. As such, we prove this by induction over the (lexicographically ordered) quadruple $(A_m^{\pm}, i, \mathcal{D}, \mathcal{E})$, where $A_m^{\pm}$ is the formula cut out, i is the case number in the theorem statement, $\mathcal{D}$ is the left-hand proof of the cut, and $\mathcal{E}$ is the right-hand proof of the cut.

This proof relies critically on admissibility of weakening and contraction (where permitted by the mode) in ADJ^F, both of which follow from standard structural inductions on the proofs in question. □

Theorem 11 (Admissibility of Identity for ADJ^F). *If* $\mathsf{W} \in \sigma(\Psi)$*, then:*

1. *For any* A_m^+*,* $\Psi \; ; \; A_m^+ \vdash_F A_m^+$.
2. *For any* A_m^-*,* $\Psi, A_m^- \; ; \; \cdot \vdash_F A_m^-$.

Proof. This result proceeds by first showing that suspension of arbitrary propositions (rather than only atomic propositions) is admissible, again via a standard induction, here over the structure of $A_m^{\pm}$. We write $\eta^{\pm}$ for these admissible general suspension rules where we use them.

Once we are able to suspend arbitrary propositions, combining this with the $\mathsf{focus}^{\pm}$ rules and the $\mathsf{id}^{\pm}$ rules gives the desired result immediately. □

One problem with erasure is that, in general, many propositions have the same erasures. With the aid of cut and identity, we can simplify this somewhat, by proving the following two lemmas, which allow us to remove double shifts. Of course, not all double shifts can be removed while preserving erasure, if those shifts appear in the erased proposition as well, but we can, in a sense, turn an $A_m^\pm$ which erases to C_m into a form with as few extraneous shifts as possible via these lemmas.

Lemma 3 (Double shift removal (positive))**.** *Suppose we have some A_m^+. Then, there exists B_m^+ not of the form $\downarrow_m^m \uparrow_m^m C_m^+$ such that $(B_m^+)^\bullet = (A_m^+)^\bullet$ (in the sense that they share at least one possible erasure) and for all stable antecedents Ψ, $\Psi \;;\, \cdot \vdash_F B_m^+$ iff $\Psi \;;\, \cdot \vdash_F A_m^+$.*

Proof. By induction on the structure of A_m^+. If A_m^+ does not have the form $\downarrow_m^m \uparrow_m^m C_m^+$, we may take $B_m^+ = A_m^+$ and are done.

Suppose therefore that $A_m^+ = \downarrow_m^m \uparrow_m^m C_m^+$. Applying the inductive hypothesis to C_m^+, we get B_m^+ such that $\Psi \;;\, \cdot \vdash_F B_m^+$ iff $\Psi \;;\, \cdot \vdash_F C_m^+$. It therefore remains to show that $\Psi \;;\, \cdot \vdash_F C_m^+$ iff $\Psi \;;\, \cdot \vdash_F \downarrow_m^m \uparrow_m^m C_m^+$. We construct the following two proofs as witnesses:

$$\dfrac{\dfrac{\dfrac{\Psi \;;\, \cdot \vdash_F C_m^+}{\Psi \;;\, \cdot \vdash_F \uparrow_m^m C_m^+}\,\uparrow R}{\Psi \vdash_F [\downarrow_m^m \uparrow_m^m C_m^+]}\,\downarrow R}{\Psi \;;\, \cdot \vdash_F \downarrow_m^m \uparrow_m^m C_m^+}\,\mathsf{focus}^+ \qquad \text{and} \qquad \dfrac{\Psi \;;\, \cdot \vdash_F \downarrow_m^m \uparrow_m^m C_m^+ \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle C_m^+\rangle \vdash_F [C_m^+]}\,\mathsf{id}^+}{\langle C_m^+\rangle \;;\, \cdot \vdash_F C_m^+}\,\mathsf{focus}^+}{\cdot \;;\, C_m^+ \vdash_F C_m^+}\,\eta^+}{\cdot \;;\, [\uparrow_m^m C_m^+] \vdash_F C_m^+}\,\uparrow L}{\uparrow_m^m C_m^+ \;;\, \cdot \vdash_F C_m^+}\,(\mathsf{focus}^-)^0}{\cdot \;;\, \downarrow_m^m \uparrow_m^m C_m^+ \vdash_F C_m^+}\,\downarrow L}{\Psi \;;\, \cdot \vdash_F C_m^+}\,\mathsf{cut}(3)$$

□

Lemma 4 (Double shift removal (negative))**.** *Given A_m^-, there exists B_m^- not of the form $\uparrow_m^m \downarrow_m^m C_m^-$ such that $(B_m^-)^\bullet = (A_m^-)^\bullet$ and for all stable Ψ and U_ℓ, $\Psi, B_m^- \;;\, \cdot \vdash_F U_\ell$ iff $\Psi, A_m^- \;;\, \cdot \vdash_F U_\ell$.*

Proof. By induction on the structure of A_m^-. If A_m^- does not have the form $\uparrow_m^m \downarrow_m^m C_m^-$, we may take $B_m^- = A_m^-$ and are done.

Suppose therefore that $A_m^- = \uparrow_m^m \downarrow_m^m C_m^-$. Applying the inductive hypothesis to C_m^-, we get B_m^- such that $\Psi, B_m^- \;;\, \cdot \vdash_F U_\ell$ iff $\Psi, C_m^- \;;\, \cdot \vdash_F U_\ell$. We therefore need only show that $\Psi, C_m^- \;;\, \cdot \vdash_F U_\ell$ iff $\Psi, \uparrow_m^m \downarrow_m^m C_m^- \;;\, \cdot \vdash_F U_\ell$, which is witnessed by the following two proofs:

$$\dfrac{\dfrac{\dfrac{\Psi, C_m^- \;;\, \cdot \vdash_F U_\ell}{\Psi, \downarrow_m^m C_m^- \;;\, \cdot \vdash_F U_\ell}\,\downarrow L}{\Psi \;;\, [\uparrow_m^m \downarrow_m^m C_m^-] \vdash_F U_\ell}\,\uparrow L}{\Psi, \uparrow_m^m \downarrow_m^m C_m^- \;;\, \cdot \vdash_F U_\ell}\,(\mathsf{focus}^-)^0$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\;;\;[C_m^-] \vdash_F \langle C_m^-\rangle}\;\mathsf{id}^-}{C_m^-\;;\;\cdot \vdash_F \langle C_m^-\rangle}\;(\mathsf{focus}^-)^0}{C_m^-\;;\;\cdot\vdash_F C_m^-}\;\eta^-}{C_m^-\;;\;\cdot\vdash_F [{\downarrow}^m_m C_m^-]}\;{\downarrow}R}{C_m^-\;;\;\cdot\vdash_F {\downarrow}^m_m C_m^-}\;\mathsf{focus}^+}{C_m^-\;;\;\cdot\vdash_F {\uparrow}^m_m{\downarrow}^m_m C_m^-}\;{\uparrow}R \qquad \Psi, {\uparrow}^m_m{\downarrow}^m_m C_m^-\;;\;\cdot\vdash_F U_\ell}{\Psi, C_m^-\;;\;\cdot\vdash_F U_\ell}\;\mathsf{cut}(4)$$

□

Admissibility of cut and identity also together allow us to prove admissible "unfocused" versions of the rules of ADJ^I within ADJ^F, which can then be used to reconstruct an ADJ^I proof as a whole in ADJ^F for our focalization result.

Theorem 12 (Focalization). *Suppose U_ℓ is stable and Ψ, U_ℓ are both suspension-normal. If $(\Psi)^\bullet \vdash_I (U_\ell)^\bullet$, then $\Psi\;;\;\cdot \vdash_F U_\ell$.*

Proof. We first note that cut elimination for ADJ^I allows us to only consider cut-free proofs of $(\Psi)^\bullet \vdash_I (U_\ell)^\bullet$, and likewise, identity expansion for ADJ^I allows us to assume that any applications of identity in this proof are at atomic propositions.

We then proceed by induction over this proof of $(\Psi)^\bullet \vdash_I (A_m)^\bullet$.

If the last rule used in this proof is an identity, we know that U_m must erase to an atom p_m, and so (since U_m is suspension-normal and stable), it must be either a suspended negative atom $\langle p_m^-\rangle$ or a proposition of the form ${\downarrow}^m_m{\uparrow}^m_m \ldots {\downarrow}^m_m{\uparrow}^m_m p_m^+$. Similarly, we know that either $\Psi = \Psi', {\uparrow}^m_m{\downarrow}^m_m \ldots {\uparrow}^m_m{\downarrow}^m_m p_m^-$ or $\Psi = \Psi', \langle p_m^+\rangle$, and that in either case, $\mathsf{W} \in \sigma(\Psi')$. Double shift removal allows us to assume that U_m is either $\langle p_m^-\rangle$ or p_m^+, without any shifts, and likewise allows us to simplify Ψ. With this simplification, we can construct one of the following two proofs, depending on the polarity of p_m:

$$\dfrac{\dfrac{\mathsf{W} \in \sigma(\Psi')}{\Psi', \langle p_m^+\rangle \vdash_F [p_m^+]}\;\mathsf{id}^+}{\Psi', \langle p_m^+\rangle\;;\;\cdot \vdash_F p_m^+}\;\mathsf{focus}^+ \qquad \text{or} \qquad \dfrac{\langle p_m^-\rangle \text{ stable} \qquad \dfrac{\mathsf{W} \in \sigma(\Psi')}{\Psi'\;;\;[p_m^-] \vdash_F \langle p_m^-\rangle}\;\mathsf{id}^-}{\Psi', p_m^-\;;\;\cdot \vdash_F \langle p_m^-\rangle}\;(\mathsf{focus}^-)^0$$

For each of the remaining rules, we will apply the inductive hypothesis to each premise of the last rule used, and then make use of admissibility of cut and general identity for ADJ^F (along with possibly weakening and contraction) to combine these premises with the focused equivalent of the ADJ^I rule that was used to produce the desired focused proof.

We show here a few sample cases, but note that all have similar constructions. In each case, we use a cut to replace the proposition being broken down with one that interrupts focusing at a suitable point (via additional shifts), which can then be broken down to yield a premise matching the result of the inductive hypothesis.

First, we examine the cases for $\otimes$, as a simple example that is not affected by modes.

If the last rule used in the ADJ^I proof was $\otimes L^0$, [7] then $(\Psi)^\bullet = (\Psi')^\bullet, (A_m^+)^\bullet \otimes (B_m^+)^\bullet$ for some Ψ', A_m^+, B_m^+. As such, $\Psi = \Psi', C_m^\pm$ for some $C_m^\pm$ which erases to $(A_m^+)^\bullet \otimes (B_m^+)^\bullet$. Since Ψ is suspension-normal and this $C_m^\pm$ cannot be atomic, $C_m^\pm$ must in fact be negative. Combining this with $(C_m^-)^\bullet = (A_m^+)^\bullet \otimes (B_m^+)^\bullet$, we conclude that $C_m^- = {\uparrow_m^m}{\downarrow_m^m} \ldots {\uparrow_m^m}(A_m^+ \otimes B_m^+)$. Double shift removal then allows us to assume that $C_m^- = {\uparrow_m^m}(A_m^+ \otimes B_m^+)$.

Now, we also note that the premise of this last rule is a proof that $(\Psi')^\bullet, (A_m^+)^\bullet, (B_m^+)^\bullet \vdash_I (U_\ell)^\bullet$. This is also a proof of $(\Psi')^\bullet, ({\uparrow_m^m} A_m^+)^\bullet, ({\uparrow_m^m} B_m^+)^\bullet \vdash_I (U_\ell)^\bullet$, and so we may apply the inductive hypothesis to get a proof of $\Psi', {\uparrow_m^m} A_m^+, {\uparrow_m^m} B_m^+ \;;\; \cdot \vdash_F U_\ell$.

We can then construct the following proof. Note that to reduce visual clutter, we write $\uparrow$ and $\downarrow$ to denote $\uparrow_m^m$ and $\downarrow_m^m$, respectively:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\langle A_m^+\rangle \vdash_F [A_m^+]}\,\mathsf{id}^+
}{\langle A_m^+\rangle \;;\; \cdot \vdash_F A_m^+}\,\mathsf{focus}^+
}{\langle A_m^+\rangle \;;\; \cdot \vdash_F {\uparrow} A_m^+}\,{\uparrow}R
}{\langle A_m^+\rangle \vdash_F [{\downarrow}{\uparrow} A_m^+]}\,{\downarrow}R
\quad
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\langle B_m^+\rangle \vdash_F [B_m^+]}\,\mathsf{id}^+
}{\langle B_m^+\rangle \;;\; \cdot \vdash_F B_m^+}\,\mathsf{focus}^+
}{\langle B_m^+\rangle \;;\; \cdot \vdash_F {\uparrow} B_m^+}\,{\uparrow}R
}{\langle B_m^+\rangle \vdash_F [{\downarrow}{\uparrow} B_m^+]}\,{\downarrow}R
}{\langle A_m^+\rangle, \langle B_m^+\rangle \vdash_F [({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)]}\,\otimes R
}{\langle A_m^+\rangle, \langle B_m^+\rangle \;;\; \cdot \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,\mathsf{focus}^+
}{\langle A_m^+\rangle \;;\; B_m^+ \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,\eta^+
}{\cdot \;;\; A_m^+ \bullet B_m^+ \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,\eta^+
}{\cdot \;;\; A_m^+ \otimes B_m^+ \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,\otimes L
}{\cdot \;;\; [{\uparrow}(A_m^+ \otimes B_m^+)] \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,{\uparrow}L
}{{\uparrow}(A_m^+ \otimes B_m^+) \;;\; \cdot \vdash_F ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+)}\,(\mathsf{focus}^-)^0
\quad
\dfrac{
\dfrac{
\dfrac{
\Psi', {\uparrow} A_m^+, {\uparrow} B_m^+ \;;\; \cdot \vdash_F U_\ell
}{\Psi', {\uparrow} A_m^+ \;;\; {\downarrow}{\uparrow} B_m^+ \vdash_F U_\ell}\,{\downarrow}L
}{\Psi' \;;\; {\downarrow}{\uparrow} A_m^+ \bullet {\downarrow}{\uparrow} B_m^+ \vdash_F U_\ell}\,{\downarrow}L
}{\Psi' \;;\; ({\downarrow}{\uparrow} A_m^+) \otimes ({\downarrow}{\uparrow} B_m^+) \vdash_F U_\ell}\,\otimes L
}{\Psi', {\uparrow}(A_m^+ \otimes B_m^+) \;;\; \cdot \vdash_F U_\ell}\,\mathsf{cut}(3)
$$

If the last rule used was $\otimes R$, then $(U_m)^\bullet = (A_m^+)^\bullet \otimes (B_m^+)^\bullet$ for some A_m^+, B_m^+. Since U_m is suspension-normal and stable, we can conclude that $U_m = {\downarrow_m^m}{\uparrow_m^m} \ldots {\downarrow_m^m}{\uparrow_m^m}(A_m^+ \otimes B_m^+)$, from which double shift removal allows us to assume that $U_m = A_m^+ \otimes B_m^+$.

We also note that the premises of this rule are proofs that $(\Psi_1, \Psi_2)^\bullet \vdash_I (A_m^+)^\bullet$ and $(\Psi_2, \Psi_3)^\bullet \vdash_I (B_m^+)^\bullet$, where $\Psi = \Psi_1, \Psi_2, \Psi_3$ and $\mathsf{C} \in \sigma(\Psi_2)$. Applying the inductive hypothesis to both yields proofs of $\Psi_1, \Psi_2 \;;\; \cdot \vdash_F A_m^+$ and $\Psi_2, \Psi_3 \;;\; \cdot \vdash_F B_m^+$. These can then be combined to give the

[7] The case for $\otimes L^1$ is similar, with the only difference being that in our cut, we propagate ${\uparrow}(A_m^+ \otimes B_m^+)$ to both sides of the cut, and adjust our application of the inductive hypothesis. In general, to reduce the size of proofs, we will only look at the L^0 rule variants in what is shown here, but the L^1 variants all follow similarly from their L^0 versions, either propagating a proposition to both sides of a multi-premise rule, or keeping a copy on the left when applying the $(\mathsf{focus}^-)^\alpha$ rule.

following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Psi_1, \Psi_2 \;;\; \cdot \vdash_F A^+_m}{\Psi_1, \Psi_2 \;;\; \cdot \vdash_F {\uparrow} A^+_m}\,{\uparrow}R}{\Psi_1, \Psi_2 \vdash_F [{\downarrow}{\uparrow} A^+_m]}\,{\downarrow}R \quad \dfrac{\dfrac{\Psi_1, \Psi_2 \;;\; \cdot \vdash_F B^+_m}{\Psi_1, \Psi_2 \;;\; \cdot \vdash_F {\uparrow} B^+_m}\,{\uparrow}R}{\Psi_1, \Psi_2 \vdash_F [{\downarrow}{\uparrow} B^+_m]}\,{\downarrow}R}{\dfrac{\Psi_1, \Psi_2, \Psi_3 \vdash_F [({\downarrow}{\uparrow} A^+_m) \otimes ({\downarrow}{\uparrow} B^+_m)]}{\Psi_1, \Psi_2, \Psi_3 \;;\; \cdot \vdash_F ({\downarrow}{\uparrow} A^+_m) \otimes ({\downarrow}{\uparrow} B^+_m)}\,\mathsf{focus}^+}\,\otimes R \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle A^+_m \rangle \vdash_F [A^+_m]}\,\mathsf{id}^+ \quad \dfrac{}{\langle B^+_m \rangle \vdash_F [B^+_m]}\,\mathsf{id}^+}{\langle A^+_m \rangle, \langle B^+_m \rangle \vdash_F [A^+_m \otimes B^+_m]}\,\otimes R}{\langle A^+_m \rangle, \langle B^+_m \rangle \;;\; \cdot \vdash_F A^+_m \otimes B^+_m}\,\mathsf{focus}^+}{\langle A^+_m \rangle \;;\; B^+_m \vdash_F A^+_m \otimes B^+_m}\,\eta^+}{\langle A^+_m \rangle \;;\; [{\uparrow} B^+_m] \vdash_F A^+_m \otimes B^+_m}\,{\uparrow}L}{\langle A^+_m \rangle, {\uparrow} B^+_m \;;\; \cdot \vdash_F A^+_m \otimes B^+_m}\,(\mathsf{focus}^-)^0}{{\uparrow} B^+_m \;;\; A^+_m \vdash_F A^+_m \otimes B^+_m}\,\eta^+}{{\uparrow} B^+_m \;;\; [{\uparrow} A^+_m] \vdash_F A^+_m \otimes B^+_m}\,{\uparrow}L}{{\uparrow} A^+_m, {\uparrow} B^+_m \;;\; \cdot \vdash_F A^+_m \otimes B^+_m}\,(\mathsf{focus}^-)^0}{{\uparrow} A^+_m \;;\; {\downarrow}{\uparrow} B^+_m \vdash_F A^+_m \otimes B^+_m}\,{\downarrow}L}{\cdot \;;\; {\downarrow}{\uparrow} A^+_m \bullet {\downarrow}{\uparrow} B^+_m \vdash_F A^+_m \otimes B^+_m}\,{\downarrow}L}{\cdot \;;\; ({\downarrow}{\uparrow} A^+_m) \otimes ({\downarrow}{\uparrow} B^+_m) \vdash_F A^+_m \otimes B^+_m}\,\otimes L}{\Psi_1, \Psi_2, \Psi_3 \;;\; \cdot \vdash_F A^+_m \otimes B^+_m}\,\mathsf{cut}(3)$$

Here, $\mathsf{C} \in \sigma(\Psi_2)$ is used to apply $\otimes R$ in the left branch of the cut — we have omitted this in the proof tree for space reasons.

We now move on to the rules that explicitly mention modes — ${\uparrow}L$, ${\downarrow}R$, and $\multimap L$.

If the last rule used in the ADJ^I proof was ${\uparrow}L^0$, then $(\Psi)^\bullet = (\Psi')^\bullet, {\uparrow}^m_k (A^+_k)^\bullet$ for some Ψ', A^+_k. Since Ψ is suspension-normal, we can conclude that $\Psi = \Psi', {\uparrow}^m_m {\downarrow}^m_m \ldots {\uparrow}^m_m {\downarrow}^m_m {\uparrow}^m_k A^+_k$, and double shift removal allows us to assume that $\Psi = \Psi', {\uparrow}^m_k A^+_k$. We also get as premises of the ${\uparrow}L^0$ rule that if U is at mode ℓ, [8] then $k \geq \ell$, along with a proof that $(\Psi')^\bullet, (A^+_k)^\bullet \vdash_I (U_\ell)^\bullet$. This is also a proof that $(\Psi')^\bullet, ({\uparrow}^k_k A^+_k)^\bullet \vdash_I (U_\ell)^\bullet$, to which we can apply the inductive hypothesis to get that $\Psi', {\uparrow}^k_k A^+_k \;;\; \cdot \vdash_F U_\ell$. We can then build the following focused proof:

$$\dfrac{{\uparrow}^m_k A^+_k \geq k \geq \ell \qquad \dfrac{\dfrac{\dfrac{k \geq k \qquad \dfrac{\dfrac{\dfrac{}{\langle A^+_k \rangle \;;\; \cdot \vdash_F [A^+_k]}\,\mathsf{id}^+}{\langle A^+_k \rangle \;;\; \cdot \vdash_F A^+_k}\,\mathsf{focus}^+}{\cdot \;;\; A^+_k \vdash_F A^+_k}\,\eta^+}{\cdot \;;\; [{\uparrow}^m_k A^+_k] \vdash_F A^+_k}\,{\uparrow}L}{{\uparrow}^m_k A^+_k \;;\; \cdot \vdash_F A^+_k}\,(\mathsf{focus}^-)^0}{{\uparrow}^m_k A^+_k \;;\; \cdot \vdash_F {\uparrow}^k_k A^+_k}\,{\uparrow}R \qquad \Psi', {\uparrow}^k_k A^+_k \;;\; \cdot \vdash_F U_\ell}{\Psi', {\uparrow}^m_k A^+_k \;;\; \cdot \vdash_F U_\ell}\,\mathsf{cut}(4)$$

Here, the condition on modes in ${\uparrow}L^0$ of ADJ^I appears as a condition on the cut.

If the last rule used in the ADJ^I proof was ${\downarrow}R$, then $(U_m)^\bullet = {\downarrow}^\ell_m (A^-_\ell)^\bullet$. The same reasoning as in previous cases allows us to assume that $U_m = {\downarrow}^\ell_m A^-_\ell$. We also get from the premises of the rule that $\Psi = \Psi_1, \Psi_2$ with $\Psi_1 \geq \ell$ and $\mathsf{W} \in \sigma(\Psi_2)$, along with a proof that $(\Psi_1)^\bullet \vdash_I (A^-_\ell)^\bullet$.

[8] Technically, this requires that $(U)^\bullet$ is at mode ℓ, but this is equivalent to U being at mode ℓ.

Applying the inductive hypothesis (to a slightly modified form of the proof), we get that $\Psi_1 \; ; \cdot \vdash_F \downarrow^{-}\ell_{\ell} A^{-}_{\ell}$. We can then construct the following:

$$
\dfrac{\Psi_1 \geq \ell \geq m \quad \Psi_1 \; ; \cdot \vdash_F \downarrow^{\ell}_{\ell} A^{-}_{\ell} \qquad \dfrac{\dfrac{\dfrac{A^{-}_{\ell} \geq \ell \quad \mathsf{W} \in \sigma(\Psi_2) \quad \dfrac{\dfrac{\dfrac{}{\cdot \; ; [A^{-}_{\ell}] \vdash_F \langle A^{-}_{\ell} \rangle}\,\mathsf{id}^{-}}{A^{-}_{\ell} \; ; \cdot \vdash_F \langle A^{-}_{\ell} \rangle}\,(\mathsf{focus}^{-})^{0}}{A^{-}_{\ell} \; ; \cdot \vdash_F A^{-}_{\ell}}\,\eta^{-}}{\Psi_2, A^{-}_{\ell} \vdash_F [\downarrow^{\ell}_{m} A^{-}_{\ell}]}\,{\downarrow}R}{\Psi_2, A^{-}_{\ell} \; ; \cdot \vdash_F \downarrow^{\ell}_{m} A^{-}_{\ell}}\,\mathsf{focus}^{+}}{\Psi_2 \; ; \downarrow^{\ell}_{\ell} A^{-}_{\ell} \vdash_F \downarrow^{\ell}_{m} A^{-}_{\ell}}\,{\downarrow}L}{\Psi_1, \Psi_2 \; ; \cdot \vdash_F \downarrow^{\ell}_{m} A^{-}_{\ell}}\,\mathsf{cut}(3)
$$

Again, the condition on modes from the ADJ^{I} proof's $\downarrow R$ appears as a condition on the cut in the focused proof.

As a final example case, we consider $\multimap L^{0,0}$. In this case (via similar reasoning to the previous cases), we may assume that $\Psi = \Psi_1, \Psi_2, \Psi_3, A^{+}_{m} \multimap B^{-}_{m}$. The premises of the rule give us that $\Psi_1, \Psi_2 \geq m$, $\mathsf{C} \in \sigma(\Psi_2)$, and, after applying the inductive hypothesis to the premises, that $\Psi_1, \Psi_2 \; ; \cdot \vdash_F A^{+}_{m}$ and $\Psi_2, \Psi_3, B^{-}_{m} \vdash_F U_{\ell}$. Combining all this together, we construct the following two proofs that $A^{+}_{m} \multimap B^{-}_{m} \; ; \cdot \vdash_F (\downarrow\uparrow A^{+}_{m}) \multimap (\uparrow\downarrow B^{-}_{m})$ and $\Psi_1, \Psi_2, \Psi_3, (\downarrow\uparrow A^{+}_{m}) \multimap (\uparrow\downarrow B^{-}_{m}) \; ; \cdot \vdash_F U_{\ell}$, which can be cut together with the fourth case of Theorem 10 to give the desired result:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle A^{+}_{m} \rangle \geq m \quad \mathsf{C} \in \sigma(\cdot) \quad \dfrac{}{\langle A^{+}_{m} \rangle \vdash_F [A^{+}_{m}]}\,\mathsf{id}^{+} \quad \dfrac{}{\cdot \; ; [B^{-}_{m}] \vdash_F \langle B^{-}_{m} \rangle}\,\mathsf{id}^{-}}{\langle A^{+}_{m} \rangle \; ; [A^{+}_{m} \multimap B^{-}_{m}] \vdash_F \langle B^{-}_{m} \rangle}\,{\multimap}L}{A^{+}_{m} \multimap B^{-}_{m}, \langle A^{+}_{m} \rangle \; ; \cdot \vdash_F \langle B^{-}_{m} \rangle}\,(\mathsf{focus}^{-})^{0}}{A^{+}_{m} \multimap B^{-}_{m} \; ; A^{+}_{m} \vdash_F \langle B^{-}_{m} \rangle}\,\eta^{-}}{A^{+}_{m} \multimap B^{-}_{m} \; ; [\uparrow A^{+}_{m}] \vdash_F \langle B^{-}_{m} \rangle}\,{\uparrow}L}{A^{+}_{m} \multimap B^{-}_{m}, \uparrow A^{+}_{m} \; ; \cdot \vdash_F \langle B^{-}_{m} \rangle}\,(\mathsf{focus}^{-})^{0}}{A^{+}_{m} \multimap B^{-}_{m}, \uparrow A^{+}_{m} \; ; \cdot \vdash_F B^{-}_{m}}\,\eta^{+}}{A^{+}_{m} \multimap B^{-}_{m}, \uparrow A^{+}_{m} \vdash_F [\downarrow B^{-}_{m}]}\,{\downarrow}R}{A^{+}_{m} \multimap B^{-}_{m}, \uparrow A^{+}_{m} \; ; \cdot \vdash_F \downarrow B^{-}_{m}}\,\mathsf{focus}^{+}}{A^{+}_{m} \multimap B^{-}_{m}, \uparrow A^{+}_{m} \; ; \cdot \vdash_F \uparrow\downarrow B^{-}_{m}}\,{\uparrow}R}{A^{+}_{m} \multimap B^{-}_{m} \; ; \downarrow\uparrow A^{+}_{m} \vdash_F \uparrow\downarrow B^{-}_{m}}\,{\downarrow}L}{A^{+}_{m} \multimap B^{-}_{m} \; ; \cdot \vdash_F (\downarrow\uparrow A^{+}_{m}) \multimap (\uparrow\downarrow B^{-}_{m})}\,{\multimap}R
$$

$$
\dfrac{\dfrac{\Psi_1, \Psi_2 \geq m \quad \mathsf{C} \in \sigma(\Psi_2) \quad \dfrac{\dfrac{\Psi_1, \Psi_2 \; ; \cdot \vdash_F A^{+}_{m}}{\Psi_1, \Psi_2 \; ; \cdot \vdash_F \uparrow A^{+}_{m}}\,{\uparrow}R}{\Psi_1, \Psi_2 \vdash_F [\downarrow\uparrow A^{+}_{m}]}\,{\downarrow}R \quad \dfrac{\dfrac{\Psi_2, \Psi_3, B^{-}_{m} \; ; \cdot \vdash_F U_{\ell}}{\Psi_2, \Psi_3 \; ; \downarrow B^{-}_{m} \vdash_F U_{\ell}}\,{\downarrow}L}{\Psi_2, \Psi_3 \; ; [\uparrow\downarrow B^{-}_{m}] \vdash_F U_{\ell}}\,{\uparrow}L}{\Psi_1, \Psi_2, \Psi_3 \; ; [(\downarrow\uparrow A^{+}_{m}) \multimap (\uparrow\downarrow B^{-}_{m})] \vdash_F U_{\ell}}\,{\multimap}L}{\Psi_1, \Psi_2, \Psi_3, (\downarrow\uparrow A^{+}_{m}) \multimap (\uparrow\downarrow B^{-}_{m}) \; ; \cdot \vdash_F U_{\ell}}\,\mathsf{focus}^{-}
$$

□

At face value, our focalization theorem only applies to certain sequents — those which are the erasure of some stable, suspension-normal sequent $\Psi \ ; \ \cdot \vdash_F U$. In order to have a proper focalization result, that every provable ADJ^I sequent has a provable focused counterpart, we need to also show that every ADJ^I sequent can be polarized into a stable, suspension-normal sequent, solely by the addition of shifts from a mode to itself. Since negative propositions are stable on the left and positive propositions are positive on the right, it will suffice to show that we can polarize every $\Psi \vdash_I B_m$ such that Ψ is negative and B_m is positive.

Lemma 5 (Existence of Polarizations)**.** *Suppose $\Psi \vdash_I B_m$. Then, there exists a polarized sequent $\Psi^- \vdash_I B^+_m$ which differs from the original sequent only by the addition of shifts from a mode to itself (i.e. of the form $\uparrow^m_m$ or $\downarrow^m_m$).*

Proof. It will suffice to show that any given proposition B_m can be given a valid polarization — if this yields a positive B^+_m, we can construct the negative proposition $\uparrow^m_m B^+_m$, and likewise, if it yields a negative B^-_m, we can construct $\downarrow^m_m B^-_m$. In either case, whether this B_m occurs in the antecedents or the succedent, we can give it the appropriate polarity.

That each B_m can be given a valid polarization follows by induction on the structure of B_m.

If B_m is an atom p_m, we will arbitrarily take this atom to be negative.[9]

Otherwise, we apply the inductive hypothesis to each direct subformula of B_m, insert shifts as necessary, and then use the top-level connective of B_m to build up a new $B^\pm_m$.

We consider two example cases here, first $\&_{j\in J} B^j_m$, and then $\downarrow^\ell_m B_\ell$.

If $B_m = \&_{j\in J} B^j_m$, we apply the inductive hypothesis to each B^j_m to get a polarization $(B^j_m)^\pm$. For each $(B^j_m)^-$, let $(B^j_m)^- = (B^j_m)^-$, and for each $(B^j_m)^+$, let $(B^j_m)^- = \uparrow^m_m (B^j_m)^+$. Then, define $B^-_m = \&_{j\in J}(B^j_m)^-$.

If $B_m = \downarrow^\ell_m B_\ell$, we apply the inductive hypothesis to B_ℓ to get a polarization $B^\pm_\ell$. If this polarization is already negative, we take $B^+_m = \downarrow^\ell_m B^-_\ell$. Otherwise, we take $B^+_m = \downarrow^\ell_m(\uparrow^\ell_\ell B^+_\ell)$.

The other cases are similar. □

Combining the existence of polarizations with our focalization result gives us that any provable ADJ^I sequent has a provable focused counterpart.[10]

2.4 Embeddings of example logics

A variety of logics from the computer science literature can be represented as instances of adjoint logic, potentially with some restrictions on which propositions are allowed at each mode.

Example 2 (Linear logic)**.** *We obtain intuitionistic linear logic [6, 35, 36] by using two modes,* U *(for* unrestricted *or structural) and* L *(for* linear*) with* $\mathsf{U} > \mathsf{L}$*. Moreover,* $\sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$ *and* $\sigma(\mathsf{L}) = \{\ \}$*, and the structural layer* U *contains only propositions of the form* $\uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}$.

[9] Polarizations where all atoms are positive are, of course, also possible, as are those where atoms have varying polarity. For simplicity, however, we consider only this single case, as we are currently primarily concerned with *existence* of polarizations.

[10] Indeed, many such counterparts exist — as we have seen, both polarization and erasure are non-deterministic.

In this representation the exponential modality is decomposed into shift modalities $!A_{\mathsf{L}} = {\downarrow}^{\mathsf{U}}_{\mathsf{L}}{\uparrow}^{\mathsf{U}}_{\mathsf{L}}A_{\mathsf{L}}$*, while all other connectives are represented as (the* L*-version of) themselves.*

Example 3 (LNL)**.** *We obtain Benton's LNL [7] much like linear logic with two modes* $\mathsf{U} > \mathsf{L}$*, but here, rather than only allowing* ${\uparrow}^{\mathsf{U}}_{\mathsf{L}}A_{\mathsf{L}}$ *in the structural layer, here, we allow the full set of (multiplicative) connectives, where we write* $\times = \otimes_{\mathsf{U}}$ *and* $\rightarrow = \multimap_{\mathsf{U}}$*. Of course, we can work with additive connectives as well, but LNL avoids them for technical reasons, and so we omit them here.*

Benton's notation for shifts uses F *and* G *for down and up, respectively, and his first formulation of LNL uses two separate contexts for linear and unrestricted propositions, whereas we mix both types of propositions into a single context, as in the "parsimonious" presentation of LNL [7, Technical Report]. We then rely on the declaration of independence to force that unrestricted succedents depend only on unrestricted antecedents, which, in Benton's work, is accomplished via a lemma stating that any provable sequent satisfies this condition.*

As we have mentioned before, some further examples embed more naturally into ADJ^{I}, because they are primarily structural in nature, and so are generally also presented with implicit structural rules.

Example 4 (Judgmental S4)**.** *The modal logic S4 adds two modal operators* $\Box$ *and* $\Diamond$ *to a structural logic. To model* $\Box$*, we look to a judgmental presentation of S4 with separate contexts for valid and true propositions [75]. The fragment of S4 without* $\Diamond$ *then arises from two modes* V *(validity) and* U *(truth) with* $\mathsf{V} > \mathsf{U}$ *and* $\sigma(\mathsf{V}) = \sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$*. The declaration of independence here expresses that validity is categorical with respect to truth—that is, a proof of* A_{V} *must not depend on any hypotheses of the form* B_{U}*. Previous calculi enforced this instead by separating the antecedents into two zones, much like the linear and unrestricted contexts of linear logic.*

Analogous to the encoding of linear logic, we only need to allow ${\uparrow}^{\mathsf{V}}_{\mathsf{U}}A_{\mathsf{U}}$ *in the validity layer. Under that interpretation, we encode* $\Box A_{\mathsf{U}} = {\downarrow}^{\mathsf{V}}_{\mathsf{U}}{\uparrow}^{\mathsf{V}}_{\mathsf{U}}A_{\mathsf{U}}$*, which is entirely analogous to the representation of* $!A$ *in linear logic. This type of double shift, first up, then down, allows us to model* comonads.

Note that we cannot easily model $\Diamond A$*, which is not a* normal *modality in the technical sense that it does* not *satisfy* $\Diamond(A \multimap B) \multimap (\Diamond A \multimap \Diamond B)$*. Reed [88] provides a less direct, but adequate encoding, while Licata et al. [59] use the* 2*-categorical structure that generalizes our preorder to provide a more elegant representation, so this is expressible in an adjoint framework, if not in our presentation of it.*

Example 5 (Lax logic)**.** *Lax logic [31, 75] encodes a weaker form of truth called* lax truth*. We can represent it as a substructural adjoint logic with two modes,* $\mathsf{U} > \mathsf{X}$*, and* $\sigma(\mathsf{U}) = \sigma(\mathsf{X}) = \{\mathsf{W}, \mathsf{C}\}$*. As in the previous examples, we restrict the lax layer* X *to a single form of proposition* ${\downarrow}^{\mathsf{U}}_{\mathsf{X}}A_{\mathsf{U}}$*, which is sufficient for us to define the lax modality* $\bigcirc A_{\mathsf{U}} = {\uparrow}^{\mathsf{U}}_{\mathsf{X}}{\downarrow}^{\mathsf{U}}_{\mathsf{X}}A_{\mathsf{U}}$*. All other connectives are straightforwardly encoded as themselves.*

The double shift ${\uparrow}^{\mathsf{U}}_{\mathsf{X}}{\downarrow}^{\mathsf{U}}_{\mathsf{X}}$ *that we use to represent the lax modality is an example of a (strong) monad. In fact, such double shifts will always form strong monads.*

Chapter 3

The Semi-Axiomatic Sequent Calculus

The proof theory that we present in chapter 2 is based on the sequent calculus, which is well-suited to giving several different presentations of adjoint logic, and to reasoning about the structure of proofs. Sequent calculus presentations of logic can be directly used as a basis for programming languages, with a notable example given by Caires and Pfenning's interpretation of the linear sequent calculus as a form of concurrent programming language [12]. However, this approach yields a language where all communication is *synchronous* — a process cannot send a message unless its intended recipient is already waiting to receive that message. This presents a few problems in our adjoint setting — firstly, that we may simply wish to model asynchronous communication (for instance, if we want to model shared memory, rather than message-passing), and secondly and more critically, there are circumstances where synchronous communication simply does not make sense. For instance, consider a multicast message with multiple recipients — do all recipients of the message need to be waiting at the same time in order for this message to be sent? How would all of these participants synchronize? In our attempts to resolve these issues and provide a firm semantics for a language based on adjoint logic, but which supports asynchronous communication, we incidentally also developed a new presentation of logic, the *semi-axiomatic sequent calculus* [26], which we will examine in this chapter. This prior publication focuses on the proof theory of the *structural* semi-axiomatic calculus. Here, we will examine the adjoint case, which adds some additional complexities (but is also simpler in other aspects — working with explicit structural rules, which we will do here, can be convenient proof-theoretically). While the semi-axiomatic sequent calculus is not inherently related to adjoint logic, it is nevertheless a contribution of this thesis, and is used extensively in the following chapters as the basis for programming languages.

3.1 Recovering Asynchrony via Cut

In the usual interpretation of the linear sequent calculus as a basis for programming [12], we interepret cut reductions as computation steps — a message is sent (and received) via a principal cut reduction, between a right rule and a corresponding left rule. For instance, consider the following cut between $\oplus R^i$ and $\oplus L$ (using the system ADJ^E):

$$\dfrac{\Gamma_1 \geq m \geq k \quad \dfrac{i \in J \quad \overset{\mathcal{D}}{\Gamma_1 \vdash A^i_m}}{\Gamma_1 \vdash \oplus_{j \in J} A^j_m}\,\oplus R^i \quad \dfrac{\overset{\mathcal{E}_j}{\Gamma_2, A^j_m \vdash C_k} \quad \text{for all } j \in J}{\Gamma_2, \oplus_{j \in J} A^j_m \vdash C_k}\,\oplus L}{\Gamma_1, \Gamma_2 \vdash C_k}\,\mathsf{cut}$$

This cut reduces to the following proof, using that $i \in J$ to conclude that $\mathcal{E}_i$ must exist:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash A^i_m} \quad \overset{\mathcal{E}_i}{\Gamma_2, A^i_m \vdash C_k}}{\Gamma_1, \Gamma_2 \vdash C_k}\,\mathsf{cut}$$

Here, the rule $\oplus L$, being a positive left rule, is invertible, with its premises being provable from its conclusion, and so contains no information. It is therefore natural to view this reduction as a transfer of information (the single label i) from the process represented by the $\oplus R^i$ to the process represented by the $\oplus L$, selecting a branch in the latter.

This is *synchronous* in that both processes evolve at once. To make it asynchronous, we would like for the sending process and the receiving process to be able to take their send/receive steps separately. One approach to this, which is often taken in the π-calculus, is to make sending a message a terminal action. For a process to send a message and continue, then, it must spawn a separate process whose sole role is to send that message. We may even identify this short-lived process with the message itself, so that the spawning of this process is a message send. If we want our message (containing the label i) to disappear upon receipt, then the corresponding rule should have no premises — that is, it should be an *axiom*. If we assume that the premise of the $\oplus R^i$ rule is an identity (which then has no further premises), we get the following (ADJ^E) sequent calculus proof, which we take as the basis for the axiomatic rule $\oplus R^0_i$:

$$\dfrac{i \in J \quad \dfrac{}{A^i_m \vdash A^i_m}\,\mathsf{id}}{A^i_m \vdash \oplus_{j \in J} A^j_m}\,\oplus R^i \qquad \text{becomes} \qquad \dfrac{i \in J}{A^i_m \vdash \oplus_{j \in J} A^j_m}\,\oplus R^0_i$$

In order to prove $\oplus_{j \in J} A^j_m$, all we need is to have some $i \in J$ for which we can establish A^i_m. Now, if we examine a cut reduction between $\oplus R^0_i$ and $\oplus L$, we see that the message (represented by $\oplus R^0_i$) disappears, while still selecting the i branch in $\oplus L$:

$$\dfrac{m \geq k \quad \dfrac{i \in J}{A^i_m \vdash \oplus_{j \in J} A^j_m}\,\oplus R^0_i \quad \dfrac{\overset{\mathcal{E}_j}{\Gamma_2, A^j_m \vdash C_k} \quad \text{for all } j \in J}{\Gamma_2, \oplus_{j \in J} A^j_m \vdash C_k}\,\oplus L}{A^i_m, \Gamma_2 \vdash C_k}\,\mathsf{cut}$$

reduces to

$$\overset{\mathcal{E}_i}{\Gamma_2, A^i_m \vdash C_k}$$

Similarly, if we replace the other non-invertible rules of the sequent calculus with axiomatic forms, we have, at least at an intuitive level, made it possible to work with asynchronous communication. Sending a message is accomplished via a cut with one of these new axioms, and receiving a message behaves much the same as in the synchronous case, except that the message is made explicit as a process represented by an axiom.

$$\frac{}{A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Gamma_1 \geq m \geq k \quad \Gamma_1 \vdash A_m \quad \Gamma_2, A_m \vdash C_k}{\Gamma_1, \Gamma_2 \vdash C_k}\ \mathsf{cut}$$

$$\frac{\mathsf{W} \in \sigma(m) \quad \Gamma \vdash C_k}{\Gamma, A_m \vdash C_k}\ \mathsf{weaken} \qquad \frac{\mathsf{C} \in \sigma(m) \quad \Gamma, A_m, A_m \vdash C_k}{\Gamma, A_m \vdash C_k}\ \mathsf{contract}$$

$$\frac{i \in J}{A_m^i \vdash \oplus_{j \in J} A_m^j}\ \oplus R_i^0 \qquad \frac{\Gamma, A_m^j \vdash C_k \quad \text{for all } j \in J}{\Gamma, \oplus_{j \in J} A_m^j \vdash C_k}\ \oplus L$$

$$\frac{\Gamma \vdash A_m^j \quad \text{for all } j \in J}{\Gamma \vdash \&_{j \in J} A_m^j}\ \& R \qquad \frac{i \in J}{\&_{j \in J} A_m^j \vdash A_m^i}\ \& L_i^0$$

$$\frac{}{A_m, B_m \vdash A_m \otimes B_m}\ \otimes R^0 \qquad \frac{\Gamma, A_m, B_m \vdash C_k}{\Gamma, A_m \otimes B_m \vdash C_k}\ \otimes L$$

$$\frac{\Gamma, A_m \vdash B_m}{\Gamma \vdash A_m \multimap B_m}\ \multimap R \qquad \frac{}{A_m, A_m \multimap B_m \vdash B_m}\ \multimap L^0$$

$$\frac{}{\cdot \vdash \mathbf{1}_m}\ \mathbf{1}R^0 \qquad \frac{\Gamma \vdash C_k}{\Gamma, \mathbf{1}_m \vdash C_k}\ \mathbf{1}L$$

$$\frac{}{A_\ell \vdash {\downarrow}_m^\ell A_\ell}\ {\downarrow}R^0 \qquad \frac{\Gamma, A_\ell \vdash C_k}{\Gamma, {\downarrow}_m^\ell A_\ell \vdash C_k}\ {\downarrow}L$$

$$\frac{\Gamma \vdash A_k}{\Gamma \vdash {\uparrow}_k^m A_k}\ {\uparrow}R \qquad \frac{}{{\uparrow}_k^m A_k \vdash A_k}\ {\uparrow}L^0$$

Figure 3.1: A semi-axiomatic presentation of ADJ^E

3.2 The Semi-Axiomatic Sequent Calculus SAX

With the motivation of the previous section in mind, we will now show a full semi-axiomatic presentation of ADJ^E. Note that while we use ADJ^E as an example here, we can likewise transform ADJ^I to a semi-axiomatic form, and this will serve as a basis for several of the programming languages in Chapter 4.

One interesting side-effect of presenting adjoint logic in a semi-axiomatic form is that nearly all of the side conditions on modes disappear — $\downarrow R$, $\uparrow L$, and $\multimap L$ all have conditions on the modes of their components in a sequent calculus presentation, but these conditions are all trivially satisfied in their axiomatic forms, and so do not appear explicitly in the rules. In essence, the conditions needed to use these rules in a more general context are subsumed by the conditions on cut.

A natural question about this semi-axiomatic calculus is whether it can derive the same sequents as the usual sequent calculus — after all, we have severely restricted the contexts in which the non-invertible rules may be used. In fact, these two calculi derive (in the presence of cut) exactly the same sequents, and, moreover, the rules of the semi-axomatic calculus can be used to prove the rules of the regular sequent calculus, and vice versa. We express this as the following theorem (which, while only described here for ADJ^E, can be generalized to ADJ^I, and to other logics):

Theorem 13. *Each rule of the semi-axiomatic form of* ADJ^E *(Figure 3.1) is derivable in a sequent calculus presentation of* ADJ^E*, and vice versa.*

Consequently, a sequent $\Gamma \vdash A_m$ *is provable in the sequent calculus presentation of* ADJ^E *if and only if it is provable in the semi-axiomatic presentation of* ADJ^E*.*

Proof. Each axiom of the semi-axiomatic calculus is derivable (as our method of constructing the axioms suggests) from the corresponding non-axiom rule, by constructing a proof where all premises of that rule are applications of the identity. For instance, consider the left implication rule ($\multimap R$ is already the same in both calculi):

$$\frac{A_m \geq m \qquad \dfrac{}{A_m \vdash A_m}\,\mathsf{id} \qquad \dfrac{}{B_m \vdash B_m}\,\mathsf{id}}{A_m, A_m \multimap B_m \vdash B_m}\,{\multimap} L$$

We see that this proof gives exactly $\multimap L^0$.

In the other direction, we make use of cut to apply an axiom in a more general setting. Again examining the left implication case, we see that by cutting together $\multimap L^0$ with the premises of the $\multimap L$ rule, we recover $\multimap L$:

$$\frac{{\color{red}\Gamma_1 \geq m} \geq k \qquad \Gamma_1 \vdash A_m \qquad \dfrac{A_m, A_m \multimap B_m \geq m \geq k \qquad \dfrac{}{A_m, A_m \multimap B_m \vdash B_m}\,{\multimap} L^0 \qquad \Gamma_2, B_m \vdash C_k}{\Gamma_2, A_m, A_m \multimap B_m \vdash C_k}\,\mathsf{cut}}{\Gamma_1, \Gamma_2, A_m \multimap B_m \vdash C_k}\,\mathsf{cut}$$

Note that when read bottom-up, several of the side conditions on modes are automatically satisfied (assuming the conclusion $\Gamma_1, \Gamma_2, A_m \multimap B_m \vdash C_k$ is well-formed). Only that $\Gamma_1 \geq m$ (highlighted in red) still needs to be proven, since we can infer $m \geq k$ from well-formedness of the bottom sequent, and $A_m, A_m \multimap B_m \geq m$ is trivially true. Thus, despite the more complicated side conditions, only the one that occurs in $\multimap L$ to begin with needs to be proven.

The other axioms can be shown to be interderivable with their non-axiomatic counterparts in a similar manner, making use of cut and identity, and so for a given logic, we can directly translate between SAX proofs and sequent calculus proofs. □

3.3 Cut Elimination for SAX

One immediate observation from the translations above is that cut elimination is no longer obvious for SAX — we need to use cuts to recover the original sequent calculus rules, so we cannot simply rely on cut elimination in the sequent calculus to get cut elimination in SAX. In fact, full cut elimination is not possible in SAX. We can, however, still prove a form of normalization, where only certain types of cuts are allowed. In our recovery of the $\multimap L$ rule from $\multimap L^0$ (and indeed in the other such derivations of the original sequent calculus rules from their axiomatic counterparts), all of the cuts that we use are *analytic* — that is, the cut formula is a subformula of some other part of the conclusion of the cut. If we allow only analytic cuts, while we do not have full cut elimination, we do still retain the subformula property, as these cuts do not introduce any new formulae. Because we only use analytic cuts in translating from SAX to the sequent calculus, we can use cut elimination for the sequent calculus along with our translations between sequent calculus and SAX to prove the subformula property, simply by translating to sequent calculus, eliminating cuts, and translating back, producing only analytic cuts. However, this approach is too imprecise to provide a clear relation to computational behavior, and so we work instead with a more limited form of cut which we call a *snip*, allowed to be used only if the snip formula is used as a sub-principal formula in an axiom in one or both of the premises of the snip.

To be precise, we label each occurrence of a formula with a variable, so that we can distinguish between two different copies of the same formula A_m. For each axiom, some number of immediate sub-formulae of the principal formula appear. Consider for example $\multimap L^0$:

$$\frac{}{x : A_m^*, y : A_m \multimap B_m \vdash z : B_m^*} \multimap L^0$$

Here, the principal formula is $A_m \multimap B_m$, and the sub-principal formulae are A_m and B_m, which we mark with a $*$ to denote that they are ***eligible*** to be used in a snip. This information can then be propagated downwards through a derivation to determine whether a given cut is a snip or not. For each rule, we mark variables as eligible in the conclusion if they are eligible in at least one premise of the rule.

Snips are then cuts with one of the following forms:[1]

$$\frac{\Gamma_1 \geq m \geq k \quad \Gamma_1 \vdash x : A_m^* \quad \Gamma_2, x : A_m \vdash z : C_k}{\Gamma_1, \Gamma_2 \vdash z : C_k} \mathsf{Snip}^1$$

$$\frac{\Gamma_1 \geq m \geq k \quad \Gamma_1 \vdash x : A_m \quad \Gamma_2, x : A_m^* \vdash z : C_k}{\Gamma_1, \Gamma_2 \vdash z : C_k} \mathsf{Snip}^2$$

Just as we can make use of cut elimination in the sequent calculus to prove that we can eliminate all non-analytic cuts in SAX, we can likewise show that we can eliminate all non-snip cuts in SAX by the same method, showing that the cuts introduced by translating from the sequent calculus to SAX are in fact snips. It is also possible to directly prove admissibility of cut in SAX with snips, and thereby eliminate non-snip cuts directly, and this approach and its cut reductions will form the basis of computational behavior of SAX-based programming languages. We will first present in detail cut elimination for the semi-axiomatic presentation of ADJ^E, but note that (as for the sequent calculus), implicit and explicit structural rules require different proof strategies to handle them.

As is typical, and as in our previous cut elimination results, we first show that cut is admissible, and this result comprises the bulk of the proof. Note that in what follows, we will write "cut-free" to describe proofs in which all cuts are snips, rather than proofs which contain no cuts whatsoever.

[1] If the cut formula A_m is eligible in both premises of the snip, we may freely label it as either Snip^1 or Snip^2.

Since we are working with explicit structural rules, we will need to generalize our cut admissibility result to work with multicut, as in Section 2.1.1.

Theorem 14 (Multicut admissibility for SAX)**.** *Suppose we have cut-free proofs $\mathcal{D}$ of $\Gamma_1 \vdash A_m$ and $\mathcal{E}$ of $\Gamma_2, A_m^n \vdash z : C_k$ in SAX, and that $\Gamma_1 \geq m \geq k$ and $n \in \mu(m)$.*

Then, there is a cut-free SAX proof $\mathcal{F}$ of $\Gamma_1, \Gamma_2 \vdash z : C_k$.

Proof. We will prove a slightly stronger statement in order to simplify our induction, adding the following condition on formulae in Γ_1, Γ_2. Each $y : B_\ell$ in Γ_1, Γ_2 which is eligible in $\mathcal{D}$ or $\mathcal{E}$ must either also be eligible in $\mathcal{F}$, or B_ℓ must be a strict subformula of A_m. This is necessary in the cases where either $\mathcal{D}$ or $\mathcal{E}$ ends in a snip (which is possible due to our relaxed definition of "cut-free').

To avoid a need for "multisnips" and the associated question of what needs to be eligible for a multisnip to be used, we will take a slightly different approach here than in proving cut admissible in the sequent calculus presentation of ADJ^E, in which we first reduce all multicuts to ordinary cuts, and then can handle all remaining cases assuming that $n = 1$.

We then proceed by induction on the (lexicographically ordered) quadruple $(A_m, \mathcal{D}, \mathcal{E}, n)$. Here, n is necessary to allow us to reduce multicuts on $n > 1$ to simple cuts (with $n = 1$).

Reducing Multicut to Cut There are two cases to consider here, where $n = 0$, and where $n > 1$.

If $n = 0$, then, since $n \in \mu(m)$, we know that $\mathsf{W} \in \sigma(m)$. We also know that $\Gamma_1 \geq m$, so $\mathsf{W} \in \sigma(\Gamma_1)$ as well, meaning that we can freely weaken (in several steps) all of Γ_1. We then construct the following proof:

$$\begin{array}{c} \qquad\qquad\quad \mathcal{E} \\ \mathsf{W} \in \sigma(\Gamma_1) \quad \Gamma_2 \vdash z : C_k \\ \qquad \vdots\ \mathsf{weaken}^* \\ \Gamma_1, \Gamma_2 \vdash z : C_k \end{array}$$

If $n > 1$, since $n \in \mu(m)$, we know that $\mathsf{C} \in \sigma(m)$, and so also, since $\Gamma_1 \geq m$, $\mathsf{C} \in \sigma(\Gamma_1)$. This means that we may repeatedly apply contraction to duplicate all of Γ_1. We then choose $i, j \geq 1$ such that $i + j = n$, and construct the following proof, noting that $i, j < n$, and so we may apply the inductive hypothesis:

$$\mathsf{C} \in \sigma(\Gamma_1) \quad \begin{array}{c} \dfrac{i \in \mu(m) \quad \Gamma_1 \vdash A_m \qquad \dfrac{j \in \mu(m) \quad \Gamma_1 \vdash A_m \quad \Gamma_2, A_m^i, A_m^j \vdash z : C_k}{\Gamma_1, \Gamma_2, A_m^i \vdash z : C_k}\ \text{i.h.}(A_m, \mathcal{D}, \mathcal{E}, j)}{\Gamma_1, \Gamma_1, \Gamma_2 \vdash z : C_k}\ \text{i.h.}(A_m, \mathcal{D}, \mathcal{E}, i) \\ \vdots\ \mathsf{contract}^* \\ \Gamma_1, \Gamma_2 \vdash z : C_k \end{array}$$

We omit the condition $\Gamma_1 \geq m \geq k$ from the uses of the inductive hypothesis for space reasons, but it holds by assumption in both cases. That $i, j \in \mu(m)$ follows from the fact that $\mathsf{C} \in \sigma(m)$, and so $\mu(m)$ contains all positive integers.

At this point, we have covered all cases where $n \neq 1$, and so in what remains, we may freely assume that $n = 1$. We will, however, make use of the inductive hypothesis at $n \neq 1$ in cases involving the structural rules, and it is for this reason that we needed to strengthen our result to eliminate multicut rather than working directly with cut.

Eligible Cases In the next two cases we consider, $x : A_m$ is already eligible in either $\mathcal{D}$ or $\mathcal{E}$, and we can construct a snip between $\mathcal{D}$ and $\mathcal{E}$ to give the desired result, noting that any variable eligible in $\mathcal{D}$ or $\mathcal{E}$ (besides $x : A_m$ itself, which disappears from $\mathcal{F}$) will also be eligible in $\mathcal{F}$. We note that these cases cover all instances where a right axiom ($\oplus R_i^0, \otimes R^0, \mathbf{1}R^0, {\downarrow}R^0$) is the last rule used in $\mathcal{E}$ or a left axiom ($\& L_i^0, \multimap L^0, {\uparrow}L^0$) is the last rule used in $\mathcal{D}$, as the cut formula must necessarily be eligible in these cases. We also cover the case where $\multimap L^0$ is the last rule used in $\mathcal{E}$ and the cut formula is the left-hand side of the implication, rather than the whole implication itself.

Identity Cases The identity cases are similar to a typical proof of cut admissibility. If $\mathcal{D}$ ends in an identity, then $\mathcal{E}$ is almost exactly the proof we need. Substituting to replace x gives the desired proof, as shown in the reduction below:

$$\dfrac{(y : A_m) \geq m \geq k \quad \dfrac{}{y : A_m \vdash x : A_m}\,\mathsf{id} \quad \overset{\mathcal{E}}{\Gamma_2, x : A_m \vdash z : C_k}}{\Gamma_2, y : A_m \vdash z : C_k}\,\mathsf{cut} \quad \Longrightarrow \quad \overset{\mathcal{E}[y/x]}{\Gamma_2, y : A_m \vdash z : C_k}$$

Symmetrically, if $\mathcal{E}$ ends in an identity, we can substitute in $\mathcal{D}$ to give the desired proof:

$$\dfrac{\Gamma_1 \geq k \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : C_k} \quad \dfrac{}{x : C_k \vdash z : C_k}\,\mathsf{id}}{\Gamma_1 \vdash z : C_k}\,\mathsf{cut} \quad \Longrightarrow \quad \overset{\mathcal{D}[z/x]}{\Gamma_1 \vdash z : C_k}$$

Structural Cases The next set of cases, which involve a structural rule as the last rule used in either $\mathcal{D}$ or $\mathcal{E}$, are those that are most distinctly different from any cases in the proof of cut for structural SAX. In several of these cases, where the structural rule was the last rule in $\mathcal{D}$ or was the last rule in $\mathcal{E}$, but does not have the cut formula as its principal formula, we can exchange the cut and the structural rule in the proof, allowing us to apply the inductive hypothesis (with the same cut formula A_m, but a smaller proof for either $\mathcal{D}$ or $\mathcal{E}$). We show one example of these cases, where $\mathcal{D}$ is arbitrary, and $\mathcal{E}$ ends in a contraction applied to a formula B_ℓ other than the cut formula, reducing the following proof:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : A_m} \quad \dfrac{C \in \sigma(\ell) \quad \overset{\mathcal{E}_1}{\Gamma_2, x : A_m, y : B_\ell, w : B_\ell \vdash z : C_k}}{\Gamma_2, x : A_m, y : B_\ell \vdash z : C_k}\,\mathsf{contract}}{\Gamma_1, \Gamma_2, y : B_\ell \vdash z : C_k}\,\mathsf{cut}$$

to this proof, relying on the inductive hypothesis:

$$\dfrac{C \in \sigma(\ell) \quad \dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : A_m} \quad \overset{\mathcal{E}_1}{\Gamma_2, x : A_m, y : B_\ell, w : B_\ell \vdash z : C_k}}{\Gamma_1, \Gamma_2, y : B_\ell, w : B_\ell \vdash z : C_k}\,\text{i.h.}(A_m, \mathcal{D}, \mathcal{E}_1)}{\Gamma_1, \Gamma_2, y : B_\ell \vdash z : C_k}\,\mathsf{contract}$$

We observe here that if any formula is eligible in $\mathcal{E}$, it must also be eligible in $\mathcal{E}_1$, as $\mathcal{E}_1$ is a premise of the last rule used in $\mathcal{E}$. As such, any formula eligible in $\mathcal{D}$ or $\mathcal{E}$ is also eligible in the resulting proof $\mathcal{F}$, since it will be (by the inductive hypothesis) eligible in the premise of the contraction rule.

If, instead, the cut formula is also the principal formula of the structural rule, we may conclude almost immediately by applying our inductive hypothesis. For example, we look at the case where $\mathcal{E}$ ends in a weakening of the cut formula:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : A_m} \quad \dfrac{\mathsf{W} \in \sigma(m) \quad \overset{\mathcal{E}_1}{\Gamma_2 \vdash z : C_k}}{\Gamma_2, x : A_m \vdash z : C_k}\,\mathsf{weaken}}{\Gamma_1, \Gamma_2 \vdash z : C_k}\,\mathsf{cut}$$

In this case, applying the inductive hypothesis to $(A_m, \mathcal{D}, \mathcal{E}_1)$ gives the desired result, using $\mathsf{W} \in \sigma(m)$ from $\mathcal{E}$ to justify that we may perform a multicut with 0 copies of A_m. Here, our desired eligibility condition comes immediately from the inductive hypothesis.

The remaining structural cases (three more commuting cases, and the principal case for contraction) behave similarly to those shown already.

Commuting Cases For non-axiom connective rules, commuting cases are identical to those for cut admissibility in the sequent calculus, needing only to verify that our eligibility condition holds. We show one example of such a case, where the last rule in $\mathcal{D}$ is $\oplus L$:

$$\dfrac{\Gamma_1, y : \oplus_{j \in J} B_\ell^j \geq m \geq k \quad \dfrac{\overset{\mathcal{D}_i}{\Gamma_1, w : B_\ell^i \vdash x : A_m} \quad \text{for all } i \in J}{\Gamma_1, y : \oplus_{j \in J} B_\ell^j \vdash x : A_m}\,\oplus L \quad \overset{\mathcal{E}}{\Gamma_2, x : A_m \vdash z : C_k}}{\Gamma_1, \Gamma_2, y : \oplus_{j \in J} B_\ell^j \vdash z : C_k}\,\mathsf{cut}$$

We note that if any formula (other than $y : \oplus_{j \in J} B_\ell^j$, which cannot be eligible in $\mathcal{D}$ to begin with) is eligible in $\mathcal{D}$, it is also eligible in $\mathcal{D}_i$, by definition of eligibility. We then construct the following proof:

$$\dfrac{\dfrac{\Gamma_1, w : B_\ell^i \geq m \geq k \quad \overset{\mathcal{D}_i}{\Gamma_1, w : B_\ell^i \vdash x : A_m} \quad \overset{\mathcal{E}}{\Gamma_2, x : A_m \vdash z : C_k}}{\Gamma_1, \Gamma_2, w : B_\ell^i \vdash z : C_k}\,\text{i.h.}(A_m, \mathcal{D}_i, \mathcal{E}) \quad \text{for all } i \in J}{\Gamma_1, \Gamma_2, y : \oplus_{j \in J} B_\ell^j \vdash z : C_k}\,\oplus L$$

We can see that by the inductive hypothesis, any formula which is eligible in either $\mathcal{D}_i$ or $\mathcal{E}$ is also eligible in the premise of $\oplus L$, and therefore also in $\mathcal{F}$ (other than $w : B_\ell^i$, which, while it may be eligible in $\mathcal{D}_i$, cannot have been eligible in $\mathcal{D}$), giving our desired eligibility condition. The other commuting cases behave similarly.

For axioms, commuting cases do not exist — this is because the side formulae of axioms are always eligible, and so we fall back to the eligible cases presented above.

However, we have a third class of commuting cases as well, which did not occur in the sequent calculus case. Since our cut-free proofs may contain snips, we must also consider these snips as possible last rules for $\mathcal{D}$ and $\mathcal{E}$, and since the snip formula of a snip does not appear in its conclusion, all cases involving snips are commuting cases. There are four such cases, depending on whether $\mathcal{D}$ or $\mathcal{E}$ ends with a snip, and whether it is a $\textsf{Snip}^1$ or a $\textsf{Snip}^2$, but all are similar.

We examine first the case where $\mathcal{D}$ ends with a $\textsf{Snip}^1$:

$$\dfrac{\Gamma_1,\Gamma_2 \geq m \geq k \quad \dfrac{\Gamma_1 \geq \ell \geq m \quad \overset{\mathcal{D}_1}{\Gamma_1 \vdash y : D_\ell^*} \quad \overset{\mathcal{D}_2}{\Gamma_2, y : D_\ell \vdash x : A_m}}{\Gamma_1,\Gamma_2 \vdash x : A_m}\,\textsf{Snip}^1 \quad \overset{\mathcal{E}}{\Gamma_3, x : A_m \vdash z : C_k}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash z : C_k}\,\textsf{cut}$$

Switching the order of the snip and the cut, we get the following proof:

$$\dfrac{\Gamma_1 \geq \ell \geq k \quad \overset{\mathcal{D}_1}{\Gamma_1 \vdash y : D_\ell^*} \quad \dfrac{\Gamma_2, y : D_\ell \geq m \geq k \quad \overset{\mathcal{D}_2}{\Gamma_2, y : D_\ell \vdash x : A_m} \quad \overset{\mathcal{E}}{\Gamma_3, x : A_m \vdash z : C_k}}{\Gamma_2,\Gamma_3, y : D_\ell \vdash z : C_k}\,\text{i.h.}(C_k,\mathcal{D}_2,\mathcal{E})}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash z : C_k}\,\textsf{Snip}^1$$

By the inductive hypothesis, any formula that is eligible in $\mathcal{E}$ (or in $\mathcal{D}_2$) is either eligible in the second premise of the snip, or is a strict subformula of A_m. In the former case, it remains eligible in $\mathcal{F}$, and in the latter, it need not be eligible. Formulae which are eligible in $\mathcal{D}$ must, by definition, be eligible in one of $\mathcal{D}_1$ and $\mathcal{D}_2$, and if they are eligible in $\mathcal{D}_1$, then they are also eligible in $\mathcal{F}$, and so the eligibility condition is again satisfied.

Now, if $\mathcal{E}$ ends with a $\textsf{Snip}^1$, there are two subcases, depending on which branch of the snip $x : A_m$ goes to. We show one of these cases (where $x : A_m$ is used in $\mathcal{E}_1$) below:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : A_m} \quad \dfrac{\Gamma_2 \geq \ell \geq k \quad \overset{\mathcal{E}_1}{\Gamma_2, x : A_m \vdash y : D_\ell^*} \quad \overset{\mathcal{E}_2}{\Gamma_3, y : D_\ell \vdash z : C_k}}{\Gamma_2,\Gamma_3, x : A_m \vdash z : C_k}\,\textsf{Snip}^1}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash z : C_k}\,\textsf{cut}$$

By applying the inductive hypothesis to $(A_m, \mathcal{D}, \mathcal{E}_1)$ (using well-formedness of $\mathcal{E}_1$ to get the side condition that $m \geq \ell$), we get a proof $\mathcal{F}_1$ that $\Gamma_1, \Gamma_2 \vdash y : D_\ell$. Critically, the inductive hypothesis also gives us that either $y : D_\ell$ is eligible in $\mathcal{F}_1$, or D_ℓ is a strict subformula of A_m. In the former case, we can construct $\mathcal{F}$ as a $\textsf{Snip}^1$ between $\mathcal{F}_1$ and $\mathcal{E}_2$, since $y : D_\ell$ is eligible:

$$\dfrac{\Gamma_1,\Gamma_2 \geq \ell \geq k \quad \overset{\mathcal{F}_1}{\Gamma_1,\Gamma_2 \vdash y : D_\ell^*} \quad \overset{\mathcal{E}_2}{\Gamma_3, y : D_\ell \vdash z : C_k}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash z : C_k}\,\textsf{Snip}^1$$

Here, we use well-formedness of $\mathcal{E}_1$ again to get that $m \geq \ell$, and so (combined with $\Gamma_1 \geq m$ and $\Gamma_2 \geq \ell \geq k$), we get the side condition $\Gamma_1, \Gamma_2 \geq \ell \geq k$.

If $y : D_\ell$ is not eligible in $\mathcal{F}_1$, then D_ℓ must be a strict subformula of A_m, and so we may apply the inductive hypothesis a second time, now with the smaller formula D_ℓ, despite that $\mathcal{F}_1$ is a larger proof than $\mathcal{D}$. This gives us the following proof:

$$\dfrac{\Gamma_1,\Gamma_2 \geq \ell \geq k \quad \overset{\mathcal{F}_1}{\Gamma_1,\Gamma_2 \vdash y : D_\ell} \quad \overset{\mathcal{E}_2}{\Gamma_3, y : D_\ell \vdash z : C_k}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash z : C_k}\,\text{i.h.}(D_\ell,\mathcal{F}_1,\mathcal{E}_2)$$

In both cases, the usual reasoning applies to show that any formula eligible in $\mathcal{D}$ or $\mathcal{E}_1$ is eligible in $\mathcal{F}_1$ or is a strict subformula of A_m, and thus that the same applies for $\mathcal{F}$. Likewise, any formula eligible in $\mathcal{E}_2$ is either eligible in $\mathcal{F}$ or (in the case where D_ℓ is a strict subformula of A_m) is a strict subformula of D_ℓ, and therefore also of A_m.

The case where $x : A_m$ is used in $\mathcal{E}_2$, rather than $\mathcal{E}_1$, is similar, using the inductive hypothesis to combine $\mathcal{D}$ with $\mathcal{E}_2$, and then snipping $\mathcal{E}_1$ with the resulting proof. Since $\mathcal{E}_1$ is not affected by the inductive hypothesis, $y : \mathcal{D}_\ell$ is guaranteed to be eligible, and so this snip is always possible.

In the case where $\mathcal{D}$ ends with a Snip^2, we have the following proof to reduce:

$$\dfrac{\Gamma_1, \Gamma_2 \geq m \geq k \qquad \dfrac{\Gamma_1 \geq \ell \geq m \quad \overset{\mathcal{D}_1}{\Gamma_1 \vdash y : D_\ell} \quad \overset{\mathcal{D}_2}{\Gamma_2, y : D_\ell^* \vdash x : A_m}}{\Gamma_1, \Gamma_2 \vdash x : A_m}\,\mathsf{Snip}^2 \qquad \overset{\mathcal{E}}{\Gamma_3, x : A_m \vdash z : C_k}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash z : C_k}\,\mathsf{cut}$$

By applying the inductive hypothesis to $(A_m, \mathcal{D}_2, \mathcal{E})$, we get a proof $\mathcal{F}_2$ of $\Gamma_2, \Gamma_3, y : D_\ell \vdash z : C_k$. If $y : D_\ell$ remains eligible in $\mathcal{F}_2$, we can use a snip to construct the following proof:

$$\dfrac{\Gamma_1 \geq \ell \geq k \quad \overset{\mathcal{D}_1}{\Gamma_1 \vdash y : D_\ell} \quad \overset{\mathcal{F}_2}{\Gamma_2, \Gamma_3, y : D_\ell^* \vdash z : C_k}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash z : C_k}\,\mathsf{Snip}^2$$

Otherwise, D_ℓ must be a strict subformula of A_m, and so we may apply the inductive hypothesis a second time, giving us the following:

$$\dfrac{\Gamma_1 \geq \ell \geq k \quad \overset{\mathcal{D}_1}{\Gamma_1 \vdash y : D_\ell} \quad \overset{\mathcal{F}_2}{\Gamma_2, \Gamma_3, y : D_\ell \vdash z : C_k}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash z : C_k}\,\text{i.h.}(D_\ell, \mathcal{D}_1, \mathcal{F}_2)$$

The eligibility condition in this case follows from similar reasoning to the previous case.

Finally, consider the cases where $\mathcal{E}$ ends with a Snip^2. As when $\mathcal{E}$ ends with Snip^1, there are two similar cases, depending on which branch of the snip $x : A_m$ is used in. We show here the case where $x : A_m$ is used in $\mathcal{E}_1$:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \overset{\mathcal{D}}{\Gamma_1 \vdash x : A_m} \qquad \dfrac{\Gamma_2, x : A_m \geq \ell \geq k \quad \overset{\mathcal{E}_1}{\Gamma_2, x : A_m \vdash y : D_\ell} \quad \overset{\mathcal{E}_2}{\Gamma_3, y : D_\ell^* \vdash z : C_k}}{\Gamma_2, \Gamma_3, x : A_m \vdash z : C_k}\,\mathsf{Snip}^2}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash z : C_k}\,\mathsf{cut}$$

As in the previous cases, we begin by applying the inductive hypothesis to $(A_m, \mathcal{D}, \mathcal{E}_1)$ to give $\mathcal{F}_1$ a proof of $\Gamma_1, \Gamma_2 \vdash y : D_\ell$. Since $y : D_\ell$ is eligible in $\mathcal{E}_2$, we can use a Snip^2 to combine $\mathcal{F}_1$ with $\mathcal{E}_2$, giving the desired proof:

$$\dfrac{\Gamma_1, \Gamma_2 \geq \ell \geq k \quad \overset{\mathcal{F}_1}{\Gamma_1, \Gamma_2 \vdash y : D_\ell} \quad \overset{\mathcal{E}_2}{\Gamma_3, y : D_\ell^* \vdash z : C_k}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash z : C_k}\,\mathsf{Snip}^2$$

The case where $x : A_m$ is used in $\mathcal{E}_2$ is similar to the earlier case where $\mathcal{E}$ ends with a Snip^1 and $x : A_m$ is used in $\mathcal{E}_1$ — the inductive hypothesis gives us two cases for the eligibility of $y : D_\ell$, and allows us to either use a snip or to use the inductive hypothesis a second time to construct the desired proof.

Principal Cases Finally, we examine some of the principal cases, where the cut formula is principal in both $\mathcal{D}$ and $\mathcal{E}$. In particular, we will look at the cases for shifts, as they are the main feature distinguishing adjoint logic from other logics, and we will take $\multimap$ as an example of a more usual connective.

First, we consider the case where $\mathcal{D}$ ends in $\multimap R$ and $\mathcal{E}$ in $\multimap L$ (with the principal formula in both cases being the cut formula), giving us the following proof to reduce:

$$\dfrac{\Gamma_1 \geq m \geq m \quad \dfrac{\begin{matrix}\mathcal{D}_1\\ \Gamma_1, v : A_m \vdash w : B_m\end{matrix}}{\Gamma_1 \vdash x : A_m \multimap B_m}\multimap R \quad \dfrac{}{y : A_m, x : A_m \multimap B_m \vdash z : B_m}\multimap L^0}{\Gamma_1, y : A_m \vdash z : B_m}\,\mathsf{cut}$$

From this, we construct the following proof via substitution:

$$\begin{matrix}\mathcal{D}_1[y/v, z/w]\\ \Gamma_1, y : A_m \vdash z : B_m\end{matrix}$$

Any formula in Γ_1 which is eligible in $\mathcal{D}$ is also eligible here (via $\mathcal{D}_1$), and, while $y : A_m$ and $z : B_m$ are both eligible in $\mathcal{E}$, they are strict subformulae of the cut formula $A_m \multimap B_m$, and so need not be eligible in $\mathcal{F}$ (although they may be, depending on $\mathcal{D}_1$).

Now, we examine the shift cases, first that for $\downarrow$:

$$\dfrac{y : A_\ell \geq m \geq k \quad \dfrac{}{y : A_\ell \vdash x : {\downarrow}^\ell_m A_\ell}{\downarrow}R^0 \quad \dfrac{\begin{matrix}\mathcal{E}_1\\ \Gamma_2, w : A_\ell \vdash z : C_k\end{matrix}}{\Gamma_2, x : {\downarrow}^\ell_m A_\ell \vdash z : C_k}{\downarrow}L}{\Gamma_2, y : A_\ell \vdash z : C_k}\,\mathsf{cut}$$

As in the previous case, a substitution gives us the desired proof:

$$\begin{matrix}\mathcal{E}_1[y/w]\\ \Gamma_2, y : A_\ell \vdash z : C_k\end{matrix}$$

Any formula in $\Gamma_2, z : C_k$ which is eligible in $\mathcal{E}$ is also eligible in $\mathcal{E}_1$, and so also in $\mathcal{F}$, and while $y : A_\ell$ was eligible in $\mathcal{D}$ but may not be in $\mathcal{F}$, it is (as in the previous case) a strict subformula of the cut formula ${\downarrow}^\ell_m A_\ell$, and so need not be eligible in $\mathcal{F}$.

Finally, we examine the principal case for $\uparrow$:

$$\dfrac{\Gamma_1 \geq m \geq k \quad \dfrac{\begin{matrix}\mathcal{D}_1\\ \Gamma_1 \vdash y : A_k\end{matrix}}{\Gamma_1 \vdash x : {\uparrow}^m_k A_k}{\uparrow}R \quad \dfrac{}{x : {\uparrow}^m_k A_k \vdash z : A_k}{\downarrow}L^0}{\Gamma_1 \vdash z : A_k}\,\mathsf{cut}$$

This proof reduces (again via substitution) to the following:

$$\begin{matrix}\mathcal{D}_1[z/y]\\ \Gamma_1 \vdash z : A_k\end{matrix}$$

Here, any formula (besides $x : {\uparrow}^m_k A_k$, which is not eligible in $\mathcal{D}$ to begin with) which is eligible in $\mathcal{D}$ is also eligible in $\mathcal{D}_1$, and so is again eligible in $\mathcal{F}$. The formula $z : A_k$ is eligible in $\mathcal{E}$, and while it may not be eligible in $\mathcal{F}$, it is a strict subformula of the cut formula $\uparrow_k^m A_k$, and so our eligibility property is still satisfied.

The remaining principal cases are similar to one of the above examples, with positive connectives resembling $\downarrow$ and negative connectives resembling $\uparrow$. □

Cut elimination for SAX then follows from admissibility of cut in the usual way.

Theorem 15 (Cut elimination for SAX, directly)**.** *Suppose* $\Gamma \vdash A_m$ *in SAX. Then, there is a proof of* $\Gamma \vdash A_m$ *in SAX for which all cuts are snips.*

Proof. This proof is by induction on the derivation of $\Gamma \vdash A_m$, using admissibility of cut to remove each cut as we come across it. □

Chapter 4

Languages

Having developed the proof theory of adjoint logic, we now seek to apply it as the semantics of a programming language (or family of such languages). We will largely follow the usual proofs-as-programs approach, viewing propositions of the logic as types, proofs (or proof terms) as programs, and extracting computation steps from steps of proof normalization (or cut reduction, in the sequent calculus). We will see that several different languages, each with useful properties, can be extracted from different presentations of adjoint logic, or from different interpretations of proofs in adjoint logic. By starting from adjoint logic, with its uniform handling of modes with different structural properties, we can also expect to develop resulting programming languages with similarly uniform support for programming with mixtures of structural properties.

In some sense, it is straightforward to build a type system and programming language from a logic, by reinterpreting the propositions as types, writing down some proof terms, and providing an operational semantics for the terms, often based on some form of proof reduction. However, there are a variety of choices to be made here. The logic on its own does not fully determine a programming language — in the case of intuitionistic logic, for instance, a natural deduction presentation serves as the basis for the lambda calculus, while a Hilbert-style deduction system yields a combinator calculus. Moreover, some of the finer details of the presentation, such as whether we choose to make structural rules explicit or implicit, will affect the resulting language — at minimum, if structural rules are made explicit, they must have some operational interpretation, which is unnecessary when they are implicit. Finally, even once we have settled on a set of logical rules which we would like to develop further into a programming language, the choice of reduction strategy and the choice of how to interpret computation steps will affect the resulting programming language.

As a starting point for our development of adjoint programming languages, we look to prior work in a similar space. The work of Caires and Pfenning on the session-typed π-calculus $\pi\mathsf{DILL}$ [12, 14] is doubly relevant here — firstly, because it provides an example of how to interpret a (linear) sequent calculus as the basis for a concurrent programming language, and secondly, because, with its use of the exponential !, it is a first example of a "mixed-mode" language, in which some parts of the language behave linearly, while others do not. In $\pi\mathsf{DILL}$, programs consist of a collection of concurrently running processes, which communicate with one another via (synchronous) message-passing along communication channels shared between processes, where the communications that can occur across a given channel are governed by *session types*. [42, 43] In the purely linear fragment (i.e., without the exponential !), each channel is shared between exactly two processes, one of which *provides* that channel (or provides a service along that channel), while the other *uses* that channel, or is a *client* of it, analogous to how, for instance, a function in a functional language has one definition but may be called in many different places. Note that this provider/client distinction says nothing about the direction that messages are sent — indeed, messages are often sent in both directions over the same channel at different times. By using the exponential !, πDILL also supports channels shared between more than two processes, where one provider provides a (shared) service to multiple clients. However, this service is only loosely shared, with the only operation possible on it being to create a private copy of the service, which can then be interacted with as usual for a one-client purely linear service. This sharing-via-copying approach enables modelling of replicable services, but not of some other interesting patterns of concurrent communication, such as *multicast*, where a single message is sent to multiple recipients, or *cancellation*, where the (a) client of a channel, upon deciding it no longer has need of that channel, is able to close it unilaterally, without first finishing the protocol specified by the channel type.

SILL [77, 104] builds on πDILL, and provides a different approach for combining structural and linear computations, where a structural functional language is augmented with a contextual monad within which concurrent, linearly-typed (including the exponential !) computations can be built up. This split into a separate language per mode, each of which is fully-featured, allows for programs to be written with components living entirely in the layer to which they are best-suited, without needing to go through extra encoding steps (e.g., to represent a structural function type $A \supset B$ as $!(!A \multimap B)$). Despite this, however, the concurrent layer of SILL is still much the same as πDILL, and as such does not model multicast or cancellation of channels, and the non-uniformity of the approach, with a fully distinct language at each mode, makes it not immediately clear how it can be generalized, for instance to also include an affine layer.

In this chapter, we will develop two examples of languages based on adjoint logic, although we stress that these are by no means exhaustive — many other choices can be made in the logic-to-language development process, and other such languages may well be of interest. We begin by giving a brief overview of *process calculi*, of which our languages are examples (Section 4.1). Then, as our two languages have some substantial overlap in their syntax and some of their theory, we present these shared elements (Section 4.2), before moving on to the languages themselves. Our first language (Section 4.3) is a message-passing concurrent language, based on the ideas of πDILL for interpreting sequent calculus proofs as programs. However, rather than a standard sequent calculus, we work with a semi-axiomatic[1] presentation of adjoint logic, which, as we will see, allows us to work naturally with asynchronous communication. Because of the uniformity of adjoint logic as a framework, the resulting language is similarly uniform, able to use the same syntax (and even much of the same semantics) across different modes. Moreover, the shift to asynchronous communication and the semi-axiomatic sequent calculus enables us to model multicast and cancellation, even for channels along which messages have already been sent (but not yet received). Another benefit of working in an asynchronous setting is that it enables us to shift perspective, and develop our second example language (Section 4.4), in which communication occurs via access to shared memory cells, rather than via messages. We can

[1] See Chapter 3 for details

then take advantage of the different perspective of shared memory, where communication occurs entirely in one direction, to give a *sequential* semantics to the language, which we then further refine to allow for mixed sequential-concurrent programs. This then lets us interpret the original shared-memory semantics as a form of (substructural) futures. Various restrictions on where concurrent computation is allowed enable us to also model other schemes for mixed concurrent-sequential computation. Notably, the fact that double shifts $\uparrow\downarrow$ between a given pair of modes form a monad means that we can model a concurrency monad in the style of SILL [77, 104] by taking two modes, the higher of which only allows sequential computation and the lower of which allows concurrent computation. The shift upwards from concurrent to sequential then embeds concurrent computation safely into an otherwise sequential (although, in our setting, not functional) language, giving a similar effect to the concurrency monad of SILL.

In both languages that we present, the ability to work with modes with different structural properties allows us to model a wider range of features. For instance, in a message-passing setting, working with a mode that admits contraction enables us to model multicast, while modes that admit weakening provide support for cancelling channels. In the context of shared memory, a linear mode can give some guarantees relating to garbage collection, as a linear cell must be read from (and thus deallocated) by some thread, while modes which do not admit weakening can be used for strictness checking. More notable, however, is the fact that adjoint logic provides a framework where all of these features can coexist uniformly in the same language, as we are not restricted to only a single mode, and to model behavior that depends on the relations between multiple modes. Independence between modes captures several related concepts in a programming setting, even with modes which have the same structural properties. For instance, a preorder of labels in the context of information flow security can be thought of as an (inverted) preorder of modes — information can flow from low security to high security programs, but not vice versa, exactly a statement of independence, with a high-security mode being strictly less than a low-security mode in the preorder. Several other possibilities include ghost or proof-irrelevant data, as represented by a mode below the main mode of interest, upon which this main mode therefore cannot depend. This also captures a type-driven concept of dead code elimination, as explored in [74], for instance. Similarly, higher modes than the primary mode of interest may be used for staged computation, with an up-shift serving as quotation, and a down-shift as antiquotation. For a final, more concrete example, which we explore in Section 4.4.2, we can use modes with different structural properties to model a language with both linear and non-linear futures. Linear futures have been shown to increase efficiency, even asymptotically, in some cases [10], but in practice, it is often useful for some futures to be non-linear, so that their results may be reused. These examples, while by no means exhaustive, illustrate the expressive power of programming languages based on adjoint logic. Languages which are defined in a mode-dependent way provide an interesting avenue for future work, and may be used to model even more complex behavior, at the expense of taking more work to define.

4.1 Process Calculi

The two adjoint languages that we will develop are both forms of *process calculi*, systems for modelling concurrent computation, with a running program represented by a collection of running *processes*, which may communicate with one another as part of the computation. We will give here a brief overview of the process calculus literature, focusing on the systems most similar to ours, and describe how the adjoint languages fit in. The ideas of process calculi were initially developed in parallel[2] by Hoare [41] and Milner [62]. Milner's calculus CCS is a very minimal system, focused on capturing all psosible concurrency behavior with as simple as possible a set of primitives, and therefore, for instance, does not include sequential composition of processes (as this can be derived from parallel composition and appropriate use of synchronization), and restricts communication between processes to a given set of actions or events. Hoare's CSP provides a broader range of features, including sequential composition and support for communicating more complex values between processes.

Each of these early calculi has given rise to many successors, of which the most relevant family here are forms of π-calculus [63], which extends the ideas of CCS with the treatment of communication channels as first-class data, so that, for instance, a process may send a channel to another process. The use of channels allows for modeling concurrent systems where not all processes are in communication with one another (unlike the original forms of both CCS and CSP, where communication was global). By allowing channels to be sent as part of messages, the π-calculus also enables modeling systems whose network topology changes dynamically over the course of a computation, as processes gain or lose links between each other.

Much like the initial forms of the λ-calculus, this early π-calculus was untyped, and all communication was synchronous. Various type systems have been proposed and worked with for the π-calculus or fragments thereof (See, for example, [12, 14, 38, 44, 46, 51, 94]). $\pi\mathsf{DILL}$ [12, 14], mentioned in the introduction to this chapter as a starting point for our adjoint languages, is such a type system, assigning session types based on propositions of linear logic to π-calculus. Another example of such a language comes from Qian et al. [87], which makes use of a different approach, in terms of so-called *coexponentials*, allowing for modeling of a client-server architecture, where a single server interacts with a pool of multiple clients, encompassing multicast communication as a special case. Another approach for mixing linearity with non-linearity in the context of process calculi is that of manifest sharing [4], which is also closely related to the work presented here, making use of a similar mode stratification, restricted to two modes, one linear and one structural. In a similar vein, systems for modeling shared state such as that of Rocha and Caires [93] have also been used to broaden the range of possible communication behavior with some limited nonlinearity. Our adjoint type systems are similar in some respects to each of these, as all share similar roots in linear logic, but it is the non-linear portions that lead to most of the differences, as each system handles the concept of shared channels differently.

One drawback of the π-calculus, which is apparent in some of this prior and concurrent work, and also affects us, is that its notation is, in a sense, too expressive. Once we begin to impose type systems onto the π-calculus, we rapidly find that it is possible to write syntactically well-formed process terms that are not well-typed, and so, as in several of these other session-typed systems, we will diverge from the syntax of the π-calculus.[3] In particular, while the π-calculus allows

[2] Fittingly for the subject matter

[3] A side benefit of this is that, without ties to the π-calculus, we can adopt a "neutral" syntax that is not biased towards one or the other of the two languages that we develop, and this divergence also may have helped in the development of our shared-memory language, as we were less tied to the message-passing interpretation common for the π-calculus.

for arbitrary parallel composition of processes, we do not take this as a first-class operation, instead only allowing particular types of parallel composition in process terms. In the dynamic semantics, when we reason about broader collections of processes, we will again allow for more general parallel composition, but this should be thought of as separate processes running on the same system, rather than components of a single process term.

4.2 Common Features

The two languages that we will develop from adjoint logic share many concepts in their semantics. In this section, we present these shared components, including how we handle asynchronous communication logically, some syntax and syntactic operations that are common between the languages, and the overlap in their static semantics.

4.2.1 Asynchronous Communication

Thus far, we have looked at several different sequent calculus formulations of adjoint logic, and demonstrated that each has a cut elimination procedure. The work by Caires and Pfenning on $\pi\mathsf{DILL}$ provides a way to interpret sequent calculus proofs as (session-typed) concurrent programs, with computation being given by cut reduction steps. In this interpretation, a cut reduction is a form of *synchronous* communication: two processes, one trying to send a message and the other trying to receive that message along the same channel, can interact via cut reduction, but only when both processess are ready to communicate. Many real-world communication protocols involve *asynchronous* communication, where messages are sent regardless of whether the recipient is currently ready to receive the message, and we would like to also be able to model this behavior.

Indeed, in the non-linear setting where a channel may be shared between multiple clients (along with its one provider), it is not clear *a priori* what synchronous communication would mean. Do all of the clients of a channel need to be ready to receive a message in order for the provider to send that message? Dually, is it possible for one of these clients to send a message along the channel without the other clients doing so as well? By working with an asynchronous system of semantics, we will avoid these considerations, allowing one message to be sent to multiple recipients, each of whom can receive it separately, without needing to coordinate amongst themselves. This means that the computation step where a message is received can take place *locally*, without needing to consider what other clients for that message may exist.

In the context of $\pi\mathsf{DILL}$, DeYoung et al. developed an asynchronous interpretation of the same logic DILL [25]. This interpretation makes use of fresh *continuation channels* to replace the usual synchronous approach to session types, where a channel changes type upon a message being sent (and simultaneously received) along that channel. Instead of this evolution of types for a single channel, each subsequent message sent along “the same channel” uses a fresh channel name, thought of as the continuation of the previous channel. This avoids any possible conflicts resulting from multiple messages being sent along a channel x before any have been received, leading the recipient to be unsure which message it should read first. We will make use of this idea of continuation channels, although our approach to asynchrony is slightly different, being based on a *semi-axiomatic* presentation of logic, rather than a sequent calculus presentation. As explored in Chapter 3, the semi-axiomatic sequent calculus lends itself naturally to a system of semantics based on asynchronous communication, where, using a message-passing interpretation, messages are explicit objects, represented by axioms in proofs.

4.2.2 Variables, Symbols, and their Meanings

Before presenting semantics for our several languages, we need to be precise about the distinction between several similar parts of the language: variables, symbols, and the runtime concepts that they represent. Variables, which we will generally write x, y, z, using letters near the end of the alphabet, are a part of the syntax of programs, and are given meaning at runtime by substitution. Unlike in functional languages where expressions are substituted for variables, however, here, we substitute *symbols* for variables. A symbol, which we will write a, b, c, using letters near the beginning of the alphabet, is just an atom which represents a concept in the semantics, and which is opaque to the language using it. In particular, this means that we forbid testing symbols for equality within the languages we will define, just as in many common programming languages we cannot test variables for equality, only their contents. It is possible to make a different choice here, allowing for equality tests on symbols, just as, for instance, C allows equality checks on pointers. However, in such a system, programs can make more fine-grained distinctions about memory layout, such as distinguishing whether two copies of the same data are stored in the same or different locations. This may be desirable under some circumstances, such as for analyzing the memory footprint of programs, but we prefer here to take a higher-level, more abstract approach, where we cannot make these distinctions. Note that the concept represented by a symbol may be different in different systems of semantics for the same syntax: for instance, symbols may represent addresses in memory, or they may represent channels for communication between processes. The details of what a symbol represents will become relevant as we begin to look at specific languages, but we are able to develop much of the syntax for our languages in terms of symbols, without needing to make reference to the interpretation of those symbols. This allows us to unify large portions of the presentation of these two languages, and also illustrates the ease (in this semi-axiomatic setting) of recasting message-passing as shared memory and vice versa.

4.2.3 Processes, Values, and Continuations

The three main objects that will appear (albeit with different meanings) throughout the various systems of semantics we will present are processes, values, and continuations. Processes or process terms represent the state of a concurrent program (or portion of a program) as it executes, analogous to *expressions* in functional programming. Values and continuations represent two different kinds of data, and can both appear in process terms, serving a similar role to functional values. Values are small (fixed-size) pieces of data, while continuations are pieces of data that contain an entire process term, analogous to a closure. Comparing to the functional setting, one example of a value might be a pair of symbols $\langle a, b\rangle$, while a function abstraction $\lambda x.\ e$ would be a continuation. We will see more clearly the syntactic distinction between these types of object shortly.

Formally, processes, values, and continuations have grammar given in table 4.1, where alpha-equivalent terms are identified, as usual.

Example 6 (Some basic program syntax examples)**.** *With the syntax from Table 4.1, we can write and examine some example programs, for which we will give intuition using a message-passing interpretation, where symbols represent channels, and communication along a channel consists of sending and receiving values V as messages.*

Our first simple program receives a message containing a pair of symbols along a channel z, and then outputs the reverse of that pair along the channel w.

$$\mathsf{case}\ z\ (\langle x, y\rangle \Rightarrow w.\langle y, x\rangle)$$

Once we have typing rules for message-passing, we can see that this will be a proof term for the sequent $z : A \otimes B \vdash w : B \otimes A$, expressing the commutativity of $\otimes$.

For a second example, we consider a process term for currying — given typing rules, we will see that the following is a process term for the sequent $z : (A \otimes B) \multimap C \vdash w_1 : A \multimap (B \multimap C)$.

$$\mathsf{case}\ w_1\ (\langle x, w_2\rangle \Rightarrow \mathsf{case}\ w_2\ (\langle y, w_3\rangle \Rightarrow p \leftarrow (p.\langle x, y\rangle)\ ;\ z.\langle p, w_3\rangle)))$$

Intuitively, this process receives a pair along w_1, consisting of an x of type A (the first argument to the function) and w_2 of type $B \multimap C$, along which it must implement the function. The portion $\mathsf{case}\ w_1\ (\langle x, w_2\rangle \Rightarrow \ldots)$ *captures this — the case construct receives a message from a channel, and binds the variables x and w_2 within the remainder of the term, using them to refer to the entries of the pair that it expects to receive. It then receives a second message, this time along w_2, consisting of the second function argument y and a channel w_3 along which the protocol C should be implemented. Analogously, this is implemented by the portion* $\mathsf{case}\ w_2\ (\langle y, w_3\rangle \Rightarrow \ldots)$ *of the process term. Now, to send the two arguments x and y to the function z, they must first be wrapped up into a pair p. The construct $p \leftarrow \ldots\ ;\ \ldots$ spawns a new channel p, which we intend to use for this pair, and a new process that will communicate along p. This new process, represented by the term $p.\langle x, y\rangle$, sends the pair $\langle x, y\rangle$ along this new channel p, terminating as it does so. Now, we have a channel p along which the pair of arguments x and y have been sent, and so we are able to provide this to the function z. The construct $z.\langle p, w_3\rangle$ similarly sends a pair of channels, this time p and w_3, to the function z, which will use the first channel p as its input (receiving the message $\langle x, y\rangle$ that we sent earlier), and provide the output of the function on channel w_3.*

In the languages that follow, we will interpret these objects slightly differently, especially the process terms, whose meaning is closely tied to the meaning of symbols and the details of communication. However, the core distinction of processes as terms that can be executed, values as observable data, and continuations as data containing encapsulated, paused processes will be used throughout.

One key operation that is used throughout these languages is that of passing a value to a continuation, producing a process. A continuation provides a process term (or a choice of several process terms) with some variables bound in the continuation, and a value can both select an appropriate process term and provide symbols to substitute for those values. Note that this operation is agnostic to the meaning of symbols.

Value V	Meaning
$\langle\rangle$	The terminal value, containing no information
$\ell(c)$	A symbol c, tagged with the label ℓ
$\langle a, b\rangle$	A pair of symbols a and b
$\mathsf{shift}(c_k)$	A symbol c_k at mode k, tagged with a shift

Continuation K	Meaning
$(\langle\rangle \Rightarrow P)$	Given a terminal value $\langle\rangle$, continue as P
$(j(x_j) \Rightarrow P_j)_{j\in J}$	Given the value $\ell(c)$ for some $\ell \in J$, continue as $P_\ell[c/x_j]$. This binds the variable x_j in P_j for each $j \in J$.
$(\langle x, y\rangle \Rightarrow P)$	Given the value $\langle a, b\rangle$, continue as $P[a/x, b/y]$. This binds the variables x and y in P.
$(\mathsf{shift}(x_k) \Rightarrow P)$	Given the value $\mathsf{shift}(c_k)$, continue as $P[c_k/x_k]$. This binds the variable x_k in P.

Process term P	Meaning
$c \leftarrow a$	Connect symbols a and c
$x \leftarrow P\ ;\ Q$	Allocate a new symbol with some fresh name a, spawn a process $P[a/x]$, and continue as $Q[a/x]$. These two processess may communicate using the newly allocated symbol a. Note that this construct binds the variable x in *both P and Q*, analogously to the π-calculus term $(\nu x)(P\|Q)$. This is also more restrictive than the π-calculus, in that this is our only form of parallel composition, rather than allowing arbitrary process terms to be composed.
$c.V$	Communicate value V to symbol c. This can be thought of as a "neutral" version of π-calculus terms of the form $c!V$ and $c?V$, which send and receive data across channels.
$\mathsf{case}\ c\ K$	Communicate continuation K to symbol c.
$a \leftarrow p\ b_1\ b_2\ \ldots\ b_n$	Call the named process p with arguments a and $b_1\ b_2\ \ldots b_n$ (see Section 4.2.6)

Table 4.1: Grammar for processes, values and continuations.
Here, x, y, and x_j range over variables, while $a, b, c, b_1, b_2, \ldots$ range over symbols. k ranges over modes, which (for a given instance of this grammar) are drawn from a fixed, preordered set $\mathcal{M}$. Strictly speaking, variables and symbols do not need to be labelled with modes, but we find it useful to indicate which mode is being shifted to/from in the constructs that deal with shifts, and so include it when relevant.

We write this operation as $V \triangleright K$, and define it formally as follows:

$$\begin{array}{lcl}
\langle\,\rangle \triangleright (\langle\,\rangle \Rightarrow P) & \triangleq & P \\
\ell(c) \triangleright (j(x_j) \Rightarrow P_j)_{j \in J} & \triangleq & P_\ell[c/x_\ell] \\
\langle a, b\rangle \triangleright (\langle x, y\rangle \Rightarrow P) & \triangleq & P[a/x, b/y] \\
\mathbf{shift}(c_k) \triangleright (\mathbf{shift}(x_k) \Rightarrow P) & \triangleq & P[c_k/x_k]
\end{array}$$

For other combinations of V and K, $V \triangleright K$ is left undefined — for example, we cannot pass a pair $\langle a, b\rangle$ to a continuation $(\langle\rangle \Rightarrow P)$ expecting the terminal value. However, this will suffice for defining the dynamic semantics of well-typed processes, where such mismatches are enforced to not occur by typing. In each case of the definition, we extract the encapsulated process out of the continuation K, and substitute symbols from the value V for the variables bound in K, producing a process term that can then be run. This operation will serve as the key computation step for communication in all of its forms.

4.2.4 Configurations and (Multi)set Rewriting

While we can express a wide range of programs using the process terms described in section 4.2.3, they are not sufficient on their own to describe the dynamic semantics of any of the programming languages we will examine. The issue is most obvious with the spawn construct $x \leftarrow P\ ;\ Q$ — a process evaluating this construct will become two processes, and is no longer so neatly represented by a single process term. [4] Instead, we will represent the state of a running program as a collection of *semantic objects*, representing some portion of the state (running processes, artifacts of the communication between processes, and so on). We call this collection (which, in general, is a multiset of semantic objects, potentially restricted to satisfy some conditions) a *process configuration*, or just a *configuration*, for short.

We will often work with a grammar of the following form for configurations:

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \cdot \mid \phi \mid !\phi \mid \mathcal{C}_1, \mathcal{C}_2$$

A configuration is either empty, a single *ephemeral* semantic object ϕ (of course, in specific cases, we can be more precise about what the objects are, and we may have more than one class of objects), a single *persistent* semantic object $!\phi$, or the join $\mathcal{C}_1, \mathcal{C}_2$ of two smaller configurations. We think of this join operation as being associative and commutative, so that the grammar defines a multiset, rather than a list or a tree. Note also that while this grammar allows us to build up arbitrary multisets of semantic objects, we may have some further constraints on which multisets are valid configurations. In particular, we will generally say that a configuration $\mathcal{C}$ has an *interface*, consisting of a set S of symbols which it *uses*, and another set T of symbols which it *provides*. We then restrict the join of configurations, so that $\mathcal{C}_1, \mathcal{C}_2$ can only be formed if the sets of symbols provided by $\mathcal{C}_1$ and $\mathcal{C}_2$ are disjoint, ensuring that we can always identify

[4] This is in contrast to the π-calculus, where process terms include the parallel composition of two separate processes $P \mid Q$. While $x \leftarrow P\ ;\ Q$ may be thought of as an analog to this, we prefer to treat this as a construct that spawns a new, separate process, rather than just expressing the composition of two already running processes. In this way, each process on its own executes sequentially, although it may spawn subprocesses with which it can interact, and which run concurrently with the main process.

uniquely the provider of a given symbol. No such restriction is needed on the sets of symbols *used* by the configurations, because, in general, a symbol may be used by several processes (although typing may restrict this further). We will leave the definitions of interfaces abstract for now, to be specialized when we specialize configurations and objects to the particular languages we will work with. Since this grammar contains every valid configuration even without these constraints, it may also at times be useful to prove results about all configurations by induction over this grammar — such results may also apply to some invalid configurations, but this is not a problem.

Now, to describe the dynamic semantics of a programming language whose states are configurations, we need to give a set of rules for what state transitions are allowable. We think of a configuration as representing a collection of concurrently running processes, along with some other data such as messages between processes or memory cells. By treating processes as distinct semantic objects, we are able to describe state transitions "locally" — that is, only modifying the few objects in the configuration that are directly involved in a given computation step, rather than needing to examine and modify the entire state. Multiset rewriting rules [16] provide us with a compact way of describing these sorts of local state changes. A multiset rewriting rule $\phi_1, \ldots, \phi_n \mapsto \psi_1, \ldots, \psi_m$ can only be applied to a configuration $\mathcal{C}$ which contains $\phi_1, \ldots, \phi_n$. It consumes these objects, and replaces them with $\psi_1, \ldots, \psi_m$, leaving the rest of the configuration unchanged. That is, this rule would transform a configuration $\mathcal{C}, \phi_1, \ldots, \phi_n$ into $\mathcal{C}, \psi_1, \ldots, \psi_m$. Some objects in a configuration may be *persistent*, marked with a ! (and we refer to the other objects, without a !, as *ephemeral*). These objects are not consumed by rule applications. For instance, a rule $!\phi \mapsto \psi$ could turn the configuration $\mathcal{C}, !\phi$ into $\mathcal{C}, !\phi, \psi$. Likewise, some rules may introduce new persistent objects. For instance, $\phi \mapsto !\psi$ will consume a ϕ and replace it with a persistent ψ.

4.2.5 Typing for Processes and Configurations

In each of our languages, we will take a sequent $A^1_{m_1}, \ldots, A^n_{m_n} \vdash B_k$, attach to each proposition a distinct variable x_i. [5] We then add a process term, in order to get the typing judgement $x_1 : A^1_{m_1}, \ldots, x_n : A^n_{m_n} \vdash P :: (z : B_k)$. In each case, this represents that the process P may *use* variables (or symbols) $x_1, \ldots, x_n$ for communication with other processes, with the protocol for each communication determined by the type $A^i_{m_i}$, and *provides* a variable or symbol z which other processes may use in turn. What exactly this looks like computationally depends on how we interpret the symbols that will replace variables at runtime, as well as what sort of communication steps can occur using those symbols.

We can then turn the rules of the logic (in a suitable presentation) into typing rules by applying this same transformation to each sequent and specifying what proof term/process term corresponds to each rule. The exact details of this will vary across the different languages, but the core design remains the same.

While typing for single processes is sufficient to restrict what programs can be written, in order to describe properties of running programs, including the key type-safety results of progress

[5] When examining the state of a running process, some or all of these variables may be already instantiated as symbols. Because this does not affect our typing rules, which are agnostic to whether variables or symbols are used, we do not syntactically distinguish this "variable-or-symbol" class from variables.

and preservation, we need to extend our notions of typing for process terms to typing for configurations consisting of many semantic objects. The key difference here is that while a process can only provide a single symbol, a configuration, as it consists of potentially many processes, may provide any number of symbols. The judgement for configuration typing will therefore have the form $\Gamma \vDash \mathcal{C} :: \Delta$, where Γ, Δ are contexts consisting of mutually distinct symbols along with their types. Note that here, we only allow symbols, not variables, because configurations are a *run-time* concept, and as such, any exposed variables in a configuration must be instantiated. A first intuition suggests that this judgement means that $\mathcal{C}$ may use symbols in Γ, and provides the symbols in Δ, although we will see as we develop typing rules that this intuition needs to be refined slightly. As with typing for individual processes, the details of how this judgement is defined will vary between the different languages we examine, particularly as the semantic objects that make up configurations will be different as well. However, we can already give some basic rules for the empty configuration and the join of two configurations, independent of the choice of semantic objects. When defining configurations, we thought of the join and empty as being the operation and unit of a commutative monoid. In the context of typed configurations, however, the typing rules will impose the further constraint that the provider of a symbol occurs before (i.e., to the left of) every user of that symbol. This ordering causes typed configurations to only form a non-commutative monoid, as we cannot commute the provider of a symbol a past any of the users of that symbol.

For the join of two configurations to be well-typed, we need each of the two initial configurations to be well-typed, and then we should be able to combine their types. A first attempt at a typing rule following this intuition might say that the join of $\mathcal{C}_1$ and $\mathcal{C}_2$ uses all symbols that either configuration uses, and provides all symbols that either provides:

$$\frac{\Gamma_1 \vDash \mathcal{C}_1 :: \Delta_1 \quad \Gamma_2 \vDash \mathcal{C}_2 :: \Delta_2}{\Gamma_1, \Gamma_2 \vDash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_1, \Delta_2}\ \mathsf{join?}$$

This rule has a key issue, however. In a substructural setting, it may allow for configurations to be well-typed even if they illegally reuse symbols. If some $(a : A_m)$ occurs in both Γ_1 and Γ_2, while it does appear twice in the resulting type, we may not want to allow both $\mathcal{C}_1$ and $\mathcal{C}_2$ to use a. To resolve this, we adjust the meaning of the typing judgement $\Gamma \vDash \mathcal{C} :: \Delta$ from our initial intuition — rather than $\mathcal{C}$ necessarily providing all of Δ, we instead allow some symbols from Γ to be "passed through" $\mathcal{C}$, so that Δ is a collection of symbols which are either provided by $\mathcal{C}$, or available to use in Γ, but not "used up" in $\mathcal{C}$. We will make this intuition more formal when we come to defining configuration typing for the individual languages we work with. In this abstract setting, however, this shift in perspective gives us a solution to the problem of avoiding illegal reuse. For $\mathcal{C}_1, \mathcal{C}_2$ to be well-typed, $\mathcal{C}_1$ and $\mathcal{C}_2$ must be compatible — that is, we must be able to find some context Γ_2 that $\mathcal{C}_1$ can provide, possibly with some additional symbols coming from the input Γ_1 to $\mathcal{C}_1$, and where Γ_2 is sufficient input to type $\mathcal{C}_2$. This gives us the following rule:

$$\frac{\Gamma_1 \vDash \mathcal{C}_1 :: \Gamma_2 \quad \Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3}{\Gamma_1 \vDash \mathcal{C}_1, \mathcal{C}_2 :: \Gamma_3}\ \mathsf{join}$$

This rule now ensures that $\mathcal{C}_2$ does not illegally reuse symbols already taken by $\mathcal{C}_1$, provided the typing derivation for $\mathcal{C}_1$ does not place symbols used by $\mathcal{C}_1$ into Γ_2.

In order to match this intuition for the empty configuration, we need to ensure that it can pass symbols through unchanged, giving the following rule:

$$\frac{}{\Gamma \vDash (\cdot) :: \Gamma}\ \textsf{empty}$$

Since we now have the rules for join and the empty configuration, we can prove some properties generically, allowing us to rearrange configurations without changing their type. A few of the results will, however, require some constraints on the typing rules for individual objects, which each of the languages we study will satisfy.

Theorem 16 (Associativity of Configuration Typing)**.** *Typing of configurations is associative. That is,* $\Gamma_1 \vDash (\mathcal{C}_1, \mathcal{C}_2), \mathcal{C}_3 :: \Gamma_4$ *if and only if* $\Gamma_1 \vDash \mathcal{C}_1, (\mathcal{C}_2, \mathcal{C}_3) :: \Gamma_4$.

Proof. Suppose that $\Gamma_1 \vDash (\mathcal{C}_1, \mathcal{C}_2), \mathcal{C}_3 :: \Gamma_4$ — the reverse direction is symmetric. Now, by inversion (presuming that the only additional rules we have are those for typing individual objects or indivisible collections of objects), we can find Γ_3 such that $\Gamma_1 \vDash (\mathcal{C}_1, \mathcal{C}_2) :: \Gamma_3$ and $\Gamma_3 \vDash \mathcal{C}_3 :: \Gamma_4$. Again applying inversion, we find Γ_2 with $\Gamma_1 \vDash \mathcal{C}_1 :: \Gamma_2$ and $\Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3$. Composing these in a different order, we can easily prove $\Gamma_1 \vDash \mathcal{C}_1, (\mathcal{C}_2, \mathcal{C}_3) :: \Gamma_4$. □

Associativity allows us to treat configurations as lists of objects for the purpose of typing, ignoring the fine details of how exactly portions of the list are appended together. We now observe that the typing rule for the empty configuration allows it to have many different types — one for each context Γ. We would like to generalize this to all configurations, giving us a theorem that allows us to pass arbitrary symbols through any configuration unchanged, not only the empty one. This theorem will constrain any additional rules we introduce for typing individual objects, but only a little, and we will note in the proof where we make assumptions about these rules.

Theorem 17 (Type Extension)**.** *Suppose that for each atomic object* ϕ*, if* $\Gamma \vDash \phi :: \Delta$ *and* Θ *shares no symbols with* Γ, Δ*, or* ϕ*, then* $\Gamma, \Theta \vDash \phi :: \Delta, \Theta$*. Suppose also that typing for atomic objects does not create fresh symbols — that is, if* $\Gamma \vDash \phi :: \Delta$*, then* Δ *contains only symbols which occur already either in* Γ *or in* ϕ*. Then, these properties can be extended to all configurations: If* $\Gamma \vDash \mathcal{C} :: \Delta$*, then, given* Θ *which shares no symbols with* Γ*,* Δ*, or any of the objects in* $\mathcal{C}$*,* $\Gamma, \Theta \vDash \mathcal{C} :: \Delta, \Theta$*, and additionally typing for configurations does not generate fresh symbols.*

Proof. By induction on the structure of $\mathcal{C}$.

If $\mathcal{C} = (\cdot)$, this is immediate from the typing rule for the empty configuration — Γ must be equal to Δ, and so $\Gamma, \Theta = \Delta, \Theta$ as well. Similarly, every symbol in $\Delta = \Gamma$ already occurs in Γ.

If $\mathcal{C}$ is a singleton ϕ, this follows by assumption.

If $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$, by inversion, we can find Ξ such that $\Gamma \vDash \mathcal{C}_1 :: \Xi$ and $\Xi \vDash \mathcal{C}_2 :: \Delta$. Applying the inductive hypothesis to each of these, we get that Ξ consists only of symbols from $\mathcal{C}_1$ and Γ, and Δ consists only of symbols from Ξ and $\mathcal{C}_2$, so Δ consists only of symbols from Γ and $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$. Also from the inductive hypothesis, we see that as long as Θ shares no symbols with Γ, Δ, Ξ, $\mathcal{C}_1$, or $\mathcal{C}_2$, we can extend the types of both $\mathcal{C}_1$ and $\mathcal{C}_2$, and apply the rule for join to get the desired result. Since Θ does not share symbols with Γ, Δ, and $\mathcal{C}$, it also does not share symbols with Ξ (as Ξ consists only of symbols from Γ and $\mathcal{C}_1$). □

4.2.6 Recursion

While we will already be able to work with some basic examples once we begin defining our systems of semantics, many interesting examples have a recursive nature. In this section, we will briefly present our handling of recursion, which, while basic, is sufficient to write more complex example programs. As is common for session types, we will work with *equirecursive types*, which we collect in a signature Σ, along with type definitions, process declarations (which specify the type of a named process), and process definitions (which give the behavior of a named process). While the motivation for this signature is to be able to write recursive types and processes, we note that it can also be used to name non-recursive types and processes, which can be useful in presenting examples succinctly.

We first consider type declarations $t = A_k$, which state that t is a name for the type A_k. In such a declaration, we require that A_k is *contractive*, [33] with t only allowed to occur in A_k underneath some type constructor. This rules out the definition $t = t$, and ensures that each such recursive type definition actually uniquely defines a particular type. From the perspective of functional languages, which often work with inductive types, given as least fixed points of arbitrary recursive definitions, the choice to restrict to contractive type definitions is unusual. However, in a message-passing setting, it is often natural to interpret recursive types coinductively, while in a shared-memory setting, we may find cause to interpret them either inductively or coinductively, and the restriction to contractive type definitions avoids any possible confusion between the two. Named process declarations $B^1_{m_1}, B^2_{m_2}, \ldots B^n_{m_n} \vdash p :: A_k$ are needed for typechecking named processes, and specify the types of the arguments needed to call p — $B^i_{m_i}$, in order, as inputs, and A_k as an output. Independence still applies here, as in the pure logic, and dictates that we should require $m_1, \ldots, m_n \geq k$ in order for this declaration to be well-formed. Process *definitions* then provide variables to this type declaration — we write $x \leftarrow p\ y_1\ y_2\ \ldots\ y_n$, with the y_i being the input arguments, of type $B^i_{m_i}$, respectively, x the output argument of type A_k, and P the body of the process definition. We will generally abbreviate these sequences of types and variables as $\overline{B}$ and $\overline{y}$, respectively, noting that $\overline{B}$ may be a sequence of types at multiple modes $\overline{m} = m_1, \ldots, m_n$.

With this, we can give the formal grammar of signatures:

$$\text{Signatures } \Sigma ::= \cdot \mid \Sigma, t = A_k \mid \Sigma, \overline{B_m} \vdash p :: A_k \mid \Sigma, x \leftarrow p\,\overline{y} = P$$

A signature may be empty, or may be another signature extended with a type definition, a process declaration, or a process definition.

Now, we briefly consider typing and computation for these named types and processes, noting that the details will vary depending on our choice of semantics. For a signature to be valid, we require that each process declaration $\overline{B_m} \vdash p :: A_k$ has a corresponding definition $x \leftarrow p\,\overline{y} = P$ such that $\Sigma\ ;\ \overline{y : B_m} \vdash P :: (x : A_k)$ (whatever typing rules we take for a given system). Likewise, each process definition should have a corresponding declaration. This ensures that all named processes can (in principle) be mutually recursive, as they have sufficient information to invoke all other named processes. To call a named process p, we provide it with symbols to replace its variables, using the syntax $a \leftarrow p\ \overline{b}$ (with which we augment our existing syntax for process terms). We type this process term with the following rules (designed to work with implicit structural rules, as in ADJ^I — explicit structural rules remove the need for two separate call_var rules, but since the languages we will work with are based on ADJ^I, this system fits better with them):

$$\frac{x \leftarrow p\,\overline{y} = P \in \Sigma \quad \overline{B_m} \vdash p :: A_k \in \Sigma \quad \Gamma \vdash \overline{b : B_m}}{\Sigma\ ;\ \Gamma \vdash a \leftarrow p\,\overline{b} :: (a : A_k)}\ \mathsf{call}$$

$$\frac{\Gamma, (b : B_m)^\alpha \vdash \Delta}{\Gamma, (b : B_m) \vdash \Delta, b : B_m}\ \mathsf{call\_var}^\alpha \qquad\qquad \frac{\mathsf{W} \in \sigma(\Gamma)}{\Gamma \vdash (\cdot)}\ \mathsf{call\_empty}$$

A process call is well-typed with respect to Σ and Γ if it has both a matching declaration and definition, and Γ provides the necessary input for the process. The $\mathsf{call\_var}_\alpha$ and call_empty rules define what it means for Γ to provide the input to p. As in ADJ^I, we tag the call_var rule with a variable α, which may be 1 if $\mathsf{C} \in \sigma(m)$, allowing for a symbol $b : B_m$ to be used as more than one argument to a process call if m admits contraction. Note that these typing rules are not dependent on typing for the underlying system of semantics, and so can be reused unchanged. However, the definition of validity for signatures does depend on the typing of the underlying system, as we require that each process declaration has a correspondingly typed process definition, using the typing rules of the particular language we are working in, and so a signature that is valid in one context may not necessarily be so in another.

If p is defined by $x \leftarrow p\ \overline{y} = P$, then the process call $a \leftarrow p\ \overline{b}$ should step to a process executing $P[a/x, \overline{b}/\overline{y}]$. The exact details of how this works may vary slightly in different systems of semantics, but this is the underlying intuition behind all of them.

While the rules for typing and evaluating process calls depend on Σ, no typing rule modifies Σ, and so we will generally assume that we are working with a fixed signature Σ, which we then omit from rules unless explicitly necessary. Likewise, we will generally not work directly with the call_var and call_empty rules.

Example 7 (Examples of recursive types and programs)**.** *Recursive type definitions look much the same here as in other languages. We show here some examples which are generally useful, beginning with the type of infinite bit streams, using the notation* $\oplus\{i : A_i\}$ *for an internal choice across an index set of illustrative names:*

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}\}$$

In this case, a bit stream consists of a bit, either b0 *or* b1*, followed by another bit stream. We can enable these streams to be finite by adding a third case, marking the end of the stream:*

$$\mathsf{may\_end\_bits} = \oplus\{\mathsf{b0} : \mathsf{may\_end\_bits}, \mathsf{b1} : \mathsf{may\_end\_bits}, \$: \mathbf{1}\}$$

These potentially finite bit streams look much the same as infinite bit streams — when reading/receiving data from one, it is possible to receive either a zero or one bit, followed by another (potentially finite) bit stream, or the token \$, indicating the end of the stream, which can then be closed via trivial communication at the unit type $\mathbf{1}$.

A type (or family of types) that we will regularly use in our examples is that of lists. Given a type A*, we can define* list_A*, the type of lists with elements from* A*, in roughly the usual way:*

$$\mathsf{list}_A = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : A \otimes \mathsf{list}_A\}$$

A list is either empty, and so communication can terminate via the unit type $\mathbf{1}$*, or contains a cons cell, containing an element of type* A *and the tail of the list.*

For a first, basic recursive process, we will examine a process that inverts a (potentially finite) bit stream.

```
$\mathsf{may\_end\_bits} \vdash \mathsf{flip} :: \mathsf{may\_end\_bits}$
$y \leftarrow \mathsf{flip}\ x = \mathsf{case}\ x$  ( $\mathsf{b0}(x') \Rightarrow y' \leftarrow (y' \leftarrow \mathsf{flip}\ x')$ ;
                                   $y.\mathsf{b1}(y')$
                       | $\mathsf{b1}(x') \Rightarrow y' \leftarrow (y' \leftarrow \mathsf{flip}\ x')$ ;
                                   $y.\mathsf{b0}(y')$
                       | $\$(u) \Rightarrow y.\$(u)$
                       )
```

We first attempt to read a bit from x*, and if we receive instead the end-of-stream symbol* $\$$*, we also close the stream* y*. When receiving an actual bit, we spawn a new process which flips the tail* x' *of the bit stream, communicating the result along* y'*, and then send the inverse bit of what we read, as well as this tail* y'*, along the channel* y*.*

A more complex program, which we will use as a running example, because it illustrates also the effect of working with different modes, is a standard map function. While mapping a function over a list, the function may be used any natural number of times, depending on the length of the list. As such, regardless of the mode of the type of elements in the list (which, in principle, may be linear, as map uses each element of the list exactly once), the function being mapped over them needs to be structural. To resolve this conflict, that the function must be structural, but must also match the mode of its argument, we will use a shift*, marking the function as reuseable, regardless of its base mode. We write the program below generically in the mode* k *of list elements, but using a fixed mode* U *with* $\sigma(\mathsf{U}) = \{W, C\}$ *for the function. In practice, this is a program schema, rather than a program, and needs to be separately instantiated for each mode* k *that we want to use it at, as we do not handle mode polymorphism.*

```
$\mathsf{list}_{A_k}, \uparrow_k^{\mathsf{U}} A_k \multimap B_k \vdash \mathit{map} :: \mathsf{list}_{B_k}$
$ys \leftarrow \mathit{map}\ xs\ f =$
$\mathsf{case}\ xs$  ( $\mathsf{nil}(u) \Rightarrow ys.\mathsf{nil}(u)$
         | $\mathsf{cons}(p) \Rightarrow \mathsf{case}\ p$  ( $\langle x, xs' \rangle \Rightarrow$
                         $ys' \leftarrow (ys' \leftarrow \mathsf{map}\ xs'\ f)$
                         $f' \leftarrow f.\mathsf{shift}(f')$ ;
                         $y \leftarrow f'.\langle x, y \rangle$ ;
                         $p' \leftarrow (p'.\langle y, ys' \rangle)$
                         $ys.\mathsf{cons}(p')$
                         )
         )
```

This process is largely familiar, but takes some additional low-level steps that are often not made explicit. We begin by matching on the input list xs*, and if it is empty, we output another empty list along* ys*. If* xs *is not empty, then we break down the contents* p *of its cons cell to get the first element* x *and the tail* xs' *of the list. We then spawn a new process to make a recursive call to construct* ys'*, the result of mapping* f *over* xs'*. In order to call* f*, we first need to extract a copy* f' *at mode* k *from the shifted function* f*. We can then create a new channel* y*, which we* *pass along with x to f', so that f' will run the function on x, and provide its output along y. Combining y and ys' into a pair p' takes another step (and another newly created channel for p'), after which we can construct ys by sending a cons cell containing p'.*

Of the extra steps in this process compared to a typical functional map, most consist of explicitly naming intermediate results of computation, such as the pair p of elements in a cons cell, and constructing/destructing these intermediate values in their own steps. The remaining step that is not of this form is $f' \leftarrow f.\mathsf{shift}(f')$, where we extract (a copy of) the underlying function f' from the shifted (and therefore replicable) function f. In the sections that follow, we will return to this example, seeing how it can be typed and run

4.3 Message-Passing Semantics

The first system we will examine interprets symbols as private channels between processes, along which messages can be passed. Processes can send messages along a channel, and those messages will be received by the process(es) at the other end of the channel. Under this interpretation, the messages being sent are exactly the values V, while continuations K represent processes waiting to receive messages. A message V can be sent along the channel a using the construct $a.V$. This message is then received by a process $\mathsf{case}\ a\ K$, which passes the received message V to the continuation K to decide how to proceed.

4.3.1 Static Semantics

With this model in mind for computation, we can clarify the meanings of process terms for this system of semantics, as described in table 4.2.

We can then present typing rules for our language, assigning process terms to a semi-axiomatic sequent calculus [26] presentation of ADJ^I,[6] as seen in fig. 4.1.

Example 8 (Typing of example processes)**.** *We now return to examine how some of the example processes we have looked at can be typed, beginning with non-recursive examples.*

[6] More discussion of the semi-axiomatic sequent calculus can be found in chapter 3.

Process term P	Meaning
$c \leftarrow a$	Forward messages between channels a and c.
$x \leftarrow P \ ; Q$	Allocate a new channel a, spawn a process $P[a/x]$, and continue as $Q[a/x]$. These two processess may communicate using the channel a.
$c.V$	Send message V along channel c.
$\mathsf{case}\, c\, K$	Receive a message V from channel c, then pass V to K.
$a \leftarrow p\ \overline{b}$	Call the named process p, which may communicate along channels a and $\overline{b}$.

Table 4.2: Meanings of process terms in a message-passing setting

$$\frac{(\Gamma_C, \Delta \geq m \geq r) \quad \Gamma_C, \Delta \vdash P :: (x : A_m) \quad \Gamma_C, \Delta', x : A_m \vdash Q :: (z : C_r)}{\Gamma_C, \Delta, \Delta' \vdash (x \leftarrow P \; ; Q) :: (z : C_r)} \; \mathsf{cut}$$

$$\frac{}{\Gamma_W, y : A_m \vdash x \leftarrow y :: (x : A_m)} \; \mathsf{id}$$

$$\frac{(i \in L)}{\Gamma_W, y : A_m^i \vdash x.i(y) :: (x : \oplus\{\ell : A_m^\ell\}_{\ell \in L})} \; \oplus R^0$$

$$\frac{\Gamma, (x : \oplus\{\ell : A_m^\ell\}_{\ell \in L})^\alpha, y : A_m^\ell \vdash Q_\ell :: (z : C_r) \quad \text{(for all } \ell \in L)}{\Gamma, x : \oplus\{\ell : A_m^\ell\}_{\ell \in L} \vdash \mathbf{case}\, x\, (\ell(y) \Rightarrow Q_\ell)_{\ell \in L} :: (z : C_r)} \; \oplus L_\alpha$$

$$\frac{\Gamma \vdash P_\ell :: (y : A_m^\ell) \quad \text{(for all } \ell \in L)}{\Gamma \vdash \mathbf{case}\, x\, (\ell(y) \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_m^\ell\}_{\ell \in L})} \; \& R$$

$$\frac{(i \in L)}{\Gamma_W, x : \&\{\ell : A_m^\ell\}_{\ell \in L} \vdash x.i(y) :: (y : A_m^i)} \; \& L^0$$

$$\frac{}{\cdot \vdash x.\langle\, \rangle :: (x : \mathbf{1}_m)} \; \mathbf{1}R^0 \qquad \frac{\Gamma, (x : \mathbf{1})^\alpha \vdash P :: (z : C_r)}{\Gamma, x : \mathbf{1} \vdash \mathbf{case}\, x\, (\langle\, \rangle \Rightarrow P) :: (z : C_r)} \; \mathbf{1}L_\alpha$$

$$\frac{\Gamma, w : A_m \vdash P :: (y : B_m)}{\Gamma \vdash \mathbf{case}\, x\, (\langle w, y \rangle \Rightarrow P) :: (x : A_m \multimap B_m)} \; \multimap R$$

$$\frac{}{\Gamma_W, w : A_m, x : A_m \multimap B_m \vdash x.\langle w, y \rangle :: (y : B_m)} \; \multimap L^0$$

$$\frac{}{\Gamma_W, w : A_m, y : B_m \vdash x.\langle w, y \rangle :: (x : A_m \otimes B_m)} \; \otimes R^0$$

$$\frac{\Gamma, (x : A_m \otimes B_m)^\alpha, w : A_m, y : B_m \vdash P :: (z : C_r)}{\Gamma, x : A_m \otimes B_m \vdash \mathbf{case}\, x\, (\langle w, y \rangle \Rightarrow P) :: (z : C_r)} \; \otimes L_\alpha$$

$$\frac{}{\Gamma_W, y : A_m \vdash x_k.\mathbf{shift}(y_m) :: (x : {\downarrow}_k^m A_m)} \; {\downarrow} R^0$$

$$\frac{\Gamma, (x : {\downarrow}_k^m A_m)^\alpha, y : A_m \vdash Q :: (z : C_r)}{\Gamma, x : {\downarrow}_k^m A_m \vdash \mathbf{case}\, x_k\, (\mathbf{shift}(y_m) \Rightarrow Q) :: (z :: C_r)} \; {\downarrow} L_\alpha$$

$$\frac{\Gamma \vdash P :: (y : A_k)}{\Gamma \vdash \mathbf{case}\, x_m\, (\mathbf{shift}(y_k) \Rightarrow P) :: (x : {\uparrow}_k^m A_k)} \; {\uparrow} R \qquad \frac{}{\Gamma_W, x : {\uparrow}_k^m A_k \vdash x_m.\mathbf{shift}(y_k) :: (y : A_k)} \; {\uparrow} L^0$$

Figure 4.1: Message-passing typing rules based on a semi-axiomatic presentation of ADJ^I. We label some rules with an index $\alpha \in \{0, 1\}$ to condense two rules, one which has $\alpha = 1$, requires $\mathsf{C} \in \sigma(m)$, and preserves the principal formula of the rule in the premise(s), and one which has $\alpha = 0$ and does not preserve the principal formula.

With the below typing derivation, we can see that indeed our process for reversing a pair has the expected type $A_m \otimes B_m \vdash B_m \otimes A_m$:

$$\dfrac{\dfrac{}{x : A_m, y : B_m \vdash w.\langle y, x\rangle :: w : B_m \otimes A_m}\;\otimes R^0}{z : A_m \otimes B_m \vdash \mathsf{case}\ z\ (\langle x, y\rangle \Rightarrow w.\langle y, x\rangle) :: w : B_m \otimes A_m}\;\otimes L_0$$

Likewise, currying can be assigned the expected type. In this derivation, we abbreviate the process terms, and omit the mode m (which is the same on all types) for space reasons. Similarly, the mode condition on the cut does not need to be checked, because all modes are m, and as no types are shared between the two branches of the cut, we also do not need to check that any portion of the context admits contraction.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : A, y : B \vdash p.\langle x, y\rangle :: p : A \otimes B}\;\otimes R^0 \quad \dfrac{}{z : (A \otimes B) \multimap C, p : A \otimes B \vdash z.\langle p, w_3\rangle :: w_3 : C}\;\multimap L^0}{z : (A \otimes B) \multimap C, x : A, y : B \vdash p \leftarrow \ldots ; \ldots :: w_3 : C}\;\mathsf{cut}}{z : (A \otimes B) \multimap C, x : A \vdash \mathsf{case}\ w_2\ (\langle y, w_3\rangle \Rightarrow \ldots) :: w_2 : B \multimap C}\;\multimap R}{z : (A \otimes B) \multimap C \vdash \mathsf{case}\ w_1\ (\langle x, w_2\rangle \Rightarrow \ldots) :: w_1 : A \multimap (B \multimap C)}\;\multimap R$$

For larger processes, such as our example of map, the fully written out typing derivation becomes impractically large to work with. Instead, after each line, we will write the current typing context, prefixing these comment lines with # and highlighting them for clarity.

```
$\mathsf{list}_{A_k}, {\uparrow}^{\mathsf{U}}_k A_k \multimap B_k \vdash \mathit{map} :: \mathsf{list}_{B_k}$
$ys \leftarrow \mathit{map}\ xs\ f =$
$\#\, xs : \mathsf{list}_{A_k}, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash ys : \mathsf{list}_{B_k}$
$\mathsf{case}\ xs\ \ (\ \ \mathsf{nil}(u) \Rightarrow$
$\#\, u : \mathbf{1}_k, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash ys : \mathsf{list}_{B_k}$
                $ys.\mathsf{nil}(u)$
$\#\, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash \cdot$
              $\mid\ \mathsf{cons}(p) \Rightarrow$
$\#\, p : A_k \otimes \mathsf{list}_{A_k}, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash ys : \mathsf{list}_{B_k}$
                $\mathsf{case}\ p\ \ (\ \ \langle x, xs'\rangle \Rightarrow$
$\#\, x : A_k, xs' : \mathsf{list}_{A_k}, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash ys : \mathsf{list}_{B_k}$
                              $ys' \leftarrow (ys' \leftarrow \mathsf{map}\ xs'\ f)$
$\#\, x : A_k, ys' : \mathsf{list}_{B_k}, f : {\uparrow}^{\mathsf{U}}_k(A_k \multimap B_k) \vdash ys : \mathsf{list}_{B_k}$
                              $f' \leftarrow f.\mathsf{shift}(f')\ ;$
$\#\, x : A_k, ys' : \mathsf{list}_{B_k}, f' : A_k \multimap B_k \vdash ys : \mathsf{list}_{B_k}$
                              $y \leftarrow f'.\langle x, y\rangle\ ;$
$\#\, y : B_k, ys' : \mathsf{list}_{B_k} \vdash ys : \mathsf{list}_{B_k}$
                              $p' \leftarrow (p'.\langle y, ys'\rangle)$
$\#\, p' : B_k \otimes \mathsf{list}_{B_k} \vdash ys : \mathsf{list}_{B_k}$
                              $ys.\mathsf{cons}(p')$
$\#\, \cdot \vdash \cdot$
                          $)$
              $)$
```

When moving from the first line to the second line, using the $\oplus L_0$ rule, we replace the list xs with the unit u contained in its nil. *After the third line, we have already implemented ys, and so nothing remains to prove. However, we still have f in our typing context, as it was not yet used, and so it is critical here that f is at a mode that admits weakening — otherwise, we could not type this case. Likewise, when moving from the first to the fourth line, also using $\oplus L_0$, but examining a different premise of the rule, we replace xs with the contents p of its* cons *cell. We then break down p further with the $\otimes L_0$ rule, replacing it with the head x and tail xs' of the original list xs. The remainder of the program is typed with a sequence of cuts. The left-hand branch of the first cut makes a recursive call to* map, *which is typed using the* call *rules of Section 4.2.6. Note that here, we use that f has a mode that admits contraction in order to use f in both branches of the cut. We also use that $\mathsf{U} \geq k$ in order for the newly created channel ys', which has mode k, to depend on f, which as mode U. Subsequently, we extract the underlying function f' from f, noting that as we have already duplicated f to be used in the recursive call, we no longer need it, and so this next step can be typed using $\uparrow L^0$. We can then apply f' to x, giving a result y, using the $\multimap L^0$ rule, construct a pair p' of y and ys' using the $\otimes R$ rule, and use this to implement ys, using the $\oplus R$ rule.*

4.3.2 Dynamic Semantics

To represent the dynamic semantics of this language, we will use three types of semantic objects:

- $\mathsf{proc}(P)$, representing a running process executing the process term P.
- $!^+_m\mathsf{msg}(a_m, V)$, representing a message V being sent on channel a_m. The $!^+_m$ denotes that this object is persistent in the configuration if $\mathsf{C} \in \sigma(m)$ and also the type of a_m is positive $(\otimes, \mathbf{1}, \oplus, \downarrow)$.
- $!^-_m\mathsf{srv}(a_m, K)$, representing a service K listening on channel a_m. Similarly to $!^+_m$, $!^-_m$ denotes that this object is persistent if $\mathsf{C} \in \sigma(m)$ and also the type of a_m is negative $(\multimap, \&, \uparrow)$.

Our configurations for this language then have the following grammar:

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \cdot \mid \mathsf{proc}(P) \mid !^+_m\mathsf{msg}(a_m, V) \mid !^-_m\mathsf{srv}(a_m, K) \mid \mathcal{C}_1, \mathcal{C}_2$$

We would also like to enforce statically that configurations are only valid when no two objects are attempting to provide the same channel. This is easily checked when typing configurations by rejecting types where the same channel occurs more than once in the type, but we can also specify the interface of configurations, and rely on the generic constraint that configurations can only be joined if their sets of provided symbols (here, channels) do not overlap. Since we are working in this chapter only with well-typed configurations, we will, for now, rely on typing to ensure our configurations are well-formed. Of course, a real implementation may find it useful to track information about provided and used channels more precisely, as it may simplify the process of type checking. As in the general case, we will treat the join $\mathcal{C}_1, \mathcal{C}_2$ of two configurations as a commutative and associative operation so that this grammar defines a (multi)set rather than a tree. Additionally, for clarity, when writing configurations, we will adopt the convention (consistent with the typing rule for the join of configurations) that the provider of a channel appears to the left of any clients of that channel.

$$
\begin{array}{ll}
\mathsf{proc}(x \leftarrow P \; ; \; Q) \mapsto \mathsf{proc}([a/x]P), \mathsf{proc}([a/x]Q) \; (a \text{ fresh}) & \textit{cut: allocate channel \& spawn} \\
\mathsf{proc}(d_m^+ \leftarrow c_m^+) \mapsto !_m^+\mathsf{msg}(d_m^+, c_m^+) & \textit{id: forward}^+ \\
\mathsf{proc}(d_m^- \leftarrow c_m^-) \mapsto !_m^-\mathsf{srv}(d_m^-, c_m^-) & \textit{id: forward}^- \\
\mathsf{proc}(c_m.V) \mapsto !_m^+\mathsf{msg}(c_m, V) & \textit{Positive right/negative left rules: send} \\
\mathsf{proc}(\mathbf{case}\, c_m\, K) \mapsto !_m^-\mathsf{srv}(c_m, K) & \textit{Negative right/positive left rules: listen} \\
!_m^+\mathsf{msg}(c_m, V), !_m^-\mathsf{srv}(c_m, K) \mapsto \mathsf{proc}(V \triangleright K) & \textit{Communication} \\
\mathsf{proc}(a \leftarrow p\; \overline{b}) \mapsto \mathsf{proc}(P[a/x, \overline{b}/\overline{y}]) \quad (x \leftarrow p\; \overline{y} = P \in \Sigma) & \textit{Process call}
\end{array}
$$

Figure 4.2: Reduction rules for message-passing

We can then present the dynamic semantics of this language as a collection of (multi)set rewriting rules, shown in in Figure 4.2.

In these rules, we have slightly modified the syntax of values and continuations discussed in section 4.2.3 in order to handle forwarding. To forward one channel to another, we either set up a service forwarding messages (if the type of the channel is negative), or send a message along the channel to tell the service provided along that channel of its new client (if the type of the channel is positive). This service or message will be persistent, enabling it to continue forwarding any future messages or services on the same channel as necessary. To implement this, we allow a single channel name c_m to be either a value or a continuation, and extend the operation $V \triangleright K$ with the following two cases:

$$
\begin{array}{rcl}
c_m \triangleright K & \triangleq & \mathsf{case}\; c_m\; K \\
V \triangleright c_m & \triangleq & c_m.V
\end{array}
$$

In either case, a forward to c_m interacts with either a value or a continuation to change the channel it is operating on to c_m.

Example 9 (Examples of evaluation of processes)**.** *We now examine how some sample processes evaluate, first with a short example that illustrates how forwarding works:*

In this example, we work with channel of positive type $\mathbf{1}_m$. One process sends a message along c_m, while another waits to receive a message on d_m, and a third process in the middle forwards to connect c_m and d_m. We highlight in red the semantic object(s) that are involved in taking each given step.

$$
\begin{array}{ll}
& \mathsf{proc}(c_m.\langle\rangle), \mathsf{proc}(d_m^+ \leftarrow c_m^+), \mathsf{proc}(\mathsf{case}\; d_m\; (\langle\rangle \Rightarrow P)) \\
\mapsto & !_m^+\mathsf{msg}(c_m, \langle\rangle), \mathsf{proc}(d_m^+ \leftarrow c_m^+), \mathsf{proc}(\mathsf{case}\; d_m\; (\langle\rangle \Rightarrow P)) \\
\mapsto & !_m\mathsf{msg}(c_m, \langle\rangle), \mathsf{proc}(d_m^+ \leftarrow c_m^+), \mathsf{srv}(d_m, \langle\rangle \Rightarrow P) \\
\mapsto & !_m\mathsf{msg}(c_m, \langle\rangle), !_m\mathsf{msg}(d_m^+, c_m^+), \mathsf{srv}(d_m, \langle\rangle \Rightarrow P) \\
\mapsto & !_m\mathsf{msg}(c_m, \langle\rangle), !_m\mathsf{msg}(d_m^+, c_m^+), \mathsf{proc}(\mathsf{case}\; c_m\; (\langle\rangle \Rightarrow P)) \\
\mapsto & !_m\mathsf{msg}(c_m, \langle\rangle), !_m\mathsf{msg}(d_m^+, c_m^+), \mathsf{srv}(c_m.\langle\rangle \Rightarrow P) \\
\mapsto & !_m\mathsf{msg}(c_m, \langle\rangle), !_m\mathsf{msg}(d_m^+, c_m^+), \mathsf{proc}(P)
\end{array}
$$

We can see in the fourth line of this example how the forwarding message interacts with a service listening on d_m to redirect it to listen on c_m (after one additional step from line 5 to line 6, for the newly running process to begin listening again). Note that the forwarding message is persistent if m admits contraction, so that in case d_m has multiple clients, they can each individually be forwarded to c_m.

We then move to look at a larger example, seeing how the flip *process shown in Section 4.2.6 executes to actually invert a (linear) bit stream. In the initial state of this configuration, we have the bit stream $01\$$, represented by a sequence of messages, each of which refers to the next, as well as the* flip *process. We abbreviate the body of the running process and its continuations as necessary.*

$$
\begin{array}{ll}
 & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{msg}(x_0, \mathsf{b0}(x_1)), \mathsf{proc}(y_0 \leftarrow \mathsf{flip}\ x_0) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{msg}(x_0, \mathsf{b0}(x_1)), \mathsf{proc}(\mathsf{case}\ x_0(\ldots)) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{msg}(x_0, \mathsf{b0}(x_1)), \mathsf{srv}(x_0, \ldots) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{proc}(y_1 \leftarrow (y_1 \leftarrow \mathsf{flip}\ x_1\ ;\ y_0.\mathsf{b1}(y_1))) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{proc}(y_1 \leftarrow \mathsf{flip}\ x_1), \mathsf{proc}(y_0.\mathsf{b1}(y_1))) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{msg}(x_1, \mathsf{b1}(x_2)), \mathsf{proc}(y_1 \leftarrow \mathsf{flip}\ x_1), \mathsf{msg}(y_0, \mathsf{b1}(y_1)) \\
 & \vdots \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{proc}(y_2 \leftarrow \mathsf{flip}\ x_2), \mathsf{msg}(y_1, \mathsf{b0}(y_2)), \mathsf{msg}(y_0, \mathsf{b1}(y_1)) \\
 & \vdots \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(x_2, \$(u)), \mathsf{srv}(x_2, \ldots), \mathsf{msg}(y_1, \mathsf{b0}(y_2)), \mathsf{msg}(y_0, \mathsf{b1}(y_1)) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{proc}(y_2.\$(u)), \mathsf{msg}(y_1, \mathsf{b0}(y_2)), \mathsf{msg}(y_0, \mathsf{b1}(y_1)) \\
\mapsto & \mathsf{msg}(u, \langle\rangle), \mathsf{msg}(y_2, \$(u)), \mathsf{msg}(y_1, \mathsf{b0}(y_2)), \mathsf{msg}(y_0, \mathsf{b1}(y_1))
\end{array}
$$

Much of this execution trace is unsurprising, and so we omit some repetitive steps after showing them in the first iteration. However, it is interesting to note that the message being sent along channel u is never actually received, instead being reused in implementing y_2. Eventually, a process reading this flipped bit stream may need to read from u in order to fully consume the bit stream (at least in the context of a linear or strict mode), but even then, if such a process has use for a message of type $\mathbf{1}$, it can again reuse u for this purpose.

To present and prove type safety and other results for this language, we need to give rules for typing configurations. In particular, we need to describe how each individual object is typed, which we can then combine with the generic rules for combining configurations from section 4.2.5. Moreover, as shown in that section, as long as our typing rules for individual objects satisfy type extension and don't generate fresh symbols, we are able to get associativity and type extension for the typing of full configurations, which will be invaluable in proving both progress and preservation.

The rules for typing these objects rely on typing for processes. Messages and services are treated as the processes which, in one step, produce them.

$$\frac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Gamma_1 \vdash P :: (c : A_m)}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash \mathsf{proc}(P) :: \Gamma_C, \Gamma_2, (c : A_m)}\ \mathsf{proc}$$

$$\frac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Gamma_1 \vdash a.V :: (c : A_m)}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash !^{+}_{m}\mathsf{msg}(a_m, V) :: \Gamma_C, \Gamma_2, (c : A_m)}\ \mathsf{msg}$$

$$\frac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Gamma_1 \vdash \mathsf{case}\, a\, K :: (c : A_m)}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash !^{-}_{m}\mathsf{srv}(a_m, K) :: \Gamma_C, \Gamma_2, (c : A_m)}\ \mathsf{srv}$$

$$\frac{}{\Gamma \vDash (\cdot) :: \Gamma}\ \mathsf{empty} \qquad\qquad \frac{\Gamma_1 \vDash \mathcal{C}_1 :: \Gamma_2 \quad \Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3}{\Gamma_1 \vDash \mathcal{C}_1, \mathcal{C}_2 :: \Gamma_3}\ \mathsf{join}$$

Note that in each of the singleton object rules, we need that $\Gamma_C, \Gamma_1 \geq m$, in order for independence to be satisfied when typing processes, and we also need to ensure that $\mathsf{C} \in \sigma(\Gamma_C)$, allowing it to both be used for typing a process (resp. message, service) and to be passed on for further objects in the configuration to use. However, in the empty and join rules, there are no such restrictions, because configurations can freely pass through any channels unused, even if no process in that configuration is permitted by independence to use the channel.

For handling forwarding, we treat $a.c$ and $\mathsf{case}\, a\, c$ as synonymous with $a \leftarrow c$, allowing $\mathsf{msg}(a_m, c_m)$ and $\mathsf{srv}(a_m, c_m)$ to be typed without additional rules. Note that in the msg and srv rules, the channel c that the message or service provides need not be the channel a that it is communicating along, but nor need it be different. We can distinguish these cases based on the polarity (positive or negative) of the channel a, and tracking this information explicitly in the syntax may simplify type-checking in practice, but complicates the presentation of the rules. As such, we prefer the more succinct form.

We can see immediately from these rules that the preconditions of Theorem 17 on type extension hold — the Γ_2 in each singleton rule allows us to add any additional context (not sharing channels with Γ_C, Γ_1 or the object being typed) to be passed through a single object. Moreover, our general results in section 4.2.5 tell us that we may freely reassociate configurations without affecting their types. This ability to reassociate, in particular, will be convenient when proving type safety.

We will also make use of the following lemma, which allows us to invert typing to find the provider of a given channel that occurs in the type of a configuration.

Lemma 6 (Every channel has a unique provider)**.** *Suppose* $\Gamma \vDash \mathcal{C} :: \Delta, (a : A_m)$.

Then, either $(a : A_m)$ *occurs in* Γ *(a is an externally-visible channel on the left), or we can find contexts* Γ', Δ' *and a unique object* ϕ *such that:*

- $(a : A_m)$ *does not occur in* Γ'
- ϕ *occurs in* $\mathcal{C}$
- $\Gamma' \vDash \phi :: \Delta', (a : A_m)$*. Note that in particular, this means that* a *occurs in* ϕ*, as can be seen by inversion.*

We say that this object ϕ *is a* provider *of* a*. Moreover, these two cases are mutually exclusive — it cannot be the case that* $(a : A_m)$ *occurs in* Γ and $\mathcal{C}$ *contains a provider of* a*.*

Proof. This proof proceeds by induction on the derivation of $\Gamma \vDash \mathcal{C} :: \Delta, (a : A_m)$.

If the derivation ends with the empty rule, then $\Gamma = \Delta, (a : A_m)$, and so we are in the first case, where a is externally visible on the left. As $\mathcal{C}$ is empty, it cannot contain a provider of a.

If the derivation ends with the join rule, we have that $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$, and that for some Γ', $\Gamma \vDash \mathcal{C}_1 :: \Gamma'$ and $\Gamma' \vDash \mathcal{C}_2 :: \Delta, (a : A_m)$. We first apply the inductive hypothesis to this second premise, giving us two possible cases. In the first case, $\mathcal{C}_2$ contains a (unique) provider of a and Γ' does not contain $(a : A_m)$. Since typing does not generate fresh symbols (Theorem 17) and Γ' does not contain $(a : A_m)$, we also have that Γ and $\mathcal{C}_1$ cannot contain a, concluding this case.

In the second case, Γ' contains $(a : A_m)$ (and $\mathcal{C}_2$ does not contain a provider of a). In this latter case, we apply the inductive hypothesis again to the typing of $\mathcal{C}_1$, giving us that either $\mathcal{C}_1$ contains a unique provider of a, or Γ contains $(a : A_m)$, but not both In either case, we are done.

If the derivation ends with any of the other three rules, we observe that its conclusion must have the following form:

$$\Gamma_C, \Gamma_1, \Gamma_2 \vDash \phi :: \Gamma_C, \Gamma_2, (c : C_k),$$

where $\Gamma = \Gamma_C, \Gamma_1, \Gamma_2$, and $\Delta, (a : A_m) = \Gamma_C, \Gamma_2, (c : C_k)$. If $(a : A_m)$ occurs in Γ, then it is an externally-visible channel, and so we need only check that ϕ is not a provider of a, but this is immediate from the definition. Otherwise, it cannot occur in either Γ_C or Γ_2, and so it must be the case that $(a : A_m) = (c : C_k)$, giving us the last condition needed to ensure that ϕ is a provider of a. □

4.3.3 Results

At this point, we are equipped with the tools to present and prove theorems about the semantics of this language. We begin with type safety, which, as is often the case, is broken down into a preservation-type theorem, called session fidelity, and a progress-type theorem, called deadlock-freedom. Both resemble their standard functional counterparts, but a few unusual features arise. In session fidelity, we may not retain exactly the same type for a configuration, as new channels may be allocated and exposed to the outside world. However, the types of all existing channels must remain the same, and externally visible channels cannot be destroyed, only newly created. This ensures that it is always possible to continue to interact with a configuration without unexpectedly having an existing communication fail, either by the channel disappearing or changing type.

Theorem 18 (Session fidelity (Preservation))**.** *Suppose* $\Gamma \vDash \mathcal{C} :: \Delta$ *and* $\mathcal{C} \mapsto \mathcal{C}'$*. Then* $\Gamma \vDash \mathcal{C}' :: \Delta'$ *for some* $\Delta' \supseteq \Delta$*.*

Proof. We proceed by case analysis on the choice of rule used to step from $\mathcal{C}$ to $\mathcal{C}'$.

For six of the seven rules, we start with a single object $\mathsf{proc}(P)$ in $\mathcal{C}$. Reassociate $\mathcal{C}$ as $(\mathcal{C}_1, \mathsf{proc}(P)), \mathcal{C}_2$.[7] Now, we wish to replace $\mathsf{proc}(P)$ with the right-hand side $\hat{\mathcal{C}}$ of the rule used to step, and claim that $\mathsf{proc}(P)$ and $\hat{\mathcal{C}}$ have the same type as configurations, and therefore so does the overall configuration $\mathcal{C}' = (\mathcal{C}_1, \hat{\mathcal{C}}), \mathcal{C}_2$. It therefore suffices for these rules to show that the left-hand and right-hand side have the same type as configurations, ignoring any other parts of the configuration. For the two identity rules and the send/listen rules, this is immediate, as the typing rules for the newly created message or service make use of typing for the same process

[7] $\mathcal{C}_1, (\mathsf{proc}(P), \mathcal{C}_2)$ would work as well.

that creates them. Likewise, the rule for calling processes replaces a call to a named process (which is typed using the declaration for that named process) with the definition of that process, which, given a valid signature, must be well-typed using the type specified in its declaration, and so the typing derivation for the right-hand side is nearly identical to that for the left-hand side of the rule. The cut rule is slightly more involved, as it converts one object into two, but can be seen as the following translation of proofs (where we omit side conditions on modes for space):

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma_C, \Gamma_1 \vdash P :: (x : A_m)} \quad \overset{\mathcal{E}}{\Gamma_C, \Delta_1, (x : A_m) \vdash Q :: (c : C_r)}}{\Gamma_C, \Gamma_1, \Delta_1 \vdash x \leftarrow P \ ; Q :: (c : C_r)}\ \mathsf{cut}}{\Gamma_C, \Gamma_1, \Delta_1, \Gamma_2 \vDash \mathsf{proc}(x \leftarrow P \ ; Q) :: \Gamma_C, \Gamma_2, (c : C_r)}\ \mathsf{proc}$$

becomes the join of the following two proofs:

$$\dfrac{\overset{\mathcal{D}}{\Gamma_C, \Gamma_1 \vdash P :: (x : A_m)}}{\Gamma_C, \Gamma_1, \Delta_1, \Gamma_2 \vDash \mathsf{proc}([a/x]P) :: \Gamma_C, \Delta_1, (x : A_m), \Gamma_2}\ \mathsf{proc}$$

and

$$\dfrac{\overset{\mathcal{E}}{\Gamma_C, \Delta_1, (x : A_m) \vdash Q :: (c : C_r)}}{\Gamma_C, \Delta_1, (x : A_m) \vDash \mathsf{proc}([a/x]Q) :: \Gamma_C, \Gamma_2, (c : C_r)}\ \mathsf{proc}$$

For the seventh rule, communication, we start with two objects in $\mathcal{C}$, and end up with one. This rule must be treated slightly differently depending on whether the type of the channel a being communicated along is positive or negative, and on what exactly the type is, but all are similar.

As an example case, we consider $a : B_m \otimes C_m$.

In this case, we can write (reassociating with Theorem 16 if necessary)

$$\mathcal{C} = \mathcal{C}_1, !_m\mathsf{msg}(a_m, \langle b, c\rangle), \mathcal{C}_2, \mathsf{srv}(a_m, (\langle x, y\rangle \Rightarrow P)), \mathcal{C}_3$$

Now, by inverting the typing derivation for $\mathcal{C}$, we get that there are $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ such that:

- $\Gamma \vDash \mathcal{C}_1 :: \Gamma_1$
- $\Gamma_1 \vDash !_m\mathsf{msg}(a, \langle b, c\rangle) :: \Gamma_2$
- $\Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3$
- $\Gamma_3 \vDash \mathsf{srv}(a_m, (\langle x, y\rangle \Rightarrow P)) :: \Gamma_4$
- $\Gamma_4 \vDash \mathcal{C}_3 :: \Delta$.

By inversion on the typing derivations for the message and service, respectively, we get that Γ_2 and Γ_3 must both contain $(a : B_m \otimes C_m)$. We also learn which case of the $\otimes L_\alpha$ rule is used in typing the service, and note this value of α down. Let $\beta = 1$ if m admits contraction and 0 otherwise, observing that $\beta - \alpha \in \{0, 1\}$ for any possible choices of α, β.

Write $\Gamma_2 = \Gamma_2', (a : B_m \otimes C_m)$, and let $\Gamma_2'' = \Gamma_2', (a : B_m \otimes C_m)^\beta, (b : B_m), (c : C_m)$.[8] Similarly, write $\Gamma_3 = \Gamma_3', (a : B_m \otimes C_m)$, and let $\Gamma_3'' = \Gamma_3', (a : B_m \otimes C_m)^\beta, (b : B_m), (c : C_m)$.

We then make the following three claims, whose proofs we defer for now:

[8] Note that Γ_2 may already contain $(b : B_m)$ and/or $(c : C_m)$ in some cases. Since we have distinct symbols attached to each assumption in our contexts here, we may treat them as *sets*, so adding a second copy of $(b : B_m)$ or $(c : C_m)$ is just a no-op. This greatly simplifies the number of case distinctions that need to be made.

(1) There is some $\hat{\Gamma}$ such that $\Gamma_1 \vDash C_m?(!_m\mathsf{msg}(a, \langle b, c\rangle)) :: \Gamma_2'', \hat{\Gamma}$, where the notation $C_m?(x, y)$ denotes x if $\mathsf{C} \in \sigma(m)$ and y otherwise

(2) $\Gamma_2'' \vDash \mathcal{C}_2 :: \Gamma_3''$.

(3) $\Gamma_3'' \vDash \mathsf{proc}(P[b/x, c/y]) :: \Gamma_4'$ for some $\Gamma_4' \supseteq \Gamma_4$.

Combining these and our typing information about $\mathcal{C}_1$ and $\mathcal{C}_3$ with the join rule, possibly using type extension to handle both $\hat{\Gamma}$ and a Γ_4' that is larger than Γ_4, we get that there is some $\Delta' \supseteq \Delta$ such that

$$\Gamma \vDash \mathcal{C}_1, C_m?(!_m\mathsf{msg}(a, \langle b, c\rangle)), \mathcal{C}_2, \mathsf{proc}(P[b/x, c/y]), \mathcal{C}_3 :: \Delta'.$$

As this configuration is exactly $\mathcal{C}'$, this will give the desired result, and so it suffices to prove (1)-(3).

Claim (1) We begin by applying inversion to the typing derivation of the provider of a. In this case, we get that the derivation has the following form:

$$\dfrac{\Delta_C, \Delta_1 \geq m \quad \mathsf{C} \in \sigma(\Delta_C) \quad \dfrac{}{\Delta_C, \Delta_1 \vdash a.\langle b, c\rangle :: (a : B_m \otimes C_m)}\otimes R^0}{\Delta_C, \Delta_1, \Delta_2 \vDash !_m\mathsf{msg}(a, \langle b, c\rangle) :: \Delta_C, \Delta_2, (a : B_m \otimes C_m)}\,\mathsf{msg}$$

We also conclude that Δ_C, Δ_1 contains $(b : B_m)$ and $(c : C_m)$. Now, we distinguish cases based on whether m admits contraction or not.

If m admits contraction, then, taking $\Delta_C' = \Delta_C, (b : B_m), (c : C_m)$ and Δ_1' to be the result of removing b and c from Δ_1 (if they occurred there to begin with), we get that $\Delta_C', \Delta_1' = \Delta_C, \Delta_1$, and so we can construct the following typing derivation:

$$\dfrac{\Delta_C', \Delta_1' \geq m \quad \mathsf{C} \in \sigma(\Delta_C') \quad \dfrac{}{\Delta_C', \Delta_1' \vdash a.\langle b, c\rangle :: (a : B_m \otimes C_m)}\otimes R^0}{\Delta_C', \Delta_1', \Delta_2 \vDash !_m\mathsf{msg}(a, \langle b, c\rangle) :: \Delta_C', \Delta_2, (a : B_m \otimes C_m)}\,\mathsf{msg}$$

As m admits contraction, the condition $\mathsf{C} \in \sigma(\Delta_C')$ remains valid. The above is exactly a derivation of $\Gamma_1 \vDash !_m\mathsf{msg}(a, \langle b, c\rangle) :: \Gamma_2''$, and so this case is complete.

If m does not admit contraction, then we construct the following typing derivation:

$$\dfrac{}{\Delta_C, \Delta_1, \Delta_2 \vDash (\cdot) :: \Delta_C, \Delta_1, \Delta_2}\,\mathsf{empty}$$

Since $\Gamma_2 = \Delta_C, \Delta_2, (a : B_m \otimes C_m)$, we also have that $\Gamma_2'' = \Delta_C, \Delta_2, (B : B_m), (c : C_m)$, as we are in the case where m does not admit contraction, so α must be 0. Taking $\hat{\Gamma}$ to be the remainder of Δ_1 after $(b : B_m)$ and $(c : C_m)$ are removed, we also complete this case. Note that the symbols in $\hat{\Gamma}$ cannot occur in $\mathcal{C}_2, \mathsf{proc}(P[b/x, c/y]), \mathcal{C}_3$, as they are weakened away by typing for this message in $\mathcal{C}$. As such, they are eligible for type extension.

Claim (2) We will prove by induction over the typing derivation that whenever

$$\Psi, (a : B_m \otimes C_m) \vDash \mathcal{D} :: \Psi', (a : B_m \otimes C_m),$$

it is also the case that

$$\Psi, (a : B_m \otimes C_m)^\beta (b : B_m), (c : C_m) \vDash \mathcal{D} :: \Psi', (a : B_m \otimes C_m)^\beta (b : B_m), (c : C_m),$$

where β is 1 if m admits contraction and 0 otherwise. For simplicity, we will write Δ_a for the context $(a : B_m \otimes C_m)^\beta (b : B_m), (c : C_m)$.

If the last rule used was empty, then $\Psi = \Psi'$, and the result is immediate.

If the last rule used was join, then we have that $\mathcal{D} = \mathcal{D}_1, \mathcal{D}_2$, and there is some Ψ'' such that $\Psi, (a : B_m \otimes C_m) \vDash \mathcal{D}_1 :: \Psi''$ and $\Psi'' \vDash \mathcal{D}_2 :: \Psi', (a : B_m \otimes C_m)$. In order to apply the inductive hypothesis, we need to know that Ψ'' contains $(a : B_m \otimes C_m)$. We get this from the fact that every channel has a unique provider (Lemma 6) — since $(a : B_m \otimes C_m)$ occurs on both sides of the typing derivation for $\mathcal{D}$, there can be no ϕ in $\mathcal{D}$ that provides a, so no ϕ in $\mathcal{D}_2$ can provide a either. As such, $(a : B_m \otimes C_m)$ must occur in Ψ''. Write $\Psi'' = \hat{\Psi}, (a : B_m \otimes C_m)$.

Now, applying the inductive hypothesis to both $\mathcal{D}_1$ and $\mathcal{D}_2$, we get that

$$\Psi, \Delta_a \vDash \mathcal{D}_1 :: \hat{\Psi}, \Delta_a \qquad \text{and} \qquad \hat{\Psi}, \Delta_a \vDash \mathcal{D}_2 :: \Psi', \Delta_a.$$

Applying the join rule to these gives the desired result.

Now, if the last rule used was one of the three singleton rules, $\mathcal{D}$ is a single object ϕ, and

$$\Delta_C, \Delta_1, \Delta_2 \vDash \phi :: \Delta_C, \Delta_2, (d : D_k)$$

for some $d : D_k$ and $\Delta_C, \Delta_1, \Delta_2$ a partition of $\Psi, (a : B_m \otimes C_m)$. Since $(a : B_m \otimes C_m)$ occurs both on the left and the right of this typing judgment, it must occur as part of Δ_C, Δ_2.[9]

If a occurs in Δ_2, write $\Delta_2 = \Delta_2', (a : B_m \otimes C_m)$, and let $\Delta_2'' = \Delta_2', \Delta_a$. Then, the same typing derivation used for ϕ above also shows that

$$\Delta_C, \Delta_1, \Delta_2'' \vDash \phi :: \Delta_C, \Delta_2'', (d : D_k),$$

since the context Δ_2 may be chosen arbitrarily in each of the singleton typing rules. This is exactly the desired result.

Otherwise, a occurs in Δ_C, and so we must be in the case where m admits contraction. Let Δ_2' be the context obtained from Δ_2 by adding those of $(b : B_m), (c : C_m)$ that *do not* already occur in Δ_C. With this construction, $\Delta_C, \Delta_1, \Delta_2' = \Psi, \Delta_a$, and $\Delta_C, \Delta_2', (d : D_k) = \Psi', \Delta_a$, and so, replacing Δ_2 with Δ_2' in the above typing derivation for ϕ, we get the desired result.

From this general statement and the fact that $\Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3$ (from the typing of $\mathcal{C}$), we get in particular that $\Gamma_2'' \vDash \mathcal{C}_2 :: \Gamma_3''$, which is exactly claim (2).

[9] We can be sure that it does not occur as $(d : D_k)$ and part of Δ_1 because a well-typed process term cannot have the same channel on both the left and right side of its type — this is easy to check by observing that this property is preserved by all typing rules for processes.

Claim (3) We now apply inversion to the typing derivation of the client of a. In this case, we get that the derivation has the following form:

$$\dfrac{\Delta_C, \Delta_1 \geq m \quad \mathsf{C} \in \sigma(\Delta_C) \quad \dfrac{\overset{\mathcal{D}}{\Delta'_C, \Delta'_1, (a : B_m \otimes C_m)^\alpha, (x : B_m), (y : C_m) \vdash P :: (d : D_k)}}{\Delta_C, \Delta_1 \vdash \mathsf{case}\ a\ (\langle x, y \rangle \Rightarrow P) :: (d : D_k)} \otimes L_\alpha}{\Delta_C, \Delta_1, \Delta_2 \vDash \mathsf{srv}(a_m, (\langle x, y \rangle \Rightarrow P)) :: \Delta_C, \Delta_2, (d : D_k)} \ \mathsf{srv}$$

Here, $\Gamma_3 = \Delta_C, \Delta_1, \Delta_2$ and $\Gamma_4 = \Delta_C, \Delta_2, (d : D_k)$ and $\Delta_C, \Delta_1 = \Delta'_C, \Delta'_1, (a : B_m \otimes C_m)$. We may also assume that either $\Delta_C = \Delta'_C$ or $\Delta_1 = \Delta'_1$ — that is, the only difference is the removal of a.

Let $\Delta''_C = \Delta'_C, (a : B_m \otimes C_m)^\alpha$. Now, Δ_C (and hence also Δ'_C and Δ''_C) may contain $(b : B_m)$ or $(c : C_m)$ already. We define Δ''_1 to be the result of adding $(b : B_m)$ and $(c : C_m)$ to Δ'_1 *if they do not already occur in* Δ_C, and $\Delta'_2 = \Delta_2, (a : B_m \otimes C_m)^{\beta - \alpha}$ — that is, Δ'_2 contains $(a : B_m \otimes C_m)$ exactly when m admits contraction, but the $\otimes L_0$ rule was used in typing the client of a. Then, $\Delta''_C, \Delta''_1 = \Delta'_C, \Delta'_1, (a : B_m \otimes C_m)^\alpha, (b : B_m), (c : C_m)$, and so $\Gamma''_3 = \Delta''_C, \Delta''_1, \Delta'_2$. Now, we construct the following typing derivation:

$$\dfrac{\Delta''_C, \Delta''_1 \geq m \quad \mathsf{C} \in \sigma(\Delta''_C) \quad \overset{\mathcal{D}[b/x, c/y]}{\Delta''_C, \Delta''_1 \vdash P[b/x, c/y] :: (d : D_k)}}{\Delta''_C, \Delta''_1, \Delta'_2 \vDash \mathsf{proc}(P[b/x, c/y]) :: \Delta''_C, \Delta'_2, (d : D_k)} \ \mathsf{proc}$$

This is a derivation of $\Gamma''_3 \vDash \mathsf{proc}(P[b/x, c/y]) :: \Delta''_C, \Delta'_2, (d : D_k)$, and so it will suffice to show that $\Delta''_C, \Delta'_2, (d : D_k) \supseteq \Gamma_4$. By construction, $\Delta''_C, \Delta'_2 = \Delta'_C, \Delta_2, (a : B_m \otimes C_m)^\beta$, and as $\Delta'_C, (a : B_m \otimes C_m)^\beta \supseteq \Delta_C$ (either $\Delta_C = \Delta'_C$ or $\Delta_C = \Delta'_C, (a : B_m \otimes C_m)$, but the latter case can only occur when m admits contraction, so $\beta = 1$), we see that $\Delta''_C, \Delta'_2, (d : D_k) \supseteq \Gamma_4$, and conclude the proof of (3). Note that if $(a : B_m \otimes C_m)$ does not occur in Γ_4, the well-typedness of $\mathcal{C}$ ensures that a cannot occur in $\mathcal{C}_3$ (in principle, some internal channel of $\mathcal{C}_3$ could share the name a, but we assume that channel names are unique for well-typed configurations). As such, it is eligible for type extension.

Other cases The other cases for the type A_m of a follow a similar structure. We first reassociate $\mathcal{C}$ to isolate the two interacting objects. From inversion, we can get some typing information about these objects — in particular, we find the channel that they communicate along in both Γ_2 and Γ_3, and we also find the continuation channels (in this case, $b : B_m$ and $c : C_m$) that the new process that arises from the communication will need to use. We can then construct modified contexts Γ''_2 and Γ''_3 by adding in these channels that the new process will use, and potentially removing the channel a, depending on which α case of a left rule is used to type the client of a.

Then, proving (suitable variants of) (1)-(3) is sufficient to give the result. (1) consists of showing that channels that the provider of a gives to the new process will remain available after communication, (2) shows that these channels can be "threaded through" an intervening configuration $\mathcal{C}_2$ without getting lost, and (3) shows that these, along with any channels that the *client* of a gives to the new process, are sufficient to type that new process. □

Deadlock-freedom likewise differs slightly from its functional counterpart, more closely following progress or deadlock-freedom theorems in the context of other process calculi. In systems derived from the π-calculus, deadlock-freedom generally states that any process can either take a reduction step, or there is no pending communication (see, e.g., [20, 50, 70] for a few examples of such systems and their deadlock-freedom). Our language, unlike many π-calculus systems, supports a notion of external input/output, in the form of channels that occur free in a process term or configuration, and so we need to also account for this external communication in our statement of deadlock-freedom — a process waiting on external communication cannot be reasonably expected to progress on its own, and so should not be considered in deadlock. We say that a message or a service is *poised* if it is a message of positive type (being sent from provider to client) or a service of negative type (listening for messages from its client). In either case, the object is waiting for a client to interact with it. We cannot identify poised objects purely syntactically — for instance, the message $\mathsf{msg}(a, \langle b, c \rangle)$ could be typed either with $\otimes R^0$ or with $\multimap L^0$, and is only poised in the former case. However, as we are working in a setting where everything is well-typed, we can say that an object is poised if its typing derivation ends with first the msg or srv rule, followed by the right rule for a connective. Configurations consisting entirely of poised objects serve a similar role to values in a functional language, being unable to continue computation on their own (although they may be able to continue after provided with some external input, much like a lambda abstraction can continue to take steps once provided with its argument).

Theorem 19 (Deadlock freedom (Progress)). *Suppose $(\cdot) \vDash \mathcal{C} :: \Delta$. Then, one of the following holds:*

- *There is some $\mathcal{C}'$ such that $\mathcal{C} \mapsto \mathcal{C}'$.*
- *$\mathcal{C}$ consists only of poised objects — that is, messages of positive type or services of negative type. In particular, it contains no* proc *objects.*

Proof. We begin by working with a typing derivation for $(\cdot) \vDash \mathcal{C} :: \Delta$ which is fully associated to the left — that is, (presuming $\mathcal{C}$ is non-empty) $\mathcal{C} = \mathcal{C}', \phi$ for some object ϕ, and so on.

By inversion, we can then find Δ' with $(\cdot) \vDash \mathcal{C}' :: \Delta'$ and $\Delta' \vDash \phi :: \Delta$. Applying the inductive hypothesis to $\mathcal{C}'$, either $\mathcal{C}'$ can take a step (in which case, so can the whole configuration $\mathcal{C}$), or $\mathcal{C}'$ consists entirely of poised objects.

Now, we consider ϕ. If ϕ is a proc object, it can take a step on its own, and therefore $\mathcal{C}$ can take this step. Otherwise, if ϕ is a poised msg or srv object, $\mathcal{C}$ consists entirely of poised objects and we are done.

The remaining case is where $\mathcal{C}'$ consists entirely of poised objects, and ϕ is a non-poised $\mathsf{msg}(a, V)$ or $\mathsf{srv}(a, K)$. We now distinguish cases based on whether ϕ is a message or a service, and on what shape its value or continuation takes (or, equivalently, what type ϕ expects a to have). These cases are all similar, and so we will highlight one positive and one negative example.

First, suppose ϕ is of the form $\mathsf{msg}(a, i(b))$. Since we know (as ϕ is not poised) that it is a message of negative type, this $\ell(b)$ must be typed using the $\& L^0$ rule, rather than $\oplus R^0$. We apply inversion to the typing derivation for ϕ, finding that it must have the form

$$\dfrac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \dfrac{(i \in L)}{\Gamma'_C, \Gamma'_1, a : \&\{\ell : A^\ell_m\}_{\ell \in L} \vdash a.i(b) :: (b : A^i_m)}\,\& L^0}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash \mathsf{msg}(a_m, i(b)) :: \Gamma_C, \Gamma_2, (b : A^\ell_m)}\,\mathsf{msg}$$

where $\Delta' = \Gamma_C, \Gamma_1, \Gamma_2$, $\Delta = \Gamma_C, \Gamma_2, (b : A_m^\ell)$, and $\Gamma_C, \Gamma_1 = \Gamma'_C, \Gamma'_1, a : \&\{\ell : A_m^\ell\}_{\ell \in L}$ — the particulars of whether a appears in Γ_C or Γ_1 are, thankfully, irrelevant here.

Now, we know that $(\cdot) \vDash \mathcal{C}' :: \Delta'$, and that Δ' contains $a : \&\{\ell : A_m^\ell\}_{\ell \in L}$. Applying Lemma 6, we get that $\mathcal{C}'$ must contain a provider ψ of a. That is, there are some Γ', Γ'' such that $(a : \&\{\ell : A_m^\ell\}_{\ell \in L})$ does not occur in Γ', and

$$\Gamma' \vDash \psi :: \Gamma'', (a : \&\{\ell : A_m^\ell\}_{\ell \in L})$$

Because all objects in $\mathcal{C}'$ are poised, ψ must be either a positive message or a negative service — that is, it must be typed using a right rule. If we search through the possible right rules, we find that the only way to type ψ such that $(a : \&\{\ell : A_m^\ell\}_{\ell \in L})$ occurs on the right, but not on the left, is via the $\& R$ rule. As such, ψ must be of the form $!_m \mathsf{srv}(a_m, (\ell(y) \Rightarrow P_\ell)_{\ell \in L})$.

We then observe that the communication rule applies to ϕ and ψ, as they are a compatible service/message pair, and so the overall configuration $\mathcal{C}$ can take a step.

For a positive example, we suppose that ϕ is of the form $\mathsf{srv}(a, (\mathsf{shift}(x) \Rightarrow P))$. Knowing that ϕ is not poised allows us to conclude that it is a service of positive type, and hence typed by $\downarrow L$, rather than by $\uparrow R$. Applying inversion to the typing derivation for ϕ, we find that it has the form

$$\dfrac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \dfrac{\Gamma'_C, \Gamma'_1, (a : {\downarrow}_m^\ell A_\ell)^\alpha, x : A_\ell \vdash P :: (c : C_r)}{\Gamma'_C, \Gamma'_1, a : {\downarrow}_m^\ell A_\ell \vdash \mathsf{case}\ a\ (\mathsf{shift}(x) \Rightarrow P) :: (c : C_r)} {\downarrow} L_\alpha}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash \mathsf{srv}(a_m, (\mathsf{shift}(x) \Rightarrow P)) :: \Gamma_C, \Gamma_2, (c : C_r)}$$

As in the previous case, we note that $\Delta' = \Gamma_C, \Gamma_1, \Gamma_2$ and $\Gamma_C, \Gamma_1 = \Gamma'_C, \Gamma'_1, a : {\downarrow}_m^\ell A_\ell$. In particular, this means that Δ' contains $a : {\downarrow}_m^\ell A_\ell$, and so we may apply Lemma 6 to the typing derivation for $\mathcal{C}'$ to get that $\mathcal{C}'$ contains a provider ψ of a. Applying inversion to the typing derivation for ψ, we find that it must be typed with the msg rule, followed by ${\downarrow} R^0$, analogously to the previous case. We may then conclude that ψ has the form $!_m \mathsf{msg}(a_m, \mathsf{shift}(b))$ for some b, and that the communication rule applies to ϕ and ψ, allowing $\mathcal{C}$ to take a step.

The remaining cases follow a similar pattern — we take the object ϕ to have a particular form, and use that it is not poised to determine (part of) its typing derivation. From this, we can conclude that a occurs in Δ' with some type A_m, find a provider ψ of a in $\mathcal{C}'$, and then again apply inversion to the typing derivation to see what shape ψ has. Typing then ensures that ϕ and ψ are compatible, so we can apply the communication rule. □

Combining these two theorems, we get a form of type-safety — a well-typed closed configuration is either completely composed of poised objects (and therefore blocked waiting on external communication), or it can take a step, and after that step, it remains well-typed. As such, a well-typed configuration has two possibilities: either it takes some finite number of steps and reaches a final configuration, where all objects are poised, or there is an infinite sequence of steps it can take. In either case, we get a variant of the classic motto: "Well-typed configurations do not go wrong" [61].

A natural question at this point, since the semantics do not prescribe an order in which to apply rules, is whether it is possible for some well-typed configuration to either continue infinitely taking steps or reach a poised state, depending on the choices of rules used. We can show that in fact, while there is nondeterminism in the order to apply rules, it cannot have an impact on the final result of a computation.

Theorem 20 (Diamond Lemma)**.** *Suppose $\Gamma \vDash \mathcal{C} :: \Delta$, $\mathcal{C} \mapsto \mathcal{C}_1$, and $\mathcal{C} \mapsto \mathcal{C}_2$. Then there exist $\mathcal{C}_1'$ and $\mathcal{C}_2'$ such that $\mathcal{C}_1 \mapsto \mathcal{C}_1'$, $\mathcal{C}_2 \mapsto \mathcal{C}_2'$, and $\mathcal{C}_1'$ is equivalent to $\mathcal{C}_2'$ up to a renaming of symbols.*

Proof. This proof follows the same structure as the Church-Rosser theorem for the lambda calculus. However, it is somewhat simpler, in that our reduction rules interfere less with each other than those for the lambda calculus.

We observe that most of the computation rules for this language are independent — that is, they cannot possibly operate on the same object. As such, we can apply both rules in either order, yielding the same configuration in the end (up to choice of fresh channel name in the cut rule, which is resolved by renaming).

The only way we can have two rule instances apply to the same object is if they are two different instances of the communication rule, applied to the same message or service. In this case, however, the message or service must be a provider interacting with different clients — otherwise, we would end up with a type error from some channel having multiple providers. We then observe that a message or service with multiple clients is persistent — it must be providing a channel of positive or negative type, respectively, and since it has multiple clients, the mode of that type must admit contraction. As such, we can take the two steps sequentially in either order, and as neither destroys the persistent message or service being provided, the resulting configurations are identical. □

4.4 Shared-Memory Semantics

With only minimal changes, we can provide a system of semantics based on a limited form of shared memory, which we will see is a form of substructural futures. [32, 39] In this system, we interpret symbols as addresses of memory cells, each of which can be written to once, and read from potentially many times. Processes then are able to communicate (in one direction only) by writing data into memory, which can then be read by other processes. Unlike in the message passing setting, where only values V could be messages, both values V and continuations K may be stored in memory, with continuations representing a form of paused process. Because of this, the constructs $a.V$ and $\mathsf{case}\ a\ K$ each serve dual purposes, able to either read or write from memory depending on the type of V or K, respectively. When $a.V$ is used to write to memory, it stores the value V in the cell at address a, but when it is used to read from memory, it loads the stored continuation K at address a, and continues executing $V \triangleright K$. The construct $\mathsf{case}\ a\ K$ behaves dually, either writing a continuation to memory at address a or reading a stored value V from memory and continuing as $V \triangleright K$.

In order to disambiguate, we may write superscript R or W on addresses in programs to indicate whether we are reading or writing from the address, when it is helpful to be precise. While this can be inferred by looking at the program as a whole, and so is not part of the formal syntax, we will use it to improve readability in examples and in some discussions of the language. Using this notation, we describe more precisely the meanings of process terms under a shared-memory interpretation in table 4.3

Process term P	Meaning
$c^W \leftarrow a^R$	Copy (or move) the contents of cell a to address c.
$x \leftarrow P \; ; Q$	Allocate a new memory cell with address a, spawn a process $P[a/x]$, and continue as $Q[a/x]$. $P[a/x]$ may write to a, while $Q[a/x]$ may read from a.
$c^W.V$	Write the value V to the cell at address c.
$\mathsf{case}\, c^R\ K$	Read a value V from cell c, then pass V to K.
$c^R.V$	Read a continuation K from cell c, then pass V to K.
$\mathsf{case}\, c^W\ K$	Write the continuation K to the cell at address c.
$a \leftarrow p\ \overline{b}$	Call the named process p, which may read from addresses $\overline{b}$ and write to address a.

Table 4.3: Meanings of process terms in a shared-memory setting

We can provide a set of typing rules (which are largely similar to those for the message-passing semantics of section 4.3) based on a semi-axiomatic presentation of ADJ^I. These can be found in fig. 4.3.

The differences between message-passing and shared memory semantics become more pronounced when we look at the dynamic semantics. Our shared-memory semantics use three types of semantic object:

- $\mathsf{thread}(a, P)$, representing a thread of computation executing the term P with destination a.
- $\mathsf{cell}(a, \_)$, representing an empty memory cell at address a.
- $!_m\mathsf{cell}(a_m, D)$, representing a cell at address a_m, with contents D, which may be either a value V or continuation K. Here, $!_m$ denotes that the cell is persistent if $\mathsf{C} \in \sigma(m)$, and ephemeral otherwise.

We require that every $\mathsf{proc}(a, P)$ has a corresponding empty cell $\mathsf{cell}(a, \_)$ which it will eventually write to, and likewise, that every empty cell has a corresponding process. Because of this, it is possible to simplify the semantics by leaving empty cells implicit, but this makes memory allocation less clear, and so we prefer to show these empty cells explicitly. We also require that if two objects share the same address, then they are exactly such a $\mathsf{proc}(a, P), \mathsf{cell}(a, \_)$ pair. This ensures that no two memory cells share an address, that no two processes share a destination, and that no process is attempting to write to an already filled cell, thus avoiding any possible write conflicts. We will see later that every well-typed configuration satisfies this condition automatically.

Our semantics here resemble a *destination-passing style* [53, 107] semantics for a functional language, where rather than substituting values for variables, instead, values are written to memory at a given destination, and that destination is instead substituted for a variable. For instance, when evaluating $(\lambda x.e_1)\ e_2$ with destination d, a destination-passing system of semantics

$$\frac{(\Gamma_C, \Delta \geq m \geq r) \quad \Gamma_C, \Delta \vdash P :: (x : A_m) \quad \Gamma_C, \Delta', x : A_m \vdash Q :: (z : C_r)}{\Gamma_C, \Delta, \Delta' \vdash (x \leftarrow P \,;\, Q) :: (z : C_r)}\ \mathsf{cut}$$

$$\frac{}{\Gamma_W, y : A_m \vdash x \leftarrow y :: (x : A_m)}\ \mathsf{id}$$

$$\frac{(i \in L)}{\Gamma_W, y : A^i_m \vdash x^W.i(y) :: (x : \oplus\{\ell : A^\ell_m\}_{\ell \in L})}\ \oplus R^0$$

$$\frac{\Gamma, (x : \oplus\{\ell : A^\ell_m\}_{\ell \in L})^\alpha, y : A^\ell_m \vdash Q_\ell :: (z : C_r) \quad (\text{for all } \ell \in L)}{\Gamma, x : \oplus\{\ell : A^\ell_m\}_{\ell \in L} \vdash \mathbf{case}\, x^R\, (\ell(y) \Rightarrow Q_\ell)_{\ell \in L} :: (z : C_r)}\ \oplus L_\alpha$$

$$\frac{\Gamma \vdash P_\ell :: (y : A^\ell_m) \quad (\text{for all } \ell \in L)}{\Gamma \vdash \mathbf{case}\, x^W\, (\ell(y) \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A^\ell_m\}_{\ell \in L})}\ \& R$$

$$\frac{(i \in L)}{\Gamma_W, x : \&\{\ell : A^\ell_m\}_{\ell \in L} \vdash x^R.i(y) :: (y : A^i_m)}\ \& L^0$$

$$\frac{}{\cdot \vdash x^W.\langle\,\rangle :: (x : \mathbf{1}_m)}\ \mathbf{1}R^0 \qquad \frac{\Gamma, (x : \mathbf{1})^\alpha \vdash P :: (z : C_r)}{\Gamma, x : \mathbf{1} \vdash \mathbf{case}\, x^R\, (\langle\,\rangle \Rightarrow P) :: (z : C_r)}\ \mathbf{1}L_\alpha$$

$$\frac{\Gamma, w : A_m \vdash P :: (y : B_m)}{\Gamma \vdash \mathbf{case}\, x^W\, (\langle w, y \rangle \Rightarrow P) :: (x : A_m \multimap B_m)}\ \multimap R$$

$$\frac{}{\Gamma_W, w : A_m, x : A_m \multimap B_m \vdash x^R.\langle w, y \rangle :: (y : B_m)}\ \multimap L^0$$

$$\frac{}{\Gamma_W, w : A_m, y : B_m \vdash x^W.\langle w, y \rangle :: (x : A_m \otimes B_m)}\ \otimes R^0$$

$$\frac{\Gamma, (x : A_m \otimes B_m)^\alpha, w : A_m, y : B_m \vdash P :: (z : C_r)}{\Gamma, x : A_m \otimes B_m \vdash \mathbf{case}\, x^R\, (\langle w, y \rangle \Rightarrow P) :: (z : C_r)}\ \otimes L_\alpha$$

$$\frac{}{\Gamma_W, y : A_m \vdash x^W_k.\mathbf{shift}(y_m) :: (x : {\downarrow}^m_k A_m)}\ {\downarrow}R^0$$

$$\frac{\Gamma, (x : {\downarrow}^m_k A_m)^\alpha, y : A_m \vdash Q :: (z : C_r)}{\Gamma, x : {\downarrow}^m_k A_m \vdash \mathbf{case}\, x^R_k\, (\mathbf{shift}(y_m) \Rightarrow Q) :: (z :: C_r)}\ {\downarrow}L_\alpha$$

$$\frac{\Gamma \vdash P :: (y : A_k)}{\Gamma \vdash \mathbf{case}\, x^W_m\, (\mathbf{shift}(y_k) \Rightarrow P) :: (x : {\uparrow}^m_k A_k)}\ {\uparrow}R \qquad \frac{}{\Gamma_W, x : {\uparrow}^m_k A_k \vdash x^R_m.\mathbf{shift}(y_k) :: (y : A_k)}\ {\uparrow}L^0$$

Figure 4.3: Typing rules for shared memory. Note that other than the annotations $\cdot^{R/W}$ to indicate which memory accesses are reads and which are writes, they are identical to those for message passing.

$$
\begin{array}{ll}
\mathsf{thread}(c, x \leftarrow P \,;\, Q) \mapsto \mathsf{thread}(a, P[a/x]), \mathsf{cell}(a, \_), \mathsf{thread}(c, Q[a/x]) \ (a \text{ fresh}) & \textit{cut: allocate \& spawn} \\
!_m\mathsf{cell}(c_m, D), \mathsf{thread}(d_m, d_m \leftarrow c_m), \mathsf{cell}(d_m, \_) \mapsto !_m\mathsf{cell}(d_m, D) & \textit{id: move or copy} \\
\mathsf{thread}(c_m, c_m.V), \mathsf{cell}(c_m, \_) \mapsto !_m\mathsf{cell}(c_m, V) & (\oplus R^0, \otimes R^0, \mathbf{1}R^0, {\downarrow}R^0) \\
!_m\mathsf{cell}(c_m, V), \mathsf{thread}(e_k, \mathbf{case}\, c_m\, K) \mapsto \mathsf{thread}(e_k, V \triangleright K) & (\oplus L, \otimes L, \mathbf{1}L, {\downarrow}L) \\
\mathsf{thread}(c_m, \mathbf{case}\, c_m\, K), \mathsf{cell}(c_m, \_) \mapsto !_m\mathsf{cell}(c_m, K) & (\multimap R, \& R, {\uparrow}R) \\
!_m\mathsf{cell}(c_m, K), \mathsf{thread}(d_k, c_m.V) \mapsto \mathsf{thread}(d_k, V \triangleright K) & (\multimap L^0, \& L^0, {\uparrow}L^0) \\
\mathsf{thread}(a, a \leftarrow p\, \overline{b}) \mapsto \mathsf{thread}(a, P[a/x, \overline{b}/\overline{y}]) \quad (x \leftarrow p\, \overline{y} = P \in \Sigma) & (\mathsf{call})
\end{array}
$$

Figure 4.4: Reduction rules for shared memory

might allocate a fresh destination d_2, and evaluate e_2 with destination d_2, before then evaluating $[d_2/x]e_1$ with destination d. A concurrent version of those same semantics could evaluate e_2 and $[d_2/x]e_1$ concurrently, much as a cut in our language allocates a new memory address, spawns a new process which will write to that address (like evaluating e_2 with destination d_2), and continues running a process that may read from that address (as $[d_2/x]e_1$ may read from d_2, if it needs to access the value of x).

Using these objects, we can present the dynamic semantics of this shared-memory language, using multiset rewriting rules, as in the message-passing setting. The evaluation rules are given in fig. 4.4.

We begin by examining the left/right rules for connectives, which serve as the core means of communication. In general, right rules write (a convenient mnemonic), although what type of data they write (values or continuations) depends on the polarity of the type, with positive right rules writing values to memory, while negative right rules write continuations. By contrast, left rules read data from memory, combining that data with information provided in the process term to determine how to continue. A positive left rule, for instance, contains a continuation, and reads a value out of memory to pass to that continuation.

Unlike communication via channels, where messages are sent back and forth between processes, here, a thread terminates upon writing to memory, and all communication proceeds in a single direction. This unidirectional flow of information makes it easy to give sequential semantics for this language — in fact, we will see that this can be enforced purely at the level of scheduling, by restricting in what order we can apply the rules in fig. 4.4. As such, any program in this language can be run sequentially, yielding the same result as if it were allowed to run concurrently, with multiple threads active at once. [10]

We now examine the remaining rules of cut and identity, which do not deal directly with communication, but are nevertheless important.

The cut rule is quite straightforward, though it does do several things all at once. A thread executing the process term $x \leftarrow P \,;\, Q$ allocates a new memory cell with address a, spawns a new thread to run $P[a/x]$ with destination a, and continues running $Q[a/x]$, which is entitled to read from the cell at address a, once it has been filled in.

[10] Because this language satisfies a form of confluence (see Section 4.4.1), we can guarantee that this sequential schedule has the same result as any other scheduling of the same program.

In principle, the role of identity $d \leftarrow c$ is to ensure that a future thread looking for data at address d will find the data at address c. There are several ways we could go about this — perhaps the simplest would be to let the identity serve as a pointer or redirection, and when a thread attempts to read from d, it would instead be directed to c. However, this can be inconvenient if we have many identities, as read times can grow increasingly large due to a need to follow long chains of identities. Instead, we treat identity as a move or copy operation, reading the data from the cell at address c and writing it to address d. This is a move if the cell at address c is ephemeral, as the process of reading from that cell destroys it, and a copy otherwise.

Example 10 (Map, revisited)**.** *We have already seen an intuition for how a map process executes in the context of message-passing. Now, we will re-examine this same process in the shared-memory setting, seeing how our interpretation changes. Note that since typing is the same in both languages, we do not re-examine the typing of this process term, but we do annotate each memory access to indicate whether it is a read or a write.*

```
$\mathsf{list}_{A_k}, \uparrow_k^{\mathsf{U}} A_k \multimap B_k \vdash \mathit{map} :: \mathsf{list}_{B_k}$
$ys \leftarrow \mathit{map}\ xs\ f =$
$\mathsf{case}\ xs^R$  (  $\mathsf{nil}(u) \Rightarrow$
                $ys^W.\mathsf{nil}(u)$
             |  $\mathsf{cons}(p) \Rightarrow$
                $\mathsf{case}\ p^R$  (  $\langle x, xs' \rangle \Rightarrow$
                           $ys' \leftarrow (ys'^W \leftarrow \mathsf{map}\ xs'\ f)$
                           $f' \leftarrow f^R.\mathsf{shift}(f')$ ;
                           $y \leftarrow f'^R.\langle x, y \rangle$ ;
                           $p' \leftarrow (p'^W.\langle y, ys' \rangle)$
                           $ys^W.\mathsf{cons}(p')$
                        )
             )
```

In the context of shared memory, a process may read from the cells with addresses on the left of its typing judgment, and writes to the cell on the right of the typing judgment. For this particular process, that means that we expect to have cells with addresses xs and f, containing the start of a list and a (shifted) function to map over that list, and we intend to write our output list starting at address ys. We begin by reading from xs, here matching the message-passing interpretation, where we received a message along xs. In the nil case, we write $\mathsf{nil}(u)$ *to ys, reusing the memory cell at address u for our new list. In the cons case, we read out a pointer p to another cell, which we then read from to get pointers x and xs' to the head and tail of the list, respectively. The recursive call is also similar to the message-passing case, except that ys' is the address of a newly allocated memory cell, into which the recursive call should write its result, rather than a channel along which it will communicate. Our first major difference comes on the next line, where we construct f' by reading from the cell at address f. We allocate a cell at address f', and then read and execute a paused process from cell f, with destination f'. Once this process finishes executing, its result, of type $A_k \multimap B_k$, will be stored in f'. If k does not admit contraction, then this memory cell with address f' is ephemeral, while the cell at address f is persistent. By running the (persistent) process stored in f', we were able to create and fill a cell with a new copy of the function f' that we plan to map over the list. We can then read this function, stored* *as a paused process, from f', passing it x as an argument, and y as a destination, giving us a cell at address y which will (once this new process terminates) contain the result of applying the function to x. Finally, we write the pair $\langle y, ys' \rangle$ into a new cell at address p', constructing the body of a cons cell, which we then write into ys.*

4.4.1 Configuration Typing and Results

Extending typing for processes to configurations is quite straightforward in the shared-memory setting — we need only decide how to type cells. Since empty cells contain no information and always accompany a process that will write to them, we assign a type to the process and its cell together, ignoring the (invalid) cases where they occur separately. This also means, technically, that configurations where a process and its corresponding empty cell are not adjacent to each other are not well-typed, but in practice, this is little restriction — we can always ensure, when allocating a new cell, that it is placed next to the corresponding process. Cells that contain data are also easily handled: we treat a cell containing data D the same as the process that writes D to that cell and then terminates. Taken together, we get the following rules for typing configurations:

$$\frac{\Gamma_C, \Delta \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Delta \vdash P :: (a : A_m)}{\Gamma, \Gamma_C, \Delta \vDash \mathsf{thread}(a, P), \mathsf{cell}(a, \_) :: \Gamma, \Gamma_C, a : A_m} \ \mathsf{thread}$$

$$\frac{\Gamma_C, \Delta \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Delta \vDash a_m.V :: (a_m : A_m)}{\Gamma, \Gamma_C, \Delta \vDash !_m\mathsf{cell}(a_m, V) :: \Gamma, \Gamma_C, (a_m : A_m)} \ \mathsf{cell}^+$$

$$\frac{\Gamma_C, \Delta \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \Gamma_C, \Delta \vDash \mathsf{case}\, a_m\, K :: (a_m : A_m)}{\Gamma, \Gamma_C, \Delta \vDash !_m\mathsf{cell}(a_m, K) :: \Gamma, \Gamma_C, (a_m : A_m)} \ \mathsf{cell}^-$$

$$\frac{}{\Gamma \vDash \cdot :: \Gamma} \ \mathsf{empty} \qquad \frac{\Gamma_1 \vDash \mathcal{C}_1 :: \Gamma_2 \quad \Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3}{\Gamma_1 \vDash \mathcal{C}_1, \mathcal{C}_2 :: \Gamma_3} \ \mathsf{join}$$

As in the case of the message-passing semantics, because we allow an arbitrary Γ to pass through single-object configurations unchanged, we have a type extension result for the whole system, allowing us to pass such a Γ through any configuration. Likewise, we are also able to reassociate configurations in this setting without affecting their types.

Now that we have defined configuration typing, we are equipped to present the type safety result for this language, split into two parts, corresponding to progress and preservation, as is standard.

Theorem 21 (Preservation (shared-memory))**.** *If $\Gamma \vDash \mathcal{C} :: \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$, then $\Gamma \vDash \mathcal{C}' :: \Delta'$ for some $\Delta' \supseteq \Delta$.*

That is, taking a computation step may add new memory addresses, but any existing address continues to exist and retains the same type that it had before the step.

Proof. We examine several different cases, depending on which rule was used to step from $\mathcal{C}$ to $\mathcal{C}'$.

For the call and cut rules, and for the right rules, the left-hand side consists of either a single object, or a thread and its associated empty cell. In each case, we can reassociate $\mathcal{C}$ as $(\mathcal{C}_1, (\mathsf{thread}(a, \ldots), \mathsf{cell}(a, \_))), \mathcal{C}_2$, where the thread being focused on is the one that appears on the left of the rule. We then replace this thread (and potentially its cell) with the right-hand side of the rule, and so it will suffice to check that both sides of the rule (possibly with an extra empty cell added to ensure that each thread has its corresponding cell) have the same type as configurations. For both right rules, this is immediate from examining the thread and $\mathsf{cell}^{\pm}$ typing rules. Likewise, for the call rule, we can see that the process terms on the left and on the right have the same type (using the call typing rule from Section 4.2.6), and therefore the threads containing these terms have the same type as configurations. Cut is slightly more involved, as it produces two threads from one, but as in the message-passing setting, we can use the premises of the cut that types the initial process term $x \leftarrow P \; ; Q$ to show that each of the new threads is well-typed, and that they can be joined together to give a configuration with the same type as the original thread.

The identity rule and the left rules deal with reading memory, and so each involves both a filled cell and a thread attempting to read from that cell. Since the typing rules enforce that the provider of a symbol (or here, an address) must occur to the left of any of the clients of that symbol (readers of that address), we can conclude that the cell being read from is to the left of the thread reading it. We now follow a similar approach to that used in the message-passing setting, beginning by reassociating $\mathcal{C}$ as $\mathcal{C}_1, !_m\mathsf{cell}(c_m, D)\mathcal{C}_2, \mathsf{thread}(d_k, P), \mathcal{C}_3$. Inversion on the typing derivation for $\mathcal{C}$ then gives us that there exist $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ such that

- $\Gamma \vDash \mathcal{C}_1 :: \Gamma_1$
- $\Gamma_1 \vDash !_m\mathsf{cell}(c_m, D) :: \Gamma_2$
- $\Gamma_2 \vDash \mathcal{C}_2 :: \Gamma_3$
- $\Gamma_3 \vDash \mathsf{thread}(d_k, P) :: \Gamma_4$
- $\Gamma_4 \vDash \mathcal{C}_2 :: \Delta$

Further applying inversion on the typing of the cell and thread, we get that Γ_2 and Γ_3 must both contain $(c_m : A_m)$ for some type A_m. Since addresses are unique in a configuration, this A_m must be the same in both places.

Now, as in the shared-memory case, we define contexts Γ_2'', Γ_3''. Observe that there is some minimal subset Δ_D of Γ_1 such that $\Delta_D \vDash !_m\mathsf{cell}(c_m, D) :: (c_m : A_m)$, consisting exactly of the symbols that occur free in D. Write $\Gamma_2 = \Gamma_2', (c_m : A_m)$, and let $\Gamma_2'' = \Gamma_2', (c_m : A_m)^\beta, \Delta_D$. Likewise, write $\Gamma_3 = \Gamma_3', (c_m : A_m)$, and let $\Gamma_3'' = \Gamma_3', (c_m : A_m)^\beta, \Delta_D$.

We then make the following three claims:

(1) There is $\hat{\Gamma}$ such that $\Gamma_1 \vDash C_m?(!_m\mathsf{cell}(c_m, D), \cdot) :: \Gamma_2'', \hat{\Gamma}$, where $C_m?(x, y)$ denotes x if m admits contraction and y otherwise.

(2) $\Gamma_2'' \vDash \mathcal{C}_2 :: \Gamma_3''$

(3) $\Gamma_3'' \vDash \mathsf{thread}(d_k, P') :: \Gamma_4'$, where P' is the process that results from P taking a step to read from c, and $\Gamma_4' \supseteq \Gamma_4$.

As in the message-passing case, (1)-(3) can be used, with type extension as necessary, to give the overall result.

We now prove (1)-(3) in the shared-memory setting, noting that each is quite similar to its message-passing equivalent. We work through the details where they differ, and otherwise refer back to the proof in Section 4.4.1.

Claim (1) By inversion, the cell must have a typing derivation of the following form:

$$\frac{\Delta_C, \Delta_1 \geq m \quad \mathsf{C} \in \sigma(\Delta_C) \quad \Delta_C, \Delta_1 \vdash P_D :: (c : A_m)}{\Delta_C, \Delta_1, \Delta_2 \vDash !_m\mathsf{cell}(c_m, D) :: \Delta_C, \Delta_2, (c : A_m)}\ \mathsf{cell}^{\pm}$$

where P_D is the process that writes D to c — either $c.V$ or $\mathsf{case}\ c\ K$, depending on whether D is a value V or continuation K. We also know that $\Delta_D \subseteq \Delta_C, \Delta_1$, from our choice of Δ_D. This also means that $\Delta_D \geq m$, and so if $\mathsf{C} \in \sigma(m)$, then also $\mathsf{C} \in \sigma(\Delta_D)$. Now, we distinguish cases based on whether m admits contraction or not.

If m admits contraction, then we take $\Delta'_C = \Delta_C, \Delta_D$ and $\Delta'_1 = \Delta_1 \setminus \Delta_D$. We then have that $\Delta'_C, \Delta'_1 = \Delta_C, \Delta_1$, and $\Delta'_C, \Delta_2, (c : A_m) = \Gamma''_2$, as it is formed by adding Δ_D to Γ_2, and, since m admits contraction, we are in the case where $\beta = 1$, so Γ''_2 contains $(c : A_m)$. The following typing derivation then gives the desired result:

$$\frac{\Delta'_C, \Delta'_1 \geq m \quad \mathsf{C} \in \sigma(\Delta'_C) \quad \Delta'_C, \Delta'_1 \vDash P_D :: (c : A_m)}{\Delta'_C, \Delta'_1, \Delta_2 \vDash !_m\mathsf{cell}(c_m, D) :: \Delta'_C, \Delta_2, (c : A_m)}\ \mathsf{cell}^{\pm}$$

If m does not admit contraction, then we observe that $\Gamma_1 = \Delta_C, \Delta_1, \Delta_2$, which already contains Δ_D, and that $\Gamma_2 = \Delta_C, \Delta_2, (c : A_m)$, so $\Gamma''_2 = \Delta_C, \Delta_2, \Delta_D$. Taking $\hat{\Gamma} = \Delta_1 \setminus \Delta_D$, we get that $\Gamma_1 = \Gamma''_2, \hat{\Gamma}$, and so the empty rule gives the desired result.

Claim (2) As in the message-passing setting, this follows from an induction over typing derivations, allowing us to replace $(c : A_m)$ with Δ_D on both sides of any typing derivation. The only real difference that arises is that, in the shared-memory setting, no lemma is needed to get that every address has a unique provider — this follows from our constraint on well-formedness of configurations.

Claim (3) Now, apply inversion to the typing derivation of the thread $\mathsf{thread}(d_k, P)$ which reads from c. Here, there are different cases, depending on which rule P uses to read, but all behave similarly. We will consider the case of $\&L^0$ using label i. In this case, $k = m$ and the typing derivation has the form

$$\frac{\Delta_C, \Delta_1 \geq m \quad \mathsf{C} \in \sigma(\Delta_C) \quad \dfrac{(i \in L)}{\Delta_C, \Delta_1 \vDash c.i(d) :: (d : D_m)}\ \&L^0}{\Delta_C, \Delta_1, \Delta_2 \vDash \mathsf{thread}(d_m, c.i(d)) :: \Delta_C, \Delta_2, (d : D_m)}\ \mathsf{thread}$$

where $\Delta_C, \Delta_1, \Delta_2 = \Gamma_3$ and $\Delta_C, \Delta_2, (d : D_m) = \Gamma_4$. We also have that Δ_C, Δ_1 contains $(c : \&\{\ell : A^{\ell}_m\}_{\ell \in L})$, along with possibly some weakenable portion. In this case, we can also get, by inversion on the typing derivation for the cell, that $D = (\ell(x) \Rightarrow P_\ell)_{\ell \in L}$ and that $\Delta_D \vdash P_i :: (x : A^i_m)$.

Let Δ'_C and Δ'_1 be the result of removing c from Δ_C and Δ_1, respectively. Then, define $\Delta''_C = \Delta_D \cap \Delta'_C$ and $\Delta''_1 = \Delta_D \cap \Delta'_1$ and $\Delta'_2 = \Delta_2, (\Delta_C \setminus \Delta_D), (\Delta_1 \setminus \Delta_D), (c : \&\{\ell : A^{\ell}_m\}_{\ell \in L})^{\beta}$.

Observe that $\Delta''_C, \Delta''_1 = \Delta_D$, that $\Delta''_C, \Delta''_1, \Delta'_2 = \Gamma''_3$, and that $\Delta''_C, \Delta'_2 \supseteq \Delta_C, \Delta_2$ (if c occurred in Δ_C, then m admits contraction, and so c also occurs in Δ'_2).

Now, we construct the following typing derivation:

$$\dfrac{\Delta''_C, \Delta''_1 \geq m \quad \mathsf{C} \in \sigma(\Delta''_C) \quad \Delta''_C, \Delta''_1 \vDash P_i[d/x] :: (d : D_m)}{\Delta''_C, \Delta''_1, \Delta'_2 \vDash \mathsf{thread}(d_m, P_i[d/x]) :: \Delta''_C, \Delta'_2, (d : D_m)}\ \mathsf{thread}$$

where the premise of this derivation comes from substituting d for x in the typing derivation for the corresponding cell. Since $\Delta''_C \subseteq \Delta_C$ and $\Delta''_1 \subseteq \Delta_1$, both side conditions for this rule are also satisfied. Now, as $\Delta''_C, \Delta''_1, \Delta'_2 = \Gamma''_3$ and $\Delta''_C, \Delta'_2, (d : D_m) \supseteq \Delta_C, \Delta_2, (d : D_m) = \Gamma_4$, this case is complete. To justify our use of type extension, we also note that Δ''_C, Δ'_2 extends Δ_C, Δ_2 with, at most, some weakenable portion of Δ'_1 and $(c : \&\{\ell : A^\ell_m\}_{\ell \in L})$. Since these symbols do not occur in Γ_4, well-typedness of $\mathcal{C}$ ensures that they cannot occur in $\mathcal{C}_3$, and so are eligible for type extension.

As in the message-passing setting, the remaining communication cases are similar — indeed, here, we are able to prove them more uniformly, only needing to resort to cases to prove (3). □

While this is slightly weaker than the standard preservation theorem, as a configuration may gain new addresses that it provides via allocation, it is nevertheless strong enough to, along with progress, give us type safety in the sense that well-typed configurations cannot get stuck — indeed, all that we need of preservation for that result is that a well-typed configuration remains well-typed after taking a computation step.

As in the message-passing case, our notion of progress is slightly different from the standard functional notion — the role that values play in a functional language is taken on by filled memory cells (or, rather, by configurations consisting only of filled memory cells). Like functional values, these configurations cannot take any further steps, and so serve as the natural end state of computation.

Definition 3. *If a configuration $\mathcal{C}$ consists entirely of filled memory cells (that is, objects of the form $!_m\mathsf{cell}(a_m, D)$), we say that $\mathcal{C}$ is* final. *We will often suggestively write $\mathcal{F}$ for final configurations.*

Final configurations, similar to poised configurations in the message-passing setting, are unable to take steps on their own, though external input may enable them to be further evaluated. In the message-passing setting, this input comes in the form of an external message or service interacting on the same channel as some object in the configuration, while in the shared-memory setting, it always consists of some process reading from a cell in the configuration. With the concept of final configurations, we can give our progress theorem for shared memory, analogous to those for message passing, or in the usual functional setting.

Theorem 22 (Progress (shared-memory))**.** *If $(\cdot) \vDash \mathcal{C} :: \Delta$, then one of the following holds:*

1. *There is some $\mathcal{C}'$ such that $\mathcal{C} \mapsto \mathcal{C}'$.*
2. *$\mathcal{C}$ is a final configuration.*

Proof. If $\mathcal{C}$ is not final, then it contains some $\mathsf{thread}(a_m, P)$. Associate $\mathcal{C} = \mathcal{C}_1, \mathsf{thread}(a_m, P), \mathcal{C}_2$, where $\mathcal{C}_1$ is final (that is, select out the leftmost thread in $\mathcal{C}$).

Now, there are several cases depending on what P is. If P ends in a cut, a call, or a right rule, then $\mathsf{thread}(a_m, P)$ can take a step on its own (potentially also using its associated empty cell), and so $\mathcal{C}$ can take a step using the same rule, applied to this thread. Otherwise, P ends in a left rule or an identity (and so, as we will see, needs to read from some memory cell at address d_k).

If we apply inversion to the typing derivation for $\mathcal{C}$, we get that there are Δ', Δ'', and A_m such that $(\cdot) \vDash \mathcal{C}_1 :: \Delta'$ and $\Delta' \vDash \mathsf{thread}(a, P), \mathsf{cell}(a, \_) :: \Delta'', (a : A_m)$.

Now, we examine exactly which rule P ends in. All of the remaining cases are similar, and so we show the case of $\otimes L$ as an example. In this case, the typing derivation for $\mathsf{thread}(a, P)$ has the following form:

$$\dfrac{\Gamma_C, \Gamma_1 \geq m \quad \mathsf{C} \in \sigma(\Gamma_C) \quad \dfrac{\Gamma'_C, \Gamma'_1, (d : B_k \otimes C_k)^\alpha, (b : B_k), (c : C_k) \vdash Q[b/x, c/y] :: (a : A_m)}{\Gamma'_C, \Gamma'_1, (d : B_k \otimes C_k) \vdash \mathsf{case}\ c\ (\langle x, y\rangle \Rightarrow Q) :: (a : A_m)} \otimes L_\alpha}{\Gamma_C, \Gamma_1, \Gamma_2 \vDash \mathsf{thread}(a, P), \mathsf{cell}(a, \_) :: \Gamma_C, \Gamma_2, (a : A_m)} \,\mathsf{thread}$$

Here, $\Delta' = \Gamma_C, \Gamma_1, \Gamma_2$ and $\Gamma'_C, \Gamma'_1, (d : B_k \otimes C_k) = \Gamma_C, \Gamma_1$ — d occurs in one of Γ_C and Γ_1, but not both.

We see now that P is attempting to read from address d, and we also get that Δ' contains $(d : B_k \otimes C_k)$ — inversion in the other cases will give similar results, with P attempting to read from some address that occurs in Δ', and whose type in Δ' matches what P expects to read.

Now, we claim that $\mathcal{C}_1$ contains a corresponding $!_k\mathsf{cell}(d_k, D)$. This follows by induction on the typing derivation of $\mathcal{C}_1$, in a similar manner to Lemma 6 in the message-passing case, using the fact that $\mathcal{C}_1$ is final to get that the provider of d_k must be a cell. In the particular case we are considering, where $d : B_k \otimes C_k$, we apply inversion to the typing derivation for this cell, seeing that only the $\otimes R^0$ rule can type a cell providing this type. As such, this cell must have the form $!_k\mathsf{cell}(d_k, \langle b, c\rangle)$ for some b, c. This then enables us to apply the $\otimes L$ rule to step the configuration and make progress.

Likewise, in the other cases, once we find that a cell exists with the correct type, we can apply inversion to get that the cell contents have the correct shape, allowing us to apply the corresponding computation rule, either one of the left rules or the identity rule. The only major difference is that the identity rule does not care what shape the data stored in the cell has, so we can skip the last inversion step in the identity case. □

In addition to these type-safety properties, we also have a confluence result, for which we need to define a weak notion of equivalence on configurations to account for the fact that fresh addresses are created by some steps, and their names are not prescribed by the semantics. We say $\mathcal{C}_1 \sim \mathcal{C}_2$ if there is a renaming ρ of addresses such that $\rho\mathcal{C}_1 = \mathcal{C}_2$. We can then establish the following version of the diamond property:

Theorem 23 (Diamond Property (shared-memory))**.** *Assume $\Delta \vdash \mathcal{C} :: \Gamma$. If $\mathcal{C} \mapsto \mathcal{C}_1$ and $\mathcal{C} \mapsto \mathcal{C}_2$ such that $\mathcal{C}_1 \not\sim \mathcal{C}_2$. Then there exist $\mathcal{C}'_1$ and $\mathcal{C}'_2$ such that $\mathcal{C}_1 \mapsto \mathcal{C}'_1$ and $\mathcal{C}_2 \mapsto \mathcal{C}'_2$ with $\mathcal{C}'_1 \sim \mathcal{C}'_2$.*

Proof. As in the message-passing setting, we can see that at most one computation rule applies to a given thread at any given time, and so there can be no interference between rules where threads are concerned. The only way two rules can apply to the same object is if two different threads are attempting to read from the same cell, but in this case, since $\mathcal{C}$ is well-typed (and, by preservation, so are $\mathcal{C}_1, \mathcal{C}_2$), the mode of that cell's address must admit contraction, since the cell has multiple readers, and so the cell is persistent, and can be read from in either order, yielding identical configurations. In all other cases, because the rules used to produce $\mathcal{C}_1$ and $\mathcal{C}_2$ are not acting on the same objects, we can likewise apply the rules in either order to get identical resulting configurations. □

In the fragment of the language without recursion, this gives a full confluence result via standard inductions, but even with recursion, we continue to have this form of local confluence.

Finally, while not a property of the full language, we can prove termination for the recursion-free fragment of the language. In a sequential setting, this is perhaps not so interesting, but in the concurrent setting, this shows that not only do we avoid deadlock (as evidenced by the progress theorem), we also avoid so-called *livelock*, where processes are able to take steps, but the overall system is still stuck.

Theorem 24. *Suppose* $\cdot \vdash \mathcal{C} :: \Gamma$*, and* $\mathcal{C}$ *is a configuration in the recursion-free fragment of the language (formally, this can be enforced by requiring that it is well-typed with respect to an empty signature of process and type definitions). Then, there is some final configuration* $\mathcal{F}$ *such that* $\mathcal{C} \mapsto^* \mathcal{F}$*. Note that by the diamond property above, this configuration* $\mathcal{F}$ *is, in fact, unique (up to renaming of addresses, particularly to handle addresses that are freshly allocated during the computation).*

A proof of this theorem in the purely structural case can be found in [26], and extends naturally to the adjoint case, with slightly more bookkeeping about which cells are persistent.

4.4.2 Sequentiality

The languages we have examined thus far are both highly concurrent — any time we want to compose two computations, we have only one tool to do it: the cut rule, which, regardless of our exact choice of semantics, allows the two computations to run concurrently. While this is often convenient, and gives quite a bit of freedom to implement a scheduler for running processes, there are several reasons that we may want to compose two computations *sequentially* instead. Many existing programming languages are purely or primarily sequential, and so to model them or their features accurately requires the ability to run programs (or parts of programs) sequentially. Somewhat more subtly, while in theory, maximizing concurrency should be highly efficient, as many pieces of a larger computation can run at once, in practice, there is overhead to spawning a new process or thread, as well as overhead in scheduling whenever the number of threads exceeds the number of processors. This inefficiency is most pronounced when the work done by each thread is relatively small, as this means that the overhead needed to create a new thread is a larger portion of the total work needed by the thread. Unfortunately, in the shared-memory language we define, almost every thread is short-lived, yielding a near-worst-case scenario for concurrency overhead.

One interesting observation is that our shared-memory language (section 4.4), while naturally concurrent, is already easy to schedule in a sequential manner. Because the flow of information in communication steps is unidirectional, with a process terminating after it writes to memory (so it cannot then read from memory at a later point), we can achieve sequential computation by ensuring that we always schedule the writer of a cell before any readers of the cell. This is easily accomplished by running the left-hand side of a cut before the right-hand side, and will give us a sequential computation similar to a *destination-passing style* [53, 107] implementation of a functional language, where values are stored in *destinations*, rather than in variables.

Sequentiality Primitives

As a result, it is most natural for our exploration of sequentiality to begin with the shared-memory language, where we have some intuition to fall back on for how sequential composition should behave. However, our language is not expressive enough as is to enforce sequentiality — each of our atomic computations is quite small, and when we compose two of them, we get no guarantee of what order they are executed in. There are several possible additions to the language that allow us to recover sequential composition, each of which appears to have the same expressive power as the others. We will examine a few of these in turn, before selecting one to focus on.

Most obviously, we can implement sequential composition by creating a new primitive operation for it. A sequential cut $x \overset{\mathsf{seq}}{\leftarrow} P \ ; \ Q$ should be typed the same way as the ordinary cut — we still need both P and Q to be well-typed, and for the result of P to be useable by Q at matching types. The only difference, then, comes in the dynamic semantics, where we would like to say that P must fully execute, storing its result in x, before Q is allowed to begin running. Within our system of semantics, there is no clear way to do this — threads are always able to run unless blocked on input, and without knowing more information about the type of x, we have no way to block Q waiting for x. As such, this approach is unsuitable unless we somehow modify our dynamic semantics, perhaps by adding additional semantic objects to represent paused or blocked threads.

If we consider the case where $x : \mathbf{1}$ as an example, we can in this case block Q on x, by defining

$$x \overset{\mathsf{seq}}{\leftarrow} P \ ; Q \triangleq x \leftarrow P \ ; \mathsf{case}\ x\ (\langle\rangle \Rightarrow Q).$$

It is likewise possible to block Q on x as long as $x : A$ for some positive type A $(\mathbf{1}, \otimes, \oplus, \downarrow)$, using much the same construction. For negative types, however, this construction is not possible, as reading from a negatively typed x is a terminal operation. We may then consider several possible ways to extend this construction to negative types.

One option is to force a positive type by adding an extra connective without any logical content, such as $\downarrow_m^m$, to the type of x. If, for instance, $x : A_m$, we let $y : \downarrow_m^m A_m$, and define

$$x \overset{\mathsf{seq}}{\leftarrow} P \ ; Q \triangleq y \leftarrow (x \leftarrow P \ ; y.\mathsf{shift}(x)) \ ; \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow Q).$$

While this does allow Q to be blocked on y, it only defers the problem, as $x \leftarrow P \ ; \ y.\mathsf{shift}(x)$ can still write to y in two steps, first executing the cut and then writing to y. In order for this approach to work, we need some way to enforce that y is only written to if x is already written to. One means of accomplishing this is to add the ability to atomically write to multiple locations at once. With a new construct $y.\mathsf{shift}(x.V)$, we atomically write V to x and $\mathsf{shift}(x)$ to y. Of course, this construct only works for positively-typed x as written, but we can provide similar constructs for negatively-typed x. With such a construct, instead of cutting together P with a write to y, we replace all writes to x in P with atomic writes to both x and y. This substitution operation is somewhat involved, however, which makes this approach inconvenient, despite the relative simplicity of the new construct and the fact that its dynamic semantics do not require any new semantic objects.

Another approach, likewise building on the example for positively-typed x, is to add a new *blocking read* construct $\mathsf{case}\ x\ (y \Rightarrow Q)$, which attempts to read x, blocks until it succeeds, and then binds the result of the read as y in Q. This serves the same purpose as the read from x in the case of $\mathbf{1}$ or other positive types, but is agnostic to the type of x. We can think of this alternately as a *blocking identity* — it behaves almost identically to the process $y \leftarrow (y \leftarrow x) \; ; \; Q$, except that the latter allows Q to run before x has been read from and y written to. With this construct, we can define

$$x \overset{\mathsf{seq}}{\leftarrow} P \; ; \; Q \triangleq x \leftarrow P \; ; \; \mathsf{case}\ x\ (y \Rightarrow Q[y/x])$$

This definition, finally, is implementable without needing to make any changes to the semantics, with only a single new construct, and without the need to make any substitutions more complicated than variable-for-variable, and so is the one we will focus on throughout this section.

With an intuition for the construct we wish to add to the language to model sequential computation, we now need to formally define this construct. The typing rule for this blocking read or identity is straightforward:

$$\frac{\Gamma, (x : A_m)^{\alpha} y : A_m \vdash Q :: (z : C_k)}{\Gamma, x : A_m \vdash \mathsf{case}\ x\ (y \Rightarrow Q) :: (z : C_k)}\ \mathsf{id\_block}^{\alpha}$$

Note that as with many other rules, we use an index α to combine two versions of the same rule, where the $\alpha = 1$ case requires m to admit contraction, and allows for x to be reused. For this particular rule, it will actually suffice to only take $\alpha = 0$, as any reuse of x within Q corresponds exactly to a use of y (since x must already have been written to in order for Q to execute), which has the same mode, and so is reuseable if x is.

Indeed, this is the same typing that we expect from the intuition that this blocking identity should behave the same as $y \leftarrow x \; ; \; Q$, which has the following typing derivation

$$\frac{(x : A_m \geq m \geq k) \qquad \dfrac{}{x : A_m \vdash y \leftarrow x :: (y : A_m)}\ \mathsf{id} \qquad \Gamma, y : A_m \vdash Q :: (z : C_k)}{\Gamma, x : A_m \vdash y \leftarrow (y \leftarrow x) \; ; \; Q}\ \mathsf{cut}$$

in which the conditions on modes are always satisfied when the conclusion of the derivation is well-formed. The evaluation rule of this construct is likewise straightforward:

$$!_m\mathsf{cell}(a, D), \mathsf{thread}(c, \mathsf{case}\ a\ (y \Rightarrow Q)) \mapsto \mathsf{thread}(c, Q[a/y])$$

A blocking read can only continue to execute the process within when the cell it is trying to read from has been filled, and redirects the process to the correct address to load information from that cell. Note that unlike an identity, this does not perform any move or copy operation, and so does not need an extra cell.

Because this is expressed using the same system of semantics as the core concurrent language, any reasoning tools that we develop (in particular the theory of equivalence we present in chapter 5) apply not only to concurrent programs, but also to sequential ones, and to mixed concurrent-sequential programs. This also allows for the possibility to prove concurrent programs equivalent to sequential ones, perhaps justifying some more efficient concurrent implementation's correctness by reference to a simpler sequential one.

It may also be possible to restrict concurrency by mode — from a purely scheduling perspective, certainly we can choose to schedule processes of mode m sequentially, while processes of mode k are scheduled concurrently, so that a cut creating a fresh address of mode k is treated as a concurrent cut, and a cut creating a fresh address of mode m is treated sequentially. By restricting what constructs are allowed at a given mode, we can also have mode-dependent concurrency even without fine-grained control over scheduling (provided that we have one of the several mentioned new constructs for implementing sequential cuts). If a mode m is restricted so that the only cuts allowed at mode m (interpreted either to mean cuts that create a fresh address of mode m, or cuts that take place within a process whose overall mode is m) are of the form given for sequential cuts, then a process at mode m will always run sequentially. However, by using shifts to a different mode k, which does not have this restriction (or, if we want to enforce that *all* computations at mode k are concurrent, which has the different restriction that no blocking identities are allowed at mode k), a computation at mode m may be able to nevertheless contain concurrent subcomputations. Depending on the relationship between m and k, we could allow both concurrent and sequential computations to depend on each other, with $m \leq k$ and $k \leq m$, using the modes only to distinguish at the type level which computations are concurrent, or we could restrict dependence by taking either $m < k$ or $k < m$, giving two different systems where either sequential computations may not depend on the result of concurrent ones, or vice versa. [11]

Example 11 (Map, sequentially)**.** *As a final re-examining of our running example of map, we will look first at a purely sequential version of map, to illustrate that (at least with the syntactic sugar of $x \overset{\mathsf{seq}}{\leftarrow} P\ ;\ Q$), very little changes in how the program is represented, making it very easy to go between sequential and concurrent computation. We will then see how we can write a mixed concurrent-sequential version of map, where we get the full benefits of being able to compute the function f on several different list elements concurrently, without needing to spawn a new process for each small computation step.*

A purely sequential map looks nearly identical to the purely concurrent version, except that each cut has been replaced by a sequential cut:

```
$\mathsf{list}_{A_k}, \uparrow_k^{\mathsf{U}} A_k \multimap B_k \vdash \mathit{map} :: \mathsf{list}_{B_k}$
$ys \leftarrow \mathit{map}\ xs\ f =$
$\mathsf{case}\ xs^R$  (  $\mathsf{nil}(u) \Rightarrow$
                $ys^W.\mathsf{nil}(u)$
             |  $\mathsf{cons}(p) \Rightarrow$
                $\mathsf{case}\ p^R$  (  $\langle x, xs' \rangle \Rightarrow$
                          $ys' \overset{\mathsf{seq}}{\leftarrow} (ys'^W \leftarrow \mathsf{map}\ xs'\ f)$
                          $f' \overset{\mathsf{seq}}{\leftarrow} f^R.\mathsf{shift}(f')\ ;$
                          $y \overset{\mathsf{seq}}{\leftarrow} f'^R.\langle x, y \rangle\ ;$
                          $p' \overset{\mathsf{seq}}{\leftarrow} (p'^W.\langle y, ys' \rangle)$
                          $ys^W.\mathsf{cons}(p')$
                       )
             )
```

When run on a non-empty list, this first runs the recursive call on the tail of the list to completion, builds the function f', calls f', and builds the pair p', before finally writing to ys. As such,

[11] The fourth option, where k and m are incomparable, is of course also possible, but less interesting. In such a system, we can write both purely sequential and purely concurrent programs, but neither can depend on the other, and they can never interact.

any client of ys can be guaranteed that the entire list will be available as soon as a read from ys succeeds, and we avoid the overhead of creating new threads for brief computations — in particular, the creation of p' is guaranteed to take only one step, but creating f' from f may also take only a single step, depending on what exactly f is. This illustrates that in some cases (creating p'), we can be assured that a given computation is short, and so is a good candidate for a sequential cut, but for others (e.g. computing f' from f, or running f on x), it is less clear how long the computation will take, and whether it is worthwhile to spawn a new thread. As such, allowing the programmer to specify cuts as sequential if their intended use case is for a short computation may allow for better optimization than automated methods can do, due to their lack of knowledge about the runtime of certain parts of a process.

If our primary goal is to allow the function stored in f to be run concurrently on each element of a list, we can achieve this by changing a single one of the sequential cuts in the above example into a concurrent cut.

```
$\mathsf{list}_{A_k}, \uparrow^{\mathsf{U}}_{k} A_k \multimap B_k \vdash \mathit{map} :: \mathsf{list}_{B_k}$
$ys \leftarrow \mathit{map}\ xs\ f =$
$\mathsf{case}\ xs^R$  (  $\mathsf{nil}(u) \Rightarrow$
                $ys^W.\mathsf{nil}(u)$
             |  $\mathsf{cons}(p) \Rightarrow$
                $\mathsf{case}\ p^R$  (  $\langle x, xs' \rangle \Rightarrow$
                          $ys' \leftarrow (ys'^W \leftarrow \mathsf{map}\ xs'\ f)$
                          $f' \overset{\mathsf{seq}}{\leftarrow} f^R.\mathsf{shift}(f')$ ;
                          $y \overset{\mathsf{seq}}{\leftarrow} f'^R.\langle x, y \rangle$ ;
                          $p' \overset{\mathsf{seq}}{\leftarrow} (p'^W.\langle y, ys' \rangle)$
                          $ys^W.\mathsf{cons}(p')$
                       )
             )
```

We have highlighted this cut (used to create ys' from a recursive call on xs') in red. If the recursive call is allowed to make progress at the same time as the remainder of the program, we can go on to evaluate f to give f', and evaluate f' on x to give y, while the recursive call runs, spawning further new processes for further recursive calls, and continuing to compute f on subsequent elements of the list. By using a mix of sequential and concurrent cuts, we can control the concurrency behavior of a program on a fine-grained level

Call-by-Name

All of the above approaches to defining a sequential cut $x \overset{\mathsf{seq}}{\leftarrow} P$; Q seek to evaluate the left-hand side of the cut fully before the right-hand side, giving what is usually thought of as a sequential composition of processes. However, we could instead take a different approach to sequential programming, following the principles of a *call-by-name* semantics for functional programming. Rather than evaluating the left-hand side of a cut before the right-hand side, we would instead begin evaluating the right-hand side, and then, when the result of the left-hand side would be used, pausing the current process and only then running the left-hand side to get its result. This form of composition would not be sequential in the usual sense, but it still enforces that only one thread is able to make progress at a time, which is also reasonable to call a form of sequentiality.

Interestingly, this form of "call-by-name" cut, which we will write $x \overset{\mathsf{cbn}}{\leftarrow} P \mathbin{;} Q$, can be implemented in the base language without any additional features. The first piece of implementing $x \overset{\mathsf{cbn}}{\leftarrow} P \mathbin{;} Q$ is to ensure that P is blocked and cannot run until some later point. We can achieve this by putting P inside a continuation $\mathsf{case}\, y\, (\mathsf{shift}(x) \Rightarrow P)$. The cut $y \leftarrow \mathsf{case}\, y\, (\mathsf{shift}(x) \Rightarrow P) \mathbin{;} Q$ can take a single step on the left, to write the continuation to y, but P is not able to execute until y is read from. We then need to modify Q so that before reading from x, it first must read from y, allowing P to begin running (while Q is blocked waiting for x to be written to). For instance, if x is read via a term $x.V$, we can replace this with $x \leftarrow y.\mathsf{shift}(x) \mathbin{;} x.V$. While this uses a concurrent cut, the right-hand side of the cut is blocked waiting for x to be written to, which can only happen after the left-hand side of the cut runs, allowing P to execute. Other forms of read from x can be modified similarly to first require a read from y. Because y contains a continuation that needs to be run in order to yield the result x, and y is read from each time the original process Q would read from x, x is recomputed via P each time it is used, making this an implementation of call-by-name, rather than call-by-need, where only the first access to x would require P to be executed, with subsequent accesses looking up the result of this first computation. By contrast, call-by-need, where P is only executed the first time it is needed, is achievable via scheduling, but we cannot write a program that enforces a call-by-need schedule without some additional language construct, allowing P to be blocked from executing without wrapping it in a continuation that is re-executed each time it is read from. [12]

We will not make significant use of this form of sequential cut, as the need to replace all reads from x in Q means that some significant substitution is required to translate it back into the base language, but it is nevertheless interesting that this behavior can already be expressed without any changes to the language.

Futures

Now that we are thinking in a sequential context, it makes sense to consider how our language relates to other approaches to mixing sequential and concurrent computation. Many approaches exist for adding concurrency features to an otherwise sequential language, ranging from threading libraries common in C-like languages to the fork/join composition of tasks often used in algorithm design. Futures [32, 39] (or promises) are one such approach, in which a new thread is spawned to execute some computation, giving a reference to the original thread that can be used to access the result of the computation once it is done. This reference, called a future, can be passed around and treated as any other value, and may either be explicitly read from (via an operation often called *touch*) or implicitly read from when its contents are accessed. This behavior is almost identical to that of the standard concurrent cut in our language — a cut $x \leftarrow P \mathbin{;} Q$ spawns a new thread to execute P, storing its result into x, and allows Q to continue to run up until it needs to access x. We can therefore think of the concurrent cut as a form of future, and the purely concurrent language as one where all computations are wrapped in futures, allowed to

[12] Interestingly, a call-by-need scheduling, because it only runs threads as they are needed, can terminate without reaching a final configuration, if some thread is spawned, but its result never used. Of course, for this to happen, the thread must be providing an address that admits weakening.

run concurrently.

A natural question at this point, then, is what role the sequential cut plays. If we restrict the language so that only sequential cuts may be used, computation proceeds purely sequentially, and we have, in essence, recreated a functional programming language. Adding concurrent cuts back in follows the common pattern for semantics for futures (and, indeed, for many other concurrency primitives), starting with a well-understood sequential language (e.g. some form of lambda calculus), and then augmenting it with a new feature that enables concurrent computation. As a result, in this usual setting, the concurrent portion of the language is often handled in an ad hoc manner, with its own semantics that largely do not interact with the core sequential language. Our system, while similar, follows the reverse pattern, in that our core semantics handle the concurrent computation, and it is sequentiality that requires additions. However, no changes to the core semantics of the language are needed — only new syntax and corresponding rules for typing and evaluation, while the semantic objects in use stay unchanged. In the case of the call-by-name cuts, we can achieve a form of sequentiality without even needing new syntax, other than as a convenience. As a result, our system as a whole makes uniform reasoning about mixed sequential-concurrent computations not only feasible, but comparably easy to reasoning about computations in the purely concurrent base language.

In this sense, our language gives a first *uniform* system of semantics for futures. An interesting addition is that, since a cut can be defined at any mode, we can define not only usual futures, but various substructural futures, which can be read from exactly once (linear), at least once (strict), or at most once (affine), none of which had been formalized prior to this work. We are unaware of any use cases for strict or affine futures at the moment, but linear futures have been used in algorithm design in the past (e.g. [10]), where they can lead to not only constant gains in performance, but even asymptotic improvements.

Chapter 5

Program Equivalence

5.1 Introduction and Background

In addition to having a language with desirable properties as a whole (such as type safety), we often would like to be able to reason about specific programs in a language, for a variety of different reasons. For instance, we might want to prove that a program satisfies some specification with respect to its inputs and output, or that it has some particular runtime. One particular aspect of this is program equivalence. At a basic level, we often think of this as meaning equivalence in behavior — in the functional setting, a natural concept is *extensional equivalence*, wherein we treat two programs as equivalent if they both yield the same value (or, in the presence of recursion, if they both fail to terminate). Intuitively, equivalent programs can be substituted for each other into some larger program without affecting the overall result.

Given a notion of equivalence for one of our languages, we can prove particular programs equivalent to each other, such as showing that a simple (and easy to understand or prove correct) version of some program is equivalent to a more complex (but perhaps more efficient) version. We can also apply this more generally to various forms of optimization transformations, rather than looking at specific programs, and so may be able to prove an optimization pass of a compiler sound, for instance. For a concretely practical, if not extremely complex, example, Jang et al. [47] develop a language based on a natural deduction system for adjoint logic. In the current implementation of this language, a compilation step is performed to translate from the high-level functional syntax, matching natural deduction, to (a variant of) the language presented in Section 4.4. This compilation step introduces many indirections in the form of cuts where one premise is an identity, and, from a computational perspective, these indirections cause overhead, as each one requires a new allocation and a move or copy operation. As such, the implementation includes an optimization pass that eliminates such cuts.[1] With our system for equivalence, we can show that this optimization pass is sound — an important result, if an unsurprising one. This involves showing that for any process term P, the configuration $\mathsf{thread}(a, P), \mathsf{cell}(a, \_)$ is equivalent to $\mathsf{thread}(a, x \leftarrow P[x/a] \ ; \ a \leftarrow x), \mathsf{cell}(a, \_)$, and a similar result for cuts in the other direction.

In our treatment of equivalence in this chapter, we will restrict our attention to the shared-

[1] Information from personal communication with Frank Pfenning, 2024.

memory language presented in Section 4.4, and, in particular, to the *recursion-free* fragment of that language. We develop a core theory of equivalence, focusing on how the more distinct features of this language, particularly the modes and sharing of cells, affect equivalence. The literature contains several approaches to handling equivalence for recursive programs (and recursive types), beginning with approaches based on domain theory [80], which have then been built upon to give more syntactic approaches [9], to step-indexing and its variants [1, 28, 29]. In a similar vein, the work of Somayyajula on termination and partial correctness for recursive programs via logical relations [99, 100], while addressing a different goal than equivalence, uses techniques that may be applicable here as well. More directly related is the work of Balzer et al. [5], which deals with equivalence for recursive session-typed processes in a calculus similar to the purely linear fragment of our message-passing calculus (Section 4.3). We expect an extension of equivalence to apply to recursive processes and types to be possible using some of these techniques, but that this is orthogonal to the new issues of modes, and so treating it, because of the technical detail involved, would obscure what features of equivalence come from the adjoint nature of the underlying language and type system.

In what follows, we will focus in particular on three examples of equivalence: intensional or syntactic equivalence, in which two programs are treated as the same only when they are syntactically equal (up to renaming of bound variables), extensional equivalence, in which two programs are equivalent if they have the same communication behaviour when viewed as black boxes, and proof-irrelevant equivalence, in which any two programs with the same type are equivalent. The first and last of these are extreme examples of equivalence, where as little or as much as possible is equated, while extensional equivalence is a natural concept of equivalence, focusing primarily on the result of computation, rather than the process used to get there. Accordingly, both proof-irrelevant equivalence and intensional equivalence are relatively easy to define, while extensional equivalence is more difficult. However, in keeping with the theme of the rest of this work, we would like to have a *uniform* way of handling different notions of equivalence, and so will endeavor to present the simpler equivalences using the same tools as extensional equivalence.

This uniformity will also allow us to explore *mode-dependent equivalence*. Just as we may assign different structural properties to different modes, or allow different connectives or sets of rules, it may be useful to equip modes with separate notions of equivalence. We will then explore what conditions must be satisfied by these equivalences, analogous to independence for modes, in order to combine them into a coherent notion of equivalence on the whole mode structure.

5.1.1 Background

In the functional setting,[2] equivalence is generally first defined by observation. Intuitively, two programs (expressions, in this setting) should be equal when they are indistinguishable, and if we make precise what it means to observe a difference between two expressions, we can say that two programs are equal exactly when no observation distinguishes them. Of course, for this to work, we need to be precise about what observations are possible, or what features allow us to distinguish two programs from each other. As previously noted, if we allow observations of the syntax of programs, then we can distinguish between many programs that seemingly perform the

[2] See, e.g. Chapter 46 of [40]

same task, for instance the expressions $1 + 1$ and 2. Likewise, if we can observe nothing but the type, we recover a proof-irrelevant equality — any two expressions of the same type are equal.

The usual extensional equivalence is a middle ground, in which we take some basic type b for which a distinction is easily possible (for instance, booleans, where we can distinguish true from false, or natural numbers, where we have a clear notion of when natural numbers m and n are equal), and use this as the basis for observations. An observation (for a program/expression of type τ) then consists of a way to turn an expression of type τ into an expression with type b, generally some form of an expression "with a hole", where the hole in the expression can be filled by another expression of type τ, yielding a well-formed expression overall. This can be used to test if two expressions are distinct by filling the hole with each of the expressions in turn, and evaluating the resulting expression. If they yield different elements of the type b, then they are distinct.[3] We then define equivalence as the absence of any distinguishing observations, yielding the notion of *observational equivalence*.[4]

A key problem with observational equivalence is this need to quantify over all observations, which makes it difficult to reason about. It is therefore useful to have a different description of equivalence that is more feasible to work with, and which, ideally, describes the same relation, so that we can reason easily about something which still captures the intuitive idea of observational equivalence. This is the goal of developing notions of *logical equivalence*, following the method of *logical relations*, first named as such by Statman [101], but based on earlier work by Tait [102], Girard [37], Plotkin [81], and Reynolds [89]. The intuition leading to this approach is that properties of a language, including equivalence, but also, more simply, normalization, or the Church-Rosser property in the λ-calculus, can be difficult to prove by a basic induction over programs. A standard example of the problem is that the λ-calculus term $(\lambda x.M)\ N$ reduces to $[N/x]M$, which may not be smaller than either $\lambda x.M$ or N, and so a syntax-driven proof of normalization gets stuck here, unable to apply the inductive hypothesis. The method of logical relations involves defining an auxiliary object, the logical relation, in a type-directed way, allowing for proofs about logical relations to proceed by induction on the type structure. For proofs of normalization, the relevant logical relation is a type-indexed family of predicates P_τ, defined by induction on the structure of the type τ in such a way that $P_\tau(M)$ implies (generally by definition) that M is (strongly) normalizing. The key result, then, is to show that all well-typed terms M satisfy some P_τ, and so all such terms are (strongly) normalizing.

Logical relations for equivalence are slightly more complex, as equivalence is a binary, rather than unary property, but follow the same general approach — we define a binary logical relation $R_\tau(\cdot, \cdot)$ by induction over its indexing type, in such a way that R_τ reasonably captures our intuition for two programs to be equivalent at type τ. For some types, this can be almost immediate (e.g. for booleans, true and false are each equivalent only to themselves), while more complex types may need to rely on the definition of equivalence for their subtypes (e.g. equivalence for $\tau_1 \to \tau_2$ may refer to equivalence for τ_1 or for τ_2). A first key result is then to show that these

[3] Leaving aside for now the issue of recursive types and non-termination

[4] Note, however, that this notion of observational equivalence depends on what is observable, as well as what computation steps are possible, so while we may refer to a single "observational equivalence" for a given language (which then fixes the possible computation steps), there may be multiple ones depending on what is chosen to be observable. When used without further qualification, by observational equivalence, we mean an extensional form of observational equivalence, with some limited set of directly observable types.

relations R_τ do indeed form (as a family) an equivalence. Transitivity and symmetry are often built into the definition of the relation, and so this main result, often called the fundamental theorem of logical relations/logical equivalence, is that all well-typed programs are R_τ-related to themselves for some appropriate τ, analogous to the unary case.

After establishing that a logical relation defines an equivalence, which we call a logical equivalence, the question still remains of whether it defines a *useful* equivalence. For this, we aim to show that logical equivalence agrees with another, more intuitively defined notion of equivalence, such as a form of observational equivalence. At that point, we are able to combine the ease of reasoning of logical equivalence with the more natural intuition of observational equivalence.

5.1.2 Outline

We will follow a similar approach to the usual functional setting, first exploring observations and observational equivalence in the context of our language. There are several possible ways to define observations, and several ways to extend this to a notion of observational equivalence, depending on what points during computation we allow observation at. We will examine these, before focusing on a choice of observation and of equivalence that are most analogous to the usual ones. We then develop a theory of logical equivalence corresponding to this observational equivalence, noting that both can be parameterized by our choice of computation rules and our choices of what observations are possible. This logical equivalence will agree with observational equivalence (provided we choose the same parameters in both cases). We also explore what properties an equivalence needs to satisfy to be treated as a logical equivalence (and reasoned about correspondingly). Finally, with a framework for reasoning about observations, observational equivalence, and logical equivalence, we examine non-uniform equivalence, where different modes may have different local notions of equivalence. There are several constraints on how these equivalences can interact if we want to extend them conservatively to a single equivalence on the whole language, analogous to the restrictions imposed by independence on the preorder of modes, and we will see that once we find the basic necessary constraints, these are also sufficiently strong to guarantee a coherent overall equivalence.

5.2 Quantified Types

When we examine program equivalence, it will be useful to be able to work with quantified types — in particular, existential types, as these will allow us to provide examples where two different implementations of some specification are equivalent, despite varying enough to have different types. We briefly present here some rules for quantified types and the associated process terms, so that we may make use of them later. Note that we take quantifiers, like the other non-shift type constructors, to operate entirely at one mode. While nothing in principle prevents a cross-mode quantifier, it is easier to work within a single mode. Our extended grammar of types, now including type variables and quantified types, is as follows:

$$\begin{array}{rcl} A_m, B_m & ::= & A_m \multimap B_m \mid A_m \otimes B_m \mid \mathbf{1}_m \mid \oplus_{j \in J} A_m^j \mid \&_{j \in J} A_m^j \mid \uparrow_k^m A_k \mid \downarrow_m^\ell A_\ell \\ & & \mid t_m \mid \exists t_m . A_m \mid \forall t_m . A_m \end{array}$$

$$\frac{\Xi \vdash C_r \quad \Xi, t_m \,;\, \Gamma, (x : \exists t_m.A_m)^\alpha, y : A_m \vdash Q :: (z : C_r)}{\Xi \,;\, \Gamma, x : \exists t_m.A_m \vdash \textbf{case}\, x\, (\langle t_m, y\rangle \Rightarrow Q) :: (z : C_r)}\ \exists L_\alpha$$

$$\frac{\Xi \vdash B_m \quad \Xi, t_m \vdash A_m}{\Xi \,;\, \Gamma_W, (x : A_m[B_m/t_m]) \vdash z.\langle B_m, x\rangle :: (z : \exists t_m.A_m)}\ \exists R^0$$

$$\frac{\Xi \vdash B_m}{\Xi \,;\, \Gamma_W, x : \forall t_m.A_m \vdash x.\langle B_m, z\rangle :: (z : A_m[B_m/t_m])}\ \forall L^0$$

$$\frac{\Xi, t_m \,;\, \Gamma \vdash Q :: (y : A_m)}{\Xi \,;\, \Gamma \vdash :: \mathsf{case}\, z\, (\langle t_m, y\rangle \Rightarrow Q) :: (z : \forall t_m.A_m)}\ \forall R$$

Figure 5.1: Additional rules for quantified types in a semi-axiomatic presentation of ADJ^I. As in the earlier rules for ADJ^I, α ranges over the set $\{0, 1\}$, $(A_m)^0$ denotes the empty context, and $(A_m)^1$ denotes the singleton context A_m. If $\alpha = 1$, then $\mathsf{C} \in \sigma(m)$ should be treated as an additional premise of the rule.

We will use a context Ξ to track type variables. In most rules, Ξ is passed upwards unchanged, with the rules for quantifiers being the only ones that actually modify or make use of Ξ, and so we will tend to leave it implicit in other rules when possible. The rules for quantifiers (along with their associated process terms) can be found in Figure 5.1.

These rules also include two new process terms, $z.\langle B_m, x\rangle$ and $\mathsf{case}\, z\, (\langle t_m, x\rangle \Rightarrow Q)$, whose syntax is analogous to the process terms for pairs and functions. The term $z.\langle B_m, x\rangle$ communicates the pair of a type B_m and a symbol x along the symbol z. In the shared-memory setting that we are working in, this means writing this pair into memory at address z for existentials, and reading a corresponding continuation out of memory (also at address z) for universals. Dually, $\mathsf{case}\, z\, (\langle t_m, y\rangle \Rightarrow Q)$ either reads such a pair out of the cell at address z (for existentials) or writes a continuation waiting for such a pair into the cell at address z.

The existing reduction rules for shared memory (Figure 4.4) already cover how these new process terms can be used to write to memory (provided we add $\langle B_m, c\rangle$ to our grammar of values, and $(\langle t_m, x\rangle \Rightarrow Q)$ to our grammar of continuations). To address reading these new data, we need to define a new case of the operation $V \triangleright K$, desbcribing how they interact. We define $\langle B_m, c\rangle \triangleright (\langle t_m, x\rangle \Rightarrow Q) ::= Q[B_m/t_m, c/x]$, again, following the pattern of pairs and functions.

5.3 Renaming

One technical detail that we need to address before moving on to a full treatment of equivalence is that of renaming. Since the objects under consideration here are configurations of memory cells, where each cell has an address a, we need to consider, for instance, if the configurations $\mathsf{cell}(a, V)$ and $\mathsf{cell}(b, V)$ should be considered the same. There are several different notions of renaming that we could choose to work with here. If we take a renaming to be a bijection (assigning to each symbol a new, replacement symbol), we have a notion of renaming that preserves information such as how many copies of a given piece of data a configuration contains. For instance, under this notion of renaming, we might identify the configurations $\mathsf{cell}(a, V)$ and $\mathsf{cell}(b, V)$ (under the renaming that maps a to b), but not $\mathsf{cell}(a, V)$ and $\mathsf{cell}(b, V), \mathsf{cell}(c, V)$, as the former has only one distinctly-addressed cell containing the value V, while the latter has two.

However, in a setting where memory cells may be persistent, able to be read from by more than one reader, a reasonable alternative is to allow any number of persistent cells to relate to any other number of persistent cells, as long as all contain the same (or at least adequately related, in the sense of being the same up to an appropriate notion of renaming, which we will address in this section) data. This notion of renaming does identify configurations that differ in memory usage, something which may be useful to distinguish under some circumstances, but matches nicely with the intuition that we should not be able to tell at what location in memory some piece of data is stored — just whether it is stored at all and how it connects to other memory cells.

Not all relations ρ on symbols are suitable to use as renamings, however. Although it makes sense to allow persistent cells (i.e. cells whose modes admit contraction) to be merged together (or split apart) by renaming, with n cells containing the same data being renamed to have a different number k of names, this should not be possible for ephemeral cells, or for running processes which provide a symbol at a mode which does not admit contraction. We formalize this with the following definition:

Definition 4. *A relation ρ on symbols (with associated mode information, e.g. a_m or b_k) is* well-moded *if:*

- *ρ only relates symbols with the same mode. That is, if $\rho(a_m, b_k)$, then $m = k$.*
- *For each symbol ${a_m}$[5], either $\mathsf{C} \in \sigma(m)$ or the sets $\{b_m \mid \rho(a_m, b_m)\}$ and $\{b_m \mid \rho(b_m, a_m)\}$ are both subsingletons. Informally, this means that ρ relates a_m to at most one b_m in each direction, unless m admits contraction.*

While it does not appear strictly necessary for our use case, it may also be convenient to require that the relation ρ behaves well with respect to its inverse relation. In particular, it is natural for a renaming to satisfy the property that if $\rho(a_m, b_m)$ and $\rho(c_m, b_m)$, so a_m and c_m may be identified by renaming, then the sets $\{b_m \mid \rho(a_m, b_m)\}$ and $\{b_m \mid \rho(c_m, b_m)\}$ should be the same — that is, if a_m and c_m share one possible new name, they must share all possible new names.

There are several terms for relations satisfying this property — originally, they are called *difunctional relations* [91], but they also appear in the computer science ltierature as *zig-zag complete relations* or *quasi-PERs*[6] (see, for instance, Krishnaswami and Dreyer [52]). We will take the original definition of this property:

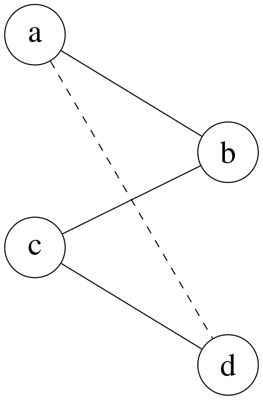
Definition 5 (Riguet [91])**.** *A relation ρ from X to Y is difunctional if $\rho\rho^{-1}\rho \subseteq \rho$, or, equivalently, if $\rho\rho^{-1}\rho = \rho$. That is, if $\rho(a, b)$, $\rho(c, b)$, and $\rho(c, d)$, then also $\rho(a, d)$ (and, conversely, if $\rho(a, d)$, then there exist b, c such that $\rho(a, b)$ and $\rho(c, b)$ and $\rho(c, d)$, but this direction is less interesting — just take $b = d$ and $c = a$).*

This definition, while concise, is not particularly illustrative, especially of the terms "zig-zag

[5] Technically for size reasons, we should restrict to some fixed set of symbols, but as in much of computer science, it is convenient to assume that all symbols are drawn from some fixed countably infinite set, and to ignore such details.

[6] quasi- partial equivalence relations

complete relation" and "quasi-PER". We illustrate the former with the following diagram:



Difunctionality says that for every instance of the zig-zag figure shown in solid lines (where a line indicates a ρ-relation from left to right), the dashed line must exist, hence zig-zag completeness. Some more early results on difunctionality illustrate that a difunctional relation can be interpreted as a heterogeneous equivalent of a (partial) equivalence relation, justifying also the term quasi-PER. Firstly, if ρ is a difunctional relation from a set X to itself, then ρ is an equivalence relation on the set $\{x \mid \rho(x,x)\}$ (or a partial equivalence relation on X).

Theorem 25 (Riguet [91], Proposition 12)**.** *If ρ is a difunctional relation on a set X, and it is* quasi-reflexive *in that whenever $\rho(a,b)$, it is also true that $\rho(a,a)$ and $\rho(b,b)$, then ρ is a partial equivalence relation on X (equivalently, an equivalence relation on its domain $\{x \mid \exists y.\rho(x,y)\}$)*

Secondly, every difunctional relation composes with its inverse to form an equivalence relation.

Theorem 26 (Riguet [91])**.** *If ρ is a difunctional relation from X to Y, and for every $x \in X$ there is a $y \in Y$ such that $\rho(x,y)$, then $\rho\rho^{-1}$ is an equivalence relation on X.*

A corollary of this is that $\rho\rho^{-1}$ is an equivalence relation on the domain of ρ (and dually, $\rho^{-1}\rho$ is an equivalence relation on the codomain of ρ).

As such, just as a (partial) equivalence relation partitions its domain into equivalence classes, a difunctional relation with domain X and codomain Y partitions both X and Y into equivalence classes, and, moreover, is a bijection between these sets of equivalence classes,thereby, in a sense, partitioning the disjoint union $X \sqcup Y$ into equivalence classes.

Lemma 7. *Suppose ρ is a difunctional relation from X to Y, and that $\rho\rho^{-1}(x,x')$ and $\rho^{-1}\rho(y,y')$. Then, the following are equivalent:*

1. $\rho(x,y)$
2. $\rho(x,y')$
3. $\rho(x',y)$
4. $\rho(x',y')$.

That is, selecting an equivalence class of X under $\rho\rho^{-1}$ uniquely determines an equivalence class of Y under $\rho^{-1}\rho$, and vice versa.

Proof. We will show the first implication from 1. to 2. — that 2. implies 3., 3. implies 4., and 4. implies 1. are all similar.

Suppose $\rho(x, y)$. Since $\rho^{-1}\rho(y, y')$, we know that there exists some $z \in X$ such that $\rho(z, y)$ and $\rho(z, y')$. Difunctionality then immediately gives us that $\rho(x, y')$ (via the zig-zag x, y, z, y'). □

Now, in order to develop our theory of renamings, we begin with some basic results about difunctional relations:

Lemma 8 (Basic Properties of Difunctional Relations)**.** *Suppose ρ is a difunctional relation from X to Y. Then:*

1. *The diagonal relation on X $\Delta_X = \{(x, x) \mid x \in X\}$ is difunctional.*
2. *ρ^{-1} is difunctional.*

Proof. That Δ_X is difunctional is immediate.

Suppose that $\rho^{-1}(a, b)$, $\rho^{-1}(c, b)$ and $\rho^{-1}(c, d)$. By definition, then, $\rho(b, a)$, $\rho(b, c)$, and $\rho(d, c)$, and so, as ρ is difunctional, we must have that $\rho(d, a)$ (and so also $\rho^{-1}(a, d)$), so ρ^{-1} is difunctional. □

This will allow us to easily show (once we have defined which such relations are renamings) that equality up to renaming is reflexive and symmetric — transitivity is more complex, as the composition of two difunctional relations need not be difunctional.

We will say (in this section and the remainder of this chapter, when handling renaming) that a relation that is both difunctional and well-moded is a *potential renaming*.

In order to reason about composition of (potential) renamings, we need to address the problem that the composition of difunctional relations is not, in general, difunctional. As a first step for defining the composition of renamings, then, we need to build a difunctional relation from the input renamings. In doing so, we will make use of the notion of the *difunctional closure* ρ^d of a relation ρ, which is the smallest difunctional relation containing ρ. The difunctional closure has a simpler definition in terms of ρ, due to Riguet:

Theorem 27 (Riguet 1950 [92])**.** *Suppose ρ is a relation from X to Y. Then, $\rho^d = \rho(\rho^{-1}\rho)^+ = (\rho\rho^{-1})^+\rho$, where $(\cdot)^+$ denotes the transitive closure.*

Equipped with this, we can compose (in a sense) potential renamings, allowing us to show that the relation $X \sim Y$ if there is a potential renaming from X to Y is an equivalence relation.

Theorem 28.

- *The diagonal relation Δ_X is a potential renaming from X to X.*
- *ρ^{-1} is a potential renaming from Y to X whenever ρ is a potential renaming from X to Y*
- *Suppose ρ is a potential renaming from X to Y, and σ is a potential renaming from Y to Z. Then, $(\rho\sigma)^d$ is a potential renaming from X to Z.*

Proof. Given any set X, we know from Lemma 8 that the diagonal relation Δ_X is difunctional. It is also easy to see that Δ_X is well-moded, as every symbol has the same mode as itself, and it relates each symbol to exactly one target in each direction. Therefore, Δ_X is a potential renaming from X to X.

We also know from Lemma 8 that if ρ is a potential renaming, then ρ^{-1} is difunctional, and it is again easy to see that ρ is well-moded if and only if ρ^{-1} is, as the definition is symmetric.

Finally, we need to establish that $(\rho\sigma)^d$ is a potential renaming from X to Z. Certainly, it is a difunctional relation from X to Z, being the difunctional closure of a relation from X to Z. It remains to check that $(\rho\sigma)^d$ is well-moded. To do this, we will show that the composition of well-moded relations is well-moded, and that the transitive closure of a quasi-reflexive[7] well-moded relation is well-moded. Using Riguet's characterization of the difunctional closure (and the fact that for any relation ρ, the relation $\rho^{-1}\rho$ is quasi-reflexive), this will suffice to show that $(\rho\sigma)^d$ is well-moded, provided that ρ and σ both are, which follows from both being potential renamings.

First, consider the composition of well-moded relations. Suppose that relations R and S are well-moded and that $RS(a_m, c_k)$. Then, there must be some b_ℓ such that $R(a_m, b_\ell)$ and $S(b_\ell, c_k)$, and since R and S are both well-moded, $m = \ell = k$. Further, if we consider some a_m with $C \notin \sigma(m)$, and examine the set $\{c_m \mid RS(a_m, c_m)\}$, we see that this can be rewritten as $\{c_m \mid \exists b_m . R(a_m, b_m) \wedge S(b_m, c_m)\}$. Since R is well-moded, there can be at most one such b_m, and since S is well-moded, there can be at most one such c_m for each b_m, and so this set is a subsingleton. The sets $\{a_m \mid RS(a_m, c_m)\}$ for fixed c_m can be shown similarly to be subsingletons.

Now, suppose R is quasi-reflexive and well-moded. If $R^+(a_m, c_k)$, there is some sequence $b^0_{m_0}, \ldots, b^n_{m_n}$ such that $a_m = b^0_{m_0}$, $c_k = b^n_{m_n}$, and $R(b^i_{m_i}, b^{i+1}_{m_{i+1}})$ for each i from 0 to $n-1$. Since R is well-moded, this means that $m = m_0 = \ldots = m_n = k$. Now, consider a_m with $C \notin \sigma(m)$, and examine $\{c_m \mid R^+(a_m, c_m)\}$. Again, for each c_m in this set, there must be a sequence $b^0_m, \ldots, b^n_m$ with $a_m = b^0_m$, $c_m = b^n_m$, and $R(b^i_m, b^{i+1}_m)$ for each i from 0 to $n-1$. Since R is well-moded, there is at most one choice for each b^{i+1}_m, determined by the value of b^i_m. However, we still need to ensure that the end result b^n_m is the same for all choices of n. Since R is quasi-reflexive, in particular, if $R(a_m, b^1_m)$, it must also be the case that $R(a_m, a_m)$. This then means that $b^1_m = a_m$, as otherwise we would have $\{b_m \mid R(a_m, b_m)\}$ containing two elements, and not being a subsingleton. We can then prove (via induction) that regardless of the choice of n, all b^i_m must be equal to a_m, and therefore so is c_m, meaning that the original set is indeed a subsingleton. As for composition, the other set we need to consider is a subsingleton via a symmetric argument.

We therefore have that $(\rho\sigma)^d$ is a potential renaming from X to Z. □

To go from potential to actual renaming, we need to also consider the configurations we are working with. In particular, if ρ is to be a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$, then we need to ensure that whenever $\rho(a_m, b_m)$, the object providing a_m in $\mathcal{C}_1$ corresponds appropriately to the object providing b_m in $\mathcal{C}_2$. We also generally want to ensure that ρ is *total* on the set of symbols used in $\mathcal{C}_1$ (and, from the other side, the symbols used in $\mathcal{C}_2$). This second condition is easy to express — a relation ρ is *total* with respect to $\mathcal{C}_1$ and $\mathcal{C}_2$ if whenever a symbol a_m appears in $\mathcal{C}_1$, there is some b_m in $\mathcal{C}_2$ such that $\rho(a_m, b_m)$, and dually, if b_m appears in $\mathcal{C}_2$, there is some a_m in $\mathcal{C}_1$ such that $\rho(a_m, b_m)$. Formally, this requires defining what it means for a symbol to appear in a configuration, but this is sufficiently straightforward that we omit the formalism.

The first condition is more complex, requiring that we address what "appropriately corresponding" semantic objects are with respect to a (potential) renaming. We begin by defining how to extend a potential renaming on symbols to also apply to values, continuations, and process

[7] A relation ρ is *quasi-reflexive* if whenever $\rho(a, b)$ holds, $\rho(a, a)$ and $\rho(b, b)$ hold as well

terms, as each of these types of data appear in our semantic objects. renaming on symbols to also apply to values, continuations, and process terms.

Definition 6. *Suppose ρ is a potential renaming. We can extend ρ to values, continuations, and process terms in the usual manner for congruences, formally defined as follows. First, we consider values:*

- $\rho(\langle\rangle, \langle\rangle)$
- $\rho(\ell(a_m), \ell(b_m))$ *if* $\rho(a_m, b_m)$.
- $\rho(\langle a_m, b_m\rangle, \langle c_m, d_m\rangle)$ *if* $\rho(a_m, c_m)$ *and* $\rho(b_m, d_m)$.
- $\rho(\mathsf{shift}(a_m), \mathsf{shift}(b_m))$ *if* $\rho(a_m, b_m)$.

Then continuations:

- $\rho(\langle\rangle \Rightarrow P, \langle\rangle \Rightarrow Q)$ *if* $\rho(P, Q)$.
- $\rho(j(x_j) \Rightarrow P, j(x_j) \Rightarrow Q)$ *if* $\rho(P, Q)$. *Note that since x_j is a* bound variable*, its name can freely vary, and it is not subject to renaming, as symbols are. Recall also that while variables may be either free or bound, symbols are* always free *in a term.*
- $\rho(\langle x, y\rangle \Rightarrow P, \langle x, y\rangle \Rightarrow Q)$ *if* $\rho(P, Q)$.
- $\rho(\mathsf{shift}(x) \Rightarrow P, \mathsf{shift}(x) \Rightarrow Q)$ *if* $\rho(P, Q)$.

Then process terms:

- $\rho(a \leftarrow b, a' \leftarrow b')$ *if* $\rho(a, a')$ *and* $\rho(b, b')$.
- $\rho(x \leftarrow P \; ; Q, x \leftarrow P' \; ; Q')$ *if* $\rho(P, P')$ *and* $\rho(Q, Q')$. *As in some of the continuations, the x here is a bound variable, which can therefore freely vary, and is not subject to renaming.*
- $\rho(c.V, c'.V')$ *if* $\rho(c, c')$ *and* $\rho(V, V')$.
- $\rho(\mathsf{case}\ c\ K, \mathsf{case}\ c'\ K')$ *if* $\rho(c, c')$ *and* $\rho(K, K')$.
- $\rho(a \leftarrow p\ \overline{b}, a' \leftarrow p\ \overline{b'})$ *if* $\rho(a, a')$ *and* $\rho(\overline{b}, \overline{b'})$*, where the relation of vectors of symbols is interpreted pointwise.*

Now, with this extension of ρ to handle the contents of configurations, we can formalize our first condition.

Definition 7. *Suppose ρ is a potential renaming. We say that ρ is* content-preserving *for a pair of configurations $\mathcal{C}_1, \mathcal{C}_2$ if whenever $\rho(a_m, b_m)$,*

- $\mathcal{C}_1$ *contains* $!_m\mathsf{cell}(a_m, D_1)$ *if and only if* $\mathcal{C}_2$ *contains* $!_m\mathsf{cell}(b_m, D_2)$*, with* $\rho(D_1, D_2)$.
- $\mathcal{C}_1$ *contains* $\mathsf{thread}(a_m, P_1)$ *if and only if* $\mathcal{C}_2$ *contains* $\mathsf{thread}(b_m, P_2)$ *with* $\rho(P_1, P_2)$.

We can then finally give our definition of renamings between configurations.

Definition 8. *Suppose ρ is a potential renaming, and $\mathcal{C}_1, \mathcal{C}_2$ are configurations. Then, ρ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$ if:*

- *ρ is total with respect to $\mathcal{C}_1$ and $\mathcal{C}_2$.*
- *ρ is content-preserving for $\mathcal{C}_1$ and $\mathcal{C}_2$.*

As we have seen, existence of *potential* renamings forms an equivalence relation on sets of symbols, defined by $X \sim Y$ if there is a potential renaming from X to Y. We would like to extend this to an equivalence relation on configurations, capturing the notion of equality up to renaming, given by $\mathcal{C}_1 \sim \mathcal{C}_2$ if there is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$. In order to establish this, we need to extend our previous result on potential renamings to actual renamings.

Theorem 29 (Equality up to renaming is an equivalence)**.** *Suppose $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ are configurations. Then,*

- *The diagonal relation on the set of symbols in $\mathcal{C}_1$ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_1$ (Renaming is reflexive)*
- *The relation ρ^{-1} is a renaming from $\mathcal{C}_2$ to $\mathcal{C}_1$ whenever ρ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$ (Renaming is symmetric)*
- *If ρ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$, and σ is a renaming from $\mathcal{C}_2$ to $\mathcal{C}_3$, then the relation $(\rho\sigma)^d$ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_3$ (Renaming is transitive)*

Proof. We already know (from Theorem 28) that each of the three mentioned relations are potential renamings, and so we focus on showing that each is total and content-preserving. We consider the three parts in turn.

That the diagonal relation on the symbols of $\mathcal{C}_1$ is total (with respect to $\mathcal{C}_1$) is immediate, with every symbol occurring in $\mathcal{C}_1$ being related to itself. That it is content-preserving is equally immediate — of course, each object $!_m\mathsf{cell}(a_m, D)$ exists in $\mathcal{C}_1$ if and only if that same cell itself exists in $\mathcal{C}_1$ (and likewise for processes), and the diagonal relation is easily seen to relate cell contents D with themselves.

Since ρ is total with respect to $\mathcal{C}_1$ and $\mathcal{C}_2$, ρ^{-1} is also total with respect to $\mathcal{C}_2$ and $\mathcal{C}_1$, as the definition of totality is symmetric. Similarly, the definition of content-preserving is also symmetric, and so since ρ is content-preserving, ρ^{-1} is as well.

As for potential renamings, the bulk of the proof lies in transitivity, due to the extra complication of the difunctional closure. We will follow the same approach, showing that totality and content-preservation are properties preserved by composition and by transitive closure (here, we do not need that $\rho^{-1}\rho$ is always quasi-reflexive, as we did for well-modedness).

We first consider composition of relations. Suppose that R and S are renamings from $\mathcal{C}_1$ to $\mathcal{C}_2$ and $\mathcal{C}_2$ to $\mathcal{C}_3$, respectively. For any symbol a_m in $\mathcal{C}_1$, since R is a renaming, there is some b_m in $\mathcal{C}_2$ such that $R(a_m, b_m)$. Then, we also have that there is some $c_m \in \mathcal{C}_3$ such that $S(b_m, c_m)$, and so $RS(a_m, c_m)$, and so RS is total.

Suppose that $RS(a_m, c_m)$, so there is some b_m with $R(a_m, b_m)$ and $S(b_m, c_m)$. Then, a cell $!_m\mathsf{cell}(a_m, D_1)$ exists in $\mathcal{C}_1$ if and only if there is $!_m\mathsf{cell}(b_m, D_2)$ in $\mathcal{C}_2$ with $R(D_1, D_2)$, which is true if and only if there is $!_m\mathsf{cell}(c_m, D_3)$ in $\mathcal{C}_3$ with $S(D_2, D_3)$. It remains only to check that $RS(D_1, D_3)$, but this is immediate from the definition of the extension of potential renamings to cell contents. The case for process terms is similar.

Now, we consider transitive closure. Suppose that R is a renaming from $\mathcal{C}_1$ to itself. Since R is total, and $R \subseteq R^+$, we can easily see that R^+ is also total (with respect to $\mathcal{C}_1$). If $R^+(a_m, c_m)$, then there is some sequence $b_m^0, \dots, b_m^n$[8] with $a_m = b_m^0$, $c_m = b_m^n$, and $R(b_m^i, b_m^{i+1})$ for each i from 0 to $n-1$. Since R is content-preserving, we get that $!_m\mathsf{cell}(b_m^i, D_i)$ exists in $\mathcal{C}_1$ if and only if $!_m\mathsf{cell}(b_m^{i+1}, D_{i+1})$ with $R(D_i, D_{i+1})$. Following the chain of iffs (or, more formally, going through an inductive argument), we see that $!_m\mathsf{cell}(a_m, D_1)$ exists in $\mathcal{C}_1$ if and only if $!_m\mathsf{cell}(c_m, D_2)$ exists in $\mathcal{C}_1$, with $R^+(D_1, D_2)$.

We therefore have that $(\rho\sigma)^d$ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_3$, as desired. □

We will, in general, treat configurations as equal if they differ only up to a renaming valid under these conditions. Of course, when we work with defining other equivalences, we will need

[8] Technically, with $n \geq 1$, but this isn't necessary for the argument, and R is quasi-reflexive in the case we work with in any case

to show that these equivalences respect renaming, in order for them to be well-formed.

One notable observation about these renamings is that equality up to renaming is *not* a congruence. That is, it is possible to give configurations $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_1', \mathcal{C}_2'$ such that $\mathcal{C}_1$ and $\mathcal{C}_1'$ are equal up to renaming, as are $\mathcal{C}_2, \mathcal{C}_2'$, but $\mathcal{C}_1, \mathcal{C}_2$ is not equal to $\mathcal{C}_1', \mathcal{C}_2'$ up to renaming. For a concrete example, we can take:

$$\begin{aligned}
\mathcal{C}_1 &= \mathsf{cell}(a, \langle\rangle) & \mathcal{C}_1' &= \mathsf{cell}(a_1, \langle\rangle)\\
\mathcal{C}_2 &= \mathsf{thread}(c, \mathsf{case}\ a\ \langle\rangle \Rightarrow c.\langle\rangle) & \mathcal{C}_2' &= \mathsf{thread}(c, \mathsf{case}\ a_2\ \langle\rangle \Rightarrow c.\langle\rangle)
\end{aligned}$$

Composing the first two configurations, we have a cell at address a, and a thread that reads from that cell. With the latter two, however, since a has been renamed to a_1 in $\mathcal{C}_1'$ and a_2 in $\mathcal{C}_2'$, these cells are no longer connected to each other, and $\mathcal{C}_1, \mathcal{C}_2$ behaves differently from $\mathcal{C}_1', \mathcal{C}_2'$. This is somewhat to be expected, however — while some symbols (addresses, in the current shared-memory context) are private to a configuration and can be freely renamed, others, including all symbols for which contraction is admissible, appear in the interface of the configuration, and affect how it composes with other configurations. Renaming these symbols that appear in the interface is akin to renaming free variables of a term in a functional language, changing which contexts it can be interpreted in.

5.4 Observations and Observational Equivalence

We begin by looking at how to define observations and observational equivalence in the context of our shared-memory concurrent language. In the functional setting, generally, some collection of values are considered to be observable, and observational equivalence can be defined from that notion — the focus is on what final result an expression reduces to. For process calculi, a different notion is often used, where rather than the final result of a computation, some collection of actions taken by the computation are considered to be observable. In systems based on the π-calculus, it is common for all interactions to be observable actions — that is, when a process sends a message to another, that message becomes observable. This gives rise to the notion of *bisimilarity* (seemingly first named as such by Park [71], building on ideas from Milner [60]. See [40] for an overview of bisimulations in a simple process calculus), under which P and Q are equivalent if whenever P may take an action α to step to some P', Q can take the same action α to step to some other Q' such that P' and Q' are also equivalent. There has been a great deal of further work on bismilarity and bisimulations (see, e.g., barbed bisimulations [64] or branching bisimulations [105]), addressing different notions of observability. In general, bisimulation-based approaches are interested in the process by which a result is reached, as well as what that result is. We will follow more closely to the functional approach, emphasizing the end result of a computation over the process by which it is reached, although there is certainly room for an alternate theory of equivalence for the same language, which takes the more fine-grained approach of work on bisimulation.

Since a single process term, when executed, may result in a much larger configuration, our analog to an expression is a configuration, rather than a process term, even if many of the configurations whose equality we are interested in consist of a single thread object, running a single process term. As in the functional case, we could then select some base type b (perhaps booleans, encoded as $\mathbf{1} \oplus \mathbf{1}$) and attempt to test for equality at this type. However, selecting a particular type for this feels somewhat artificial, and indeed we will see some examples, particularly in a linear setting, where it takes additional work to distinguish two configurations by turning their outputs into distinct booleans, while they are easy to distinguish by examining some output at a different type. Moreover, unlike expressions, which have a single result by nature, configurations provide potentially many semantic objects, each with its own associated symbol which can be used to access it. In order to compare two configurations, we therefore either need to consider the configuration as a whole, or specify which symbol(s) to compare. Comparing whole configurations seems natural, but raises some further questions: For instance, should two configurations where one is a subset of the other be distinguishable? Instead, we will avoid this issue altogether, by comparing two configurations for equivalence *at a particular symbol or set of symbols*. Intuitively, we can think of this as giving some external observer access to a fixed set of memory addresses, and allowing them to explore by following pointers, but not to read out the entire memory space looking for differences.

Our first step in defining observational equivalence in this setting is, therefore, to start with a function $\mathcal{O}(\mathcal{C}, S)$, taking a configuration and a set of symbols to some type of object that can be compared for syntactic equality (for our use cases, these will be a restricted form of configuration). This function can be thought of as defining what can be observed from a particular configuration $\mathcal{C}$ by looking at the addresses in S, without being able to take any computation steps. For a simple example, if we define $\mathcal{O}_{univ}(\mathcal{C}, S) = \{\}$, this suggests that nothing can be observed about a configuration, and if we use this as the basis for an observational equivalence, we will get the universal relation, under which any two configurations are equivalent (or, with restrictions on type to ensure that configurations of different type cannot be equivalent, we get proof-irrelevant equivalence, where two configurations are equivalent at a set S of symbols if they have the same type at those symbols). A more natural notion of observation, $\mathcal{O}_{int}$, allows us to observe data stored in cells reachable by following pointers from S. This essentially says that computations are not observable while in progress, but all finished computations are, and will yield a notion of equivalence corresponding to *intensional* equivalence, or syntactic equality up to variable renaming. Finally, and most practical of the three examples we will examine, we can define $\mathcal{O}_{ext}$ to allow values (but not continuations) stored in cells reachable by following pointers from S to be observed. This corresponds to the standard extensional equivalence of functional programming, where values of some purely positive type(s) can be directly observed, but values of negative types, in particular functions, can only be examined via experiment. In the language of call-by-push-value, for example [54, 55], values can be observed directly, but computations cannot.

With these intuitions in mind for several examples of observation, we now define them formally. It is quite easy to define $\mathcal{O}_{univ}$, which returns the empty configuration on any input. The other two are more involved, as we would like to ensure that our observation is independent of the order of objects in $\mathcal{C}$, only considering $\mathcal{C}$ and S as sets. To address this, we break down the set S into smaller sets, and see what we can observe from each individual address. Of course, we may see the same cell many times this way, but if our output object is a set, the number of times we encounter a cell does not affect the output. We first define $\mathcal{O}_{\mathit{int}}$:

$$\begin{array}{rcl}
\mathcal{O}_{\mathit{int}}(\mathcal{C}, \{\}) & = & \{\} \\
\mathcal{O}_{\mathit{int}}(\mathcal{C}, S_1 \cup S_2) & = & \mathcal{O}_{\mathit{int}}(\mathcal{C}, S_1) \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, S_2) \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, K)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, K)\} \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, \mathsf{FS}(K)) \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle\rangle)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \langle\rangle)\} \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \ell(a_m))), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \ell(a_m))\} \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, \{a_m\}) \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle a_m, b_m\rangle)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \langle a_m, b_m\rangle)\} \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, \{a_m, b_m\}) \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \mathsf{shift}(a_m))), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \mathsf{shift}(d_m))\} \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, \{a_m\}) \\
\mathcal{O}_{\mathit{int}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle A_m, a_m\rangle)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \langle A_m, a_m\rangle)\} \cup \mathcal{O}_{\mathit{int}}(\mathcal{C}, \{a_m\}) \\
\mathcal{O}_{\mathit{int}}(\mathcal{C}, \{c_m\}) & = & \{\} \qquad \text{(if } \mathcal{C} \text{ does not provide a cell at } c_m\text{)}
\end{array}$$

In a continuation, any free symbol is a pointer to a cell that the continuation may read from, and so we record having seen the continuation, and further examine the cells it may read from (with $\mathsf{FS}(K)$ denoting the set of free symbols in K). For values, we record the value as observed, and then any addresses contained in the value become the new basis for further observation. The definition of $\mathcal{O}_{ext}$ is quite similar, differing in that we disallow observing continuations, with an attempt to observe a continuation instead yielding the empty set. Note that we also disallow observing existentials — types may not necessarily be made explicit in a real implementation, and without the type, we do not know how to further observe under the existential either.

$$\begin{array}{rcl}
\mathcal{O}_{\mathit{ext}}(\mathcal{C}, \{\}) & = & \{\} \\
\mathcal{O}_{\mathit{ext}}(\mathcal{C}, S_1 \cup S_2) & = & \mathcal{O}_{\mathit{ext}}(\mathcal{C}, S_1) \cup \mathcal{O}_{\mathit{ext}}(\mathcal{C}, S_2) \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, K)), \{c_m\}) & = & \{\} \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle\rangle)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \langle\rangle)\} \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \ell(a_m))), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \ell(a_m))\} \cup \mathcal{O}_{\mathit{ext}}(\mathcal{C}, \{a_m\}) \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle a_m, b_m\rangle)), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \langle a_m, b_m\rangle)\} \cup \mathcal{O}_{\mathit{ext}}(\mathcal{C}, \{a_m, b_m\}) \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \mathsf{shift}(a_m))), \{c_m\}) & = & \{!_m\mathsf{cell}(c_m, \mathsf{shift}(a_m))\} \cup \mathcal{O}_{\mathit{ext}}(\mathcal{C}, \{a_m\}) \\
\mathcal{O}_{\mathit{ext}}((\mathcal{C}, !_m\mathsf{cell}(c_m, \langle A_m, a_m\rangle)), \{c_m\}) & = & \{\} \\
\mathcal{O}_{\mathit{ext}}(\mathcal{C}, \{c_m\}) & = & \{\} \qquad \text{(if } \mathcal{C} \text{ does not provide } c_m\text{)}
\end{array}$$

5.4.1 Observation and Renaming

Since we consider configurations which agree up to renaming to be equal, for our definitions of observation to be well-formed, they must also agree up to renaming. This will then also ensure that when we define observational equivalence based on these observations, that it respects renaming.

Since the result of observation is again a configuration, we do not need an additional definition for renaming of observations — we can just reuse the definition for configurations.

Lemma 9. *Suppose $\mathcal{C}_1, \mathcal{C}_2$ are configurations, ρ is a renaming from $\mathcal{C}_1$ to $\mathcal{C}_2$, and S_1, S_2 are sets of symbols such that $a_m \in S_1$ iff there is $b_m \in S_2$ with $\rho(a_m, b_m)$.*

Then, $\mathcal{O}_{\mathit{int}}(\mathcal{C}_1, S_1)$ and $\mathcal{O}_{\mathit{int}}(\mathcal{C}_2, S_2)$ are related by ρ, and likewise for $\mathcal{O}_{\mathit{univ}}$ and $\mathcal{O}_{\mathit{ext}}$.

Proof. For $\mathcal{O}_{\mathit{univ}}$, this is immediate, as the observations of $\mathcal{C}_1$ and $\mathcal{C}_2$ are both the empty configuration.

For $\mathcal{O}_{int}$, we proceed by induction on the construction of the observation.

If the last case used was $\mathcal{O}_{int}(\mathcal{C}_i, \{\}) = \{\}$, so the S_i are empty, the result is immediate, just as for $\mathcal{O}_{univ}$.

If the last case used was $\mathcal{O}_{int}(\mathcal{C}_i, T_i \cup U_i)$, so $S_i = T_i \cup U_i$, then, applying the inductive hypothesis, we get that $\mathcal{O}_{int}(\mathcal{C}_i, T_i)$ are related by ρ, as are $\mathcal{O}_{int}(\mathcal{C}_i, U_i)$. It remains to show that taking the union of these sets preserves their ρ-relatedness. Since the ρ used is the same, we still have that ρ only relates a non-persistent symbol to one other symbol, regardless of what configuration it is applied to. Now, suppose that $\rho(a_m^1, a_m^2)$. Since a_m^1 appears only in $\mathcal{O}_{int}(\mathcal{C}_1, T_1)$ (or only in $\mathcal{O}_{int}(\mathcal{C}_1, U_1)$), it is immediate that the contents of the cells at a_m^1 and a_m^2 are consistent, from the inductive hypothesis.

In the remaining cases, we likewise reason about unions of configurations, in the same manner (using that the sets of symbols we work with are disjoint), and use the inductive hypothesis where applicable. Since the original configurations $\mathcal{C}_1, \mathcal{C}_2$ are ρ-related, any observation at some a_m in $\mathcal{C}_1$ corresponds to an observation at b_m in $\mathcal{C}_2$ for which $\rho(a_m, b_m)$, which then therefore yields a similarly related cell.

We do note that the number of times a given related cell appears in the resulting observation may vary (e.g. if $\rho(a_m, b_m)$ and $\rho(a_m, c_m)$, and we observe $\mathcal{C}_1$ at $\{a_m\}$ but $\mathcal{C}_2$ at $\{b_m, c_m\}$), but we nevertheless have that the observations are ρ-related.

For $\mathcal{O}_{ext}$, the argument is similar, except that the continuation and existential type cases can be further simplified. □

5.4.2 From Observations to Observational Equivalence

Now, we are able to compare two configurations for equivalence by checking if they have the same observations. However, this only allows us to observe the results once computation is complete, and so does not let us distinguish computations that produce distinct results. For this, we again follow the approach used in the functional setting. Our observations are the analog of the choice of a type (e.g. bool or nat) at which to observe, although the ability to observe at multiple addresses and at arbitrary types makes them more general. To compare computations that are not yet complete, we need to be able to run them. Moreover, as some computations will produce only continuations, which we would like to be able to distinguish based on *behavior* rather than some form of syntactic equality, we need an analog to running an expression in a testing context. Unlike in the functional setting, where we need to define the concept of an expression with a hole, here, configurations are already designed to interface with other configurations. The analog to an expression with a hole is therefore just another configuration. However, if we allow for testing with arbitrary configurations, we may allow too much freedom — often, we want to compare two configurations, not as a whole, but from the perspective of a program that only has access to some of their symbols. For example, two programs may each offer a range of services, stored at different addresses, and we may want to check whether one particular service is equivalent across the programs, without necessarily enforcing that the other services are as well. We will therefore define a notion of observational equivalence relative to a set of symbols S, denoting the endpoints at which testing configurations are allowed to probe.

To formalize the notion of where a testing configuration may interact with the configurations being tested, we define here formally which addresses a configuration *provides* and which ones it *uses*, as well as a notion of *internal addresses*, which may store details not visible to the outside world. This may be thought of as a weak form of typing, where we are only considering the occurrence and mode associated with symbols, not their typing information.

Definition 9. *Suppose $\mathcal{C}$ is a configuration, in which we have assigned modes to all addresses that appear (in many of our applications, this assignment comes from a typing derivation, but this notion does not inherently require that $\mathcal{C}$ is well-typed, only well-moded). We define the following set of rules, following the same structure as the configuration typing rules, to define the judgment $\mathcal{C} :: (U, P)$, representing that $\mathcal{C}$ uses addresses in the set U, and provides addresses in the set P. As in our definition of observations, we will write $\mathsf{FS}(\cdot)$ to denote the set of free symbols (addresses) in a value, continuation, or process term.*

$$\frac{}{(\cdot) :: (\cdot, \cdot)}\ \mathsf{empty} \qquad \frac{}{\mathsf{thread}(a_m, P), \mathsf{cell}(a, \_) :: (\mathsf{FS}(P), \{a_m\})}\ \mathsf{thread}$$

$$\frac{}{!_m\mathsf{cell}(a_m, V) :: (\mathsf{FS}(V), \{a_m\})}\ \mathsf{cell}^+ \qquad \frac{}{!_m\mathsf{cell}(a_m, K) :: (\mathsf{FS}(K), \{a_m\})}\ \mathsf{cell}^-$$

$$\frac{\mathcal{C}_1 :: (U_1, P_1) \quad \mathcal{C}_2 :: (U_2, P_2)}{\mathcal{C}_1, \mathcal{C}_2 :: (U_1 \cup (U_2 \setminus P_1), (P_1 \setminus U_2) \cup P_1|_C \cup P_2)}\ \mathsf{join}$$

The first four rules cover the base cases, which are straightforward — the empty configuration neither uses nor provides any addresses, while threads and cells each provide the address that they are stored at or will eventually write to. The last rule, for join, is more complex. We might initially expect that $\mathcal{C}_1, \mathcal{C}_2$ uses all symbols used by either $\mathcal{C}_1$ or $\mathcal{C}_2$, and likewise for its provided symbols, but this definition allows for some of the symbols provided by $\mathcal{C}_1$ to also be used by $\mathcal{C}_2$. Any symbol that is provided by $\mathcal{C}_1$ and used by $\mathcal{C}_2$ is no longer needed from the environment, and so is removed from the set of used variables of the overall configuration. Likewise, we remove symbols that are provided by $\mathcal{C}_1$ but used by $\mathcal{C}_2$ from the overall set of provided symbols, if they do not admit contraction. Again, these symbols are no longer externally visible, and so should not be counted as being provided.

We can then make use of this definition to specify which test configurations are allowed, enabling us to define observational equivalence.

Definition 10. *Fix an observation function $\mathcal{O}$, a set of symbols S, and a computation relation $\mapsto$ on configurations. We say that two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ satisfying $\mathcal{C}_i :: (U_i, P_i)$ are* observationally equivalent *when observed at a set S of symbols, relative to $\mathcal{O}$ and $\mapsto$ if for every configuration $\mathcal{C}' :: (U', P')$ for which:*

- *$U' \cap (P_1 \cup P_2) \subseteq S$ — that is, the only symbols used by $\mathcal{C}'$ and provided by $\mathcal{C}_1$ or $\mathcal{C}_2$ are in S.*
- *$\mathcal{C}_1, \mathcal{C}'$ and $\mathcal{C}_2, \mathcal{C}'$ are both well-formed, so $\mathcal{C}'$ does not provide any symbols that either $\mathcal{C}_1$ or $\mathcal{C}_2$ do*
- *$\mathcal{C}_1, \mathcal{C}' \mapsto^* \mathcal{C}_1'$ and $\mathcal{C}_2, \mathcal{C}' \mapsto^* \mathcal{C}_2'$, and neither $\mathcal{C}_1'$ nor $\mathcal{C}_2'$ can take further steps*

and for any set S' of symbols such that $S' \cap (P_1 \cup P_2) \subseteq S$, $\mathcal{O}(\mathcal{C}_1', S')$ and $\mathcal{O}(\mathcal{C}_2', S')$ are equal up to renaming.

Note that this definition critically relies both on the notion of observation, specified by $\mathcal{O}$, and on the notion of computation $\mapsto$. Changing either of these may (but will not necessarily) change the resulting equivalence.

For an example, we will look at an alternate observation function that defines the same equivalence as $\mathcal{O}_{ext}$ (provided the computation relation $\mapsto$ is the same in both cases).

Example 12 (Difficulties with bool)**.** *Suppose we define an observation function $\mathcal{O}_{bool}(\mathcal{C}, S)$ that only allows observation of boolean values. More precisely, we will define* $\mathsf{bool} = \oplus\{\mathsf{true} : \mathbf{1}, \mathsf{false} : \mathbf{1}\}$*, and define* $\mathcal{O}_{bool}((\mathcal{C}, !_m\mathsf{cell}(c_m, b(d_m)), \{x\}) = \{(c_m, b)\}$*, where b is either* true *or* false*. Observing an address with something other than a boolean stored yields $\{\}$, and the cases for observing at zero or multiple addresses remain the same as for $\mathcal{O}_{ext}$. This allows us to observe, for any boolean whose address is in S, whether it is true or false, and nothing further. While this is clearly a more restrictive observation, in the functional setting, observations such as these often suffice, as whenever two values differ observably, there is often an expression that can express this distinction as a boolean, which can then be observed. This remains true in the substructural setting, but can be more involved, due to the need to use all linear (and strict) addresses. Consider, for instance, the type* $\oplus\{\mathsf{left} : A, \mathsf{right} : B\}$*. If we observe in one configuration* $\mathsf{left}(a)$*, and in another* $\mathsf{right}(b)$*, both at the same address c, we can clearly distinguish the configurations at address c because the labels stored there differ. In order to distinguish these configurations with $\mathcal{O}_{bool}$, however, we need to read from c, produce a* bool *depending on whether we are in the left or right branch, and then still need to somehow use the leftover A or B. While this is possible, for instance by writing this A or B to a new cell, it is an added inconvenience when designing tests to distinguish configurations.*

Lemma 10 (Adequacy of $\mathcal{O}_{bool}$)**.** *With the standard computation relation $\mapsto$ defined in Chapter 4, observational equivalence with respect to $\mathcal{O}_{ext}$ and with respect to $\mathcal{O}_{bool}$ agree. That is, for any configurations $\mathcal{C}_1, \mathcal{C}_2$ and any set of symbols S, $\mathcal{C}_1$ and $\mathcal{C}_2$ are observationally equivalent when observed at S, relative to $\mathcal{O}_{ext}$ and $\mapsto$ if and only if they are observationally equivalent relative to $\mathcal{O}_{bool}$ and $\mapsto$.*

Proof. It is easy to see that if some $\mathcal{C}_1', \mathcal{C}_2', S'$ satisfy $\mathcal{O}_{bool}(\mathcal{C}_1', S') \neq \mathcal{O}_{bool}(\mathcal{C}_2', S')$, then they also satisfy $\mathcal{O}_{ext}(\mathcal{C}_1', S') \neq \mathcal{O}_{ext}(\mathcal{C}_2', S')$, because any observation $\mathcal{O}_{bool}$ can make can also be made by $\mathcal{O}_{ext}$, as bool is a purely positive type.

Suppose therefore that we have some $\mathcal{C}'$ such that $\mathcal{C}_i, \mathcal{C}' \mapsto^* \mathcal{C}_i'$ for $i = 1, 2$, and that neither of these configurations can take further steps, and that we have some S' such that $\mathcal{O}_{ext}(\mathcal{C}_1', S') \neq \mathcal{O}_{ext}(\mathcal{C}_2', S')$. This means that either some address c reachable from S' by following pointers contains data of types A and B with different top-level type constructors in $\mathcal{C}_1'$ and $\mathcal{C}_2'$, or contains data of $\oplus$ type with different labels ℓ_1 and ℓ_2.

In the former case, define $\mathcal{C}'' = \mathcal{C}', \mathsf{thread}(d, \mathsf{case}\ c\,(p \Rightarrow x \leftarrow x.p\ ;\ e \leftarrow d.\mathsf{true}(e)\ ;\ e.\langle\rangle))$, where p is a pattern matching the top-level constructor of A. Then, $\mathcal{C}_1, \mathcal{C}''$ evaluates to some $\mathcal{C}_1''$ containing $\mathsf{cell}(d, \mathsf{true}(e))$, while $\mathcal{C}_2, \mathcal{C}''$ evaluates to some $\mathcal{C}_2''$, in which this new thread is unable to read from c, and so is never able to write to d. As such, no filled cell at address d exists in $\mathcal{C}_2''$, and so $\mathcal{O}_{bool}(\mathcal{C}_1'', \{d\})$ is (d, true), while $\mathcal{O}_{bool}(\mathcal{C}_2'', \{d\})$ is $\{\}$, allowing us to distinguish $\mathcal{C}_1$ and $\mathcal{C}_2$.

In the latter case, we can use a similar strategy. Define $\mathcal{C}''$ to extend $\mathcal{C}'$ with a single thread that reads from c, writes true to a cell d if c contains ℓ_1, and writes false to d if c contains ℓ_2. Some extra bookkeeping is necessary as in the previous case to ensure that all data is used, so that this works in the linear case as well, but this essentially consists of copying the contents of c elsewhere, while constructing a boolean on the side. The resulting $\mathcal{C}''$ allows us to distinguish

$\mathcal{C}_1$ and $\mathcal{C}_2$ after observing at $\{d\}$. □

This example shows that we can make changes to our observation function and get the same equivalence, but also highlights that our choice of observation function can make a significant difference in how much work we need to do with our test configurations, and illustrates some of the difficulty involved in working with observational equivalence in general.

5.4.3 Typed Equivalence

This initial definition of equivalence is very general, allowing us to talk about equivalence of configurations that are ill-formed in various ways. However, as we begin to look at logical equivalence in the next section (section 5.5), we will see that it makes sense to define equivalence based on types. As such, some ill-typed configurations may be equivalent observationally, but not logically — for a concrete example, we may consider the configuration consisting of the single object $!_m\mathsf{cell}(a, \langle a, a\rangle)$. This configuration cannot be typed without running into issues of circularity, and even attempting to explore it following a type, as we will do in defining logical equivalence, has the problem that we never reach an endpoint.

To resolve this issue and try to ensure that logical and observational equivalence coincide, we will often work with a restricted form of observational equivalence, which only considers well-typed terms.

Definition 11 (Typed Observational Equivalence)**.** *If* $\Gamma \vDash \mathcal{C}_1 :: \Delta$ *and* $\Gamma \vDash \mathcal{C}_2 :: \Delta$*, we say that* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are observationally equivalent at the interface* (Γ, Δ)*, relative to an observation* $\mathcal{O}$ *and a computation relation* $\mapsto$*, if for every configuration* $\mathcal{C}'$ *(the test configuration to probe* $\mathcal{C}_1, \mathcal{C}_2$ *with) such that:*

- $\Gamma' \vDash \mathcal{C}_i, \mathcal{C}' :: \Delta'$ *for* $i = 1, 2$.
- $\mathcal{C}_i, \mathcal{C}' \mapsto \mathcal{C}_i'$ *which cannot take further steps.*

$\mathcal{O}(\mathcal{C}_1', \Delta')$ *and* $\mathcal{O}(\mathcal{C}_2', \Delta')$ *are equal up to renaming. We will denote this (leaving* $\mathcal{O}$ *and* $\mapsto$ *implicit, to be specified in the relevant context) by* $\Gamma \vDash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$.

This is, essentially, a version of our original definition with the additional restriction that the configurations used must be well-typed.

5.5 Logical Equivalence

As in the functional setting, our notion of observational equivalence is difficult to work with in practice, as it requires quantifying over a very large set — in our case, a set of configurations with some mild constraints. Our solution to this problem is likewise similar: we develop a notion of logical equivalence that is feasible to work with, and demonstrate that it coincides with observational equivalence. The key idea here is that, unlike observational equivalence, which looks at a configuration as a whole, we would like to define a more compositional equivalence, which examines smaller pieces of configurations at a time. In particular, if we have two non-equivalent configurations, we can generally find a single cell in which they differ observably, and so at least in principle, there should be no need to examine the whole configuration at once. However, trying to define this equivalence inductively over a configuration leads to difficulties, as we may need to examine the same cell multiple times, if it is referenced in several different places, and so cannot just remove a cell from consideration after examining it. Instead, we will use types to guide our exploration of a configuration, and define logical equivalence inductively over these types.

Our work on logical equivalence builds on existing work for logical relations in the functional setting, indirectly from the early work on the subject [37, 81, 89, 101, 102], and more directly from the overview given by Harper [40], the general structure of whose proofs we follow. More directly related is the proof of termination for the semi-axiomatic sequent calculus [26], which forms part of our prior work, and which informed the development of this more complex logical relation for a similar language.

There are also several other prior and concurrent lines of work on logical relations in the setting of session-typed process calculi. Pèrez, Caires, Pfenning, and Toninho [13, 72, 73] develop a theory of linear logical relations, which they apply to the calculus $\pi\mathsf{DILL}$ to prove normalization and confluence, and to reason about program equivalence. Our logical equivalence resembles theirs notationally, although the content is quite different, both due to differences in type system (in particular, our handling of non-linear modes, which may result in persistent semantic objects), and in the language itself ($\pi\mathsf{DILL}$, being a version of the π-calculus, has its semantics defined in terms of single, large process terms, as opposed to our configurations made up of semantically distinct objects). In a classical process calculus, whose semantics are given by configurations more akin to ours, Atkey [3] makes use of logical relations in a more indirect fashion, using them to connect a system of denotational semantics to operational semantics, making it possible to reason about observational equivalence via the denotational semantics. This is less closely related to our work, but, notably, does need to address non-linearity in the setting of configurations, not via persistence, but by directly reasoning about *duplicability* or *discardability* of configurations. The most closely related line of work in this direction is given by Derakhshan et al. [22] and Balzer et al. [5], dealing with non-interference and equivalence for session-typed process calculi similar to ours, albeit in a purely linear message-passing setting. Notably, both handle logical relations for *open* configurations, allowing probes of those configurations along a specified interface, although the exact details of the interfaces they use are slightly different, only allowing observation at a single channel on the right of a configuration. We will reference some of these other lines of work again, as they relate to particular aspects of our logical relation.

We now begin by addressing the inductive structure that we will use for our logical relation. Since we are reasoning about equivalence of configurations, rather than single process terms, and configuration typing is in terms of full contexts, rather than single types, we need an appropriate ordering on contexts — this will be the *multiset ordering* over the multiset of types appearing in the context, which we build up in a few steps. We first define an ordering on types $A_k \prec B_m$ capturing that A_k is a strict subformula of B_m.

Definition 12 (Subformula ordering on types)**.** *We define the non-strict ordering $A_k \preceq B_m$ inductively as follows:*

- $C_k \preceq C_k$ *— naturally, every type is a subformula of itself.*
- $C_k \preceq \oplus_{j\in J} A_m^j$ *if* $C_k \preceq A_m^\ell$ *for some* $\ell \in J$.
- $C_k \preceq \&_{j\in J} A_m^j$ *if* $C_k \preceq A_m^\ell$ *for some* $\ell \in J$.
- $C_k \preceq A_m \otimes B_m$ *if* $C_k \preceq A_m$ *or* $C_k \preceq B_m$.

- $C_k \preceq A_m \multimap B_m$ *if* $C_k \preceq A_m$ *or* $C_k \preceq B_m$.
- $C_k \preceq {\downarrow}^r_m A_r$ *if* $C_k \preceq A_m$.
- $C_k \preceq {\uparrow}^m_r A_r$ *if* $C_k \preceq A_m$.

The first case serves as a base case, while the others allow us to recursively search through the structure of a type, looking for instances of B_m. *With this defined, we can take the strict ordering* $A_k \prec B_m$ *to be defined by* $A_k \preceq B_m$ *and* $A_k \neq B_m$.

Now, we extend this to multisets of types, using a general construction due to Dershowitz and Manna [23].

Definition 13 (Multiset ordering). *Suppose that* $\prec$ *is a well-founded strict partial order over a base set* S, *and that* M, N *are multisets whose elements are drawn from* S.

We say that M *is smaller than* N *in the multiset ordering, which we will write* $M \ll N$, *iff there exist finite multisets* X, Y *whose elements are drawn from* S *such that:*

- X *is a non-empty submultiset of* M.
- $N = (M \setminus X) + Y$, *where* $+$ *represents the union of multisets.*
- *For all* $y \in Y$, *there is some* $x \in X$ *with* $y \prec x$.

That this ordering is well-founded is also a result of Dershowitz and Manna [23]. While this allows us to very generally compare multisets of types, just as in induction over the naturals one often only moves from $n + 1$ to n, we will generally rely on two particular ways to shrink a multiset M of types to a smaller multiset N in our induction arguments. Firstly, we can remove an element A_m from M, taking $N = M \setminus \{A_m\}$. In this case, taking $X = \{A_m\}$ and $Y = \emptyset$, it is easy to see that $<\ll N$, as the first two conditions are immediate from our choices of X and Y, and the third holds vacuously, since Y is empty. Secondly, and more importantly (as the first example works as well under the usual subset relation), we can choose an element A_m from M, and replace it with some finite set of its subformulae. More precisely, we take $X = \{A_m\}$ and let Y be some multiset of subformulae of A_m. Taking $N = (M \setminus X) + Y$, the first two conditions are again immediate, and the third follows from our choice of Y to only contain subformulae of A_m, and so this operation also yields a smaller multiset. We will generally apply this to replace a formula with its immediate subformulae (e.g., replacing $\{A_m \multimap B_m\}$ with $\{A_m, B_m\}$).

Now, we will define our equivalence as a family of relations

$$\mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$$

where $\mathcal{C}_1, \mathcal{C}_2$ are configurations and Δ is a context by induction over Δ, using the multiset ordering on its multiset of types. We can then extend this to relations

$$\Gamma \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$$

where Γ is also a context of typed symbols, allowing us to talk about the equivalence of *open* configurations which depend on the symbols in Γ as well as closed configurations.

5.5.1 Equivalence at Variable Types

In order to work compositionally with quantified types $\forall t_m.\tau$ and $\exists t_m.\tau$, we will need some way to talk about equivalence at a *type variable* t_m. A standard approach here, following from

Girard's method of candidates [37], but applied in the setting of equivalence rather than termination (see, e.g., [89]), is to interpret type variables as relations: a type variable may have different implementation types A_m and B_m in configurations $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively, and we determine whether two terms of these implementation types should be equivalent based on a particular relation R. In the case of an existential type, we say that two configurations are equivalent if there exists a suitable relation under which the underlying implementations are equivalent, and for universal types, we require that for any suitable choice of relation on the inputs, the resulting outputs are equivalent.

Formally, we will work with relations R between address-configuration pairs $(a, \mathcal{C})$. Intuitively, we think of $(a_1, \mathcal{C}_1)R(a_2, \mathcal{C}_2)$ as saying that the data stored at a_1 in $\mathcal{C}_1$ and the data stored at a_2 in $\mathcal{C}_2$ represent the same underlying concept in two different ways. Not all such relations are suitable, however — if we allow arbitrary relations, then any two implementations of an existential type could be said to be equivalent, even if they can be distinguished by observation. We will therefore work with a notion of *admissible relations*,[9] which are chosen to ensure that logical equivalence and observational equivalence will agree (once suitably defined). Our conditions on admissible relations are, more or less, standard — in the related work of Caires et al. [13] on parametricity for $\pi\mathsf{DILL}$, essentially the same three conditions are given (although, as they note, one, backwards closure, can be shown to be redundant, but is nevertheless useful to make explicit).

Definition 14. *We say that a relation R on address-configuration pairs is an* admissible relation between types A^1_m and A^2_m, *written $R : A^1_m \leftrightarrow A^2_m$, if:*

1. *R only relates pairs $(a_i, \mathcal{C}_i)$ where the configurations $\mathcal{C}_i$ are well-typed and provide $a_i : A^i_m$. That is, if $(a_1, \mathcal{C}_1)\ R\ (a_2, \mathcal{C}_2)$, then for $i \in \{1, 2\}$, we have that there is some context Δ_i such that $(\cdot) \vDash \mathcal{C}_i :: \Delta_i, (a_i : A^i_m)$.*
2. *R is closed under converse reduction (often called head expansion). That is, whenever $(a_1, \mathcal{C}_1)R(a_2, \mathcal{C}_2)$ and $\mathcal{C}'_1 \mapsto \mathcal{C}_1$ and $\mathcal{C}'_2 \mapsto \mathcal{C}_2$, both $(a_1, \mathcal{C}'_1)R(a_2, \mathcal{C}_2)$ and $(a_1, \mathcal{C}_1)R(a_2, \mathcal{C}'_2)$ also hold. This property is often referred to as* backwards closure.
3. *R respects observational equivalence. That is, if $(a_1, \mathcal{C}_1)R(a_2, \mathcal{C}_2)$ and $\mathcal{C}_1$ is observationally equivalent to $\mathcal{C}'_1$ at a_1 and $\mathcal{C}_2$ is observationally equivalent to $\mathcal{C}'_2$ at a_2, then $(a_1, \mathcal{C}'_1)R(a_2, \mathcal{C}'_2)$.*

Note that in particular, this last item implies that if $(a_1, \mathcal{C}_1)R(a_2, \mathcal{C}_2)$, then $(a_1, (\mathcal{C}, \mathcal{C}_1))R(a_2, (\mathcal{C}, \mathcal{C}_2))$ whenever both $\mathcal{C}, \mathcal{C}_1$ and $\mathcal{C}, \mathcal{C}_2$ are well-formed configurations.

One useful consequence of the definition of admissible relations is that related configurations can be extended (almost) arbitrarily.

Lemma 11. *Suppose R is an admissible relation, and $(a_1, \mathcal{C}_1)\, R\, (a_2, \mathcal{C}_2)$. Given $\mathcal{F}'_1, \mathcal{F}'_2$ final such that $\mathcal{C}_1, \mathcal{F}'_1$ and $\mathcal{C}_2, \mathcal{F}'_2$ are well-formed, it is also the case that $(a_1, (\mathcal{C}_1, \mathcal{F}'_1))\ R\ (a_2, (\mathcal{C}_2, \mathcal{F}'_2))$.*

Proof. Since R respects observational equivalence, it will suffice to show that $\mathcal{C}_1, \mathcal{F}'_1$ is observationally equivalent to $\mathcal{C}_1$ at a_1 (and that a similar statement holds for $\mathcal{C}_2$ and $\mathcal{F}'_2$).

Because R is admissible, we know that $(\cdot) \vDash \mathcal{C}_1 :: \Delta, (a_1 : A_m)$ for some Δ and A_m. Write $\mathcal{C}_1 :: (\cdot, P_1)$, and $\mathcal{F}'_1 :: (U'_1, P'_1)$, noting that P_1 contains a_1. Now, fix some $\mathcal{C}' :: (S, T)$, where $S \cap (P_1 \cup P'_1) = \{a_1\}$, so the only symbol provided by $\mathcal{C}_1, \mathcal{F}'_1$ and used by $\mathcal{C}'$ is a_1. We require also

[9] Also known as *candidates* in some of the literature on logical relations

(as in the definition of observational equivalence) that $T \cap (P_1 \cup P_1')$ is empty, so that $\mathcal{C}', \mathcal{C}_1, \mathcal{F}_1'$ is well-formed.

Let $\mathcal{C}', \mathcal{C}_1 \mapsto^* \mathcal{D}_1$ such that $\mathcal{D}_1$ cannot take further steps. We then observe that $\mathcal{C}', \mathcal{C}_1, \mathcal{F}_1' \mapsto^* \mathcal{D}_1, \mathcal{F}_1'$, and, as $\mathcal{F}_1'$ is final, this cannot take further steps. Write $\mathcal{D}_2 = \mathcal{D}_1, \mathcal{F}_1'$.

Choose a set S' of symbols with $S' \cap (P_1 \cup P_1') = \{a_1\}$, and consider $\mathcal{O}_{ext}(\mathcal{D}_i, S')$. We want to show that these two observations are equal up to renaming (indeed, we claim that they are equal).

Let $\{\alpha, \beta\} = \{1, 2\}$, and suppose that $!_{m_0}\mathsf{cell}(c^0_{m_0}, V^0)$ occurs in $\mathcal{O}_{ext}(\mathcal{D}_\alpha, S')$, but does not occur in $\mathcal{O}_{ext}(\mathcal{D}_\beta, S')$. Inverting the definition of $\mathcal{O}_{ext}$, we see that we must have attempted to observe $\mathcal{D}_\alpha$ at $c^0_{m_0}$. Either $c^0_{m_0}$ is in S', or we previously observed some cell at address $c^1_{m_1}$ whose value V^1 contained $c^0_{m_0}$ as a free symbol. Iterating this reasoning, we can find a sequence $c^0_{m_0}, c^1_{m_1}, \ldots, c^n_{m_n}$ such that:

- For each $0 \leq j \leq n$, $\mathcal{D}_\alpha$ contains $!_{m_j}\mathsf{cell}(c^j_{m_j}, V^j)$
- V^j contains $c^{j-1}_{m_{j-1}}$ as a free symbol
- $c^n_{m_n} \in S'$

Now, we consider observing $\mathcal{D}_\beta$, starting from $c^n_{m_n}$. There is some smallest j such that $\mathcal{D}_\beta$ contains $!_{m_j}\mathsf{cell}(c^j_{m_j}, V^j)$, We note that the value V^j observed here must be the same as in $\mathcal{D}_\alpha$, as cells are not mutable, and the preconditions of the lemma ensure that $\mathcal{C}_1$ and $\mathcal{F}_1'$ and $\mathcal{C}'$ never provide the same address, so we do not find two conflicting cells, one from $\mathcal{D}_1$ and one from $\mathcal{F}_1'$. This means, in particular, that $c^{j-1}_{m_{j-1}}$ is still free in V_j in $\mathcal{D}_\beta$, and so the observation $\mathcal{O}_{ext}(\mathcal{D}_\beta, S')$ must contain $\mathcal{O}_{ext}(\mathcal{D}_\beta, c^{j-1}_{m_{j-1}})$. For this to fail to contain the cell $!_{m_{j-1}}\mathsf{cell}(c^{j-1}_{m_{j-1}})$ that we know exists in $\mathcal{D}_\alpha$, it must be the case that $\mathcal{D}_\beta$ does not provide $c^{j-1}_{m_{j-1}}$, as we have already established that any cell occurring in both $\mathcal{D}_\alpha$ and $\mathcal{D}_\beta$ must have the same contents in both. Since we know that the cell exists in $\mathcal{D}_\alpha$, but not in $\mathcal{D}_\beta$, it must be the case that $\alpha = 2$ and the cell exists in $\mathcal{F}_1'$. We will show that this is impossible, based on the constraint that $\mathcal{C}_1, \mathcal{F}_1'$ is well-formed, and the condition (given by the definition of observational equivalence) that $\mathcal{C}_1, \mathcal{F}_1', \mathcal{C}'$ is well-formed.

Since the cell at $c^j_{m_j}$ occurs in $\mathcal{D}_1$ already, and mentions the address $c^{j-1}_{m_{j-1}}$, either $\mathcal{D}_1$ must contain a thread providing this address, or $\mathcal{C}_1, \mathcal{C}'$ already was using this address. Since $\mathcal{C}_1$ uses no addresses (being closed), and the only address that $\mathcal{C}'$ is allowed to use that $\mathcal{C}_1, \mathcal{F}_1'$ provides is a_1, it must be the case that $\mathcal{D}_1$ contains a thread providing $c^{j-1}_{m_{j-1}}$. Now, this thread must be blocked trying to read from some address d, as we know that $\mathcal{D}_1$ cannot step further. It may be the case that d is an internal address of $\mathcal{C}_1, \mathcal{C}'$, not visible externally, but this just pushes us back to the existence of another thread, blocked on some address d'. Iterating this reasoning, we find that the only way to enable $c^{j-1}_{m_{j-1}}$ to be written to is if some address $\hat{d}$ used by $\mathcal{C}_1, \mathcal{C}'$ (and therefore not provided by $\mathcal{F}_1'$) is written to, allowing this chain of threads to make progress. Since $\hat{d}$ cannot be provided by $\mathcal{F}_1'$, $\mathcal{D}_2$ cannot contain such a cell, and we conclude that our initial assumption that the observations differed must have been false. □

Definition 15. *If Ξ is a context of type variables, we write $\xi : \Xi$ to denote that ξ is a substitution assigning to each type variable t_m in Ξ a closed type A_m.*

Given $\xi_1, \xi_2 : \Xi$, we write $\eta : \xi_1 \leftrightarrow \xi_2$ to denote that η is a family of admissible relations, one for each type variable t_m in Ξ, such that $\eta(t_m) : \xi_1(t_m) \leftrightarrow \xi_2(t_m)$.

5.5.2 Formalizing Logical Equivalence

Now that we have a way to handle type variables, and thereby also quantified types, we can begin defining logical equivalence. Recalling that we want logical equivalence to match observational equivalence (in this case, focusing on the extensional version of observational equivalence), it is natural to define logical equivalence inductively over a *context* of types and symbols, rather than over a single type, as is common in the functional setting. This also matches our definition of configuration typing, which likewise focuses on contexts rather than single types. Our judgment for logical equivalence mirrors that for typing, but with two configurations rather than one. We write

$$\Xi \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$$

to denote that $\mathcal{C}_1$ is logically equivalent to $\mathcal{C}_2$ when examined following Δ, using type variables in Ξ. Note that Δ describes what symbols or addresses we may begin examining $\mathcal{C}_1$ and $\mathcal{C}_2$ at to look for differences, but also tells us what shape of data we should expect to find, based on the types in Δ. It may be more natural to use two separate contexts, with some mapping between the symbols of one and the symbols of the other, but since we consider configurations that are related by renaming to be the same, we prefer to implicitly rename both $\mathcal{C}_i$ to match Δ — we can think of this as saying that two configurations are logically equivalent at Δ if there exist renamings ρ_1 and ρ_2 such that the configurations $\rho_i\mathcal{C}_i$ are equivalent under the formal rules we give.

We also note that it may seem simpler to look only at one symbol at a time, defining a judgment

$$\Xi \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: (a : A_m).$$

However, this approach leads to problems in the substructural case, where two observationally distinct configurations would nevertheless be treated as logically equivalent.

Definition 16 ((Extensional) Logical Equivalence)**.** *We define the logical equivalence*

$$\Xi \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$$

to hold if, for all $\xi_1, \xi_2 : \Xi$ and all $\eta : \xi_1 \leftrightarrow \xi_2$, $\eta \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$. That is, we fix an interpretation η of the type variables in Ξ, and then examine $\mathcal{C}_1$ and $\mathcal{C}_2$ in this context.

Since we identify configurations that are equal up to renaming, this definition implicitly allows us to rename $\mathcal{C}_1$ and $\mathcal{C}_2$ as it is convenient.

We then define $\eta \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ if each $\mathcal{C}_i \mapsto^* \mathcal{F}_i$ final (i.e. consisting only of filled cells) such that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$. That is, general configurations are compared for equivalence by evaluating them and comparing the resulting final configurations (analogous to values in a functional setting) for equivalence. This is analogous to the expression *or* term *interpretation of a type (or context, or sequent) in logical relations for functional languages, while the judgment over final configurations is analogous to the* value *interpretation of a type.*

We define this new judgment $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$ inductively over Δ, making use of $\eta \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ at some points. To be precise, we define the two judgments (1) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$ and (2) $\eta \; ; \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ by mutual induction on the pair (Δ, i) ordered lexicographically, where i is the number of the judgment being defined. That is, in defining the judgment (2) on general configurations, we may make use of the judgment (1) on final configurations at the same Δ, and in defining (1), we may make use of (2) on strictly smaller Δ.

In several cases we will need to make use of different configurations depending on whether a particular mode m admits contraction or not. We write $C_m?(\mathcal{C}_1, \mathcal{C}_2)$ to denote $\mathcal{C}_1$ if $\mathsf{C} \in \sigma(m)$, and $\mathcal{C}_2$ otherwise.

(1) If $\Delta = (\cdot)$, then we cannot observe anything about the configurations, and so they are equivalent — $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (\cdot)$ always holds.

(2) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : t_m), \Delta'$ if we can write each $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that

- *$\mathcal{F}_i^2 = \mathcal{F}_i|_C$ for $i = 1, 2$.*
- *$(a, (\mathcal{F}_1^1, \mathcal{F}_1^2)) \; \eta(t_m) \; (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$.*
- *$\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$.*

That is, equivalence at a type variable t_m is interpreted by the (admissible) relation $\eta(t_m)$.

(3) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \oplus\{\ell : A_m^\ell\}_{\ell \in L}), \Delta'$ if there is $j \in L$ such that

- *$\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, j(b))$*
- *$\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : A_m^j), \Delta'$.*

(4) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : B_m \otimes C_m), \Delta'$ if

- *$\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle b, c \rangle)$*
- *$\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : B_m), (c : C_m), \Delta'$*

(5) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \mathbf{1}_m), \Delta'$ if

- *$\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle\rangle)$*
- *$\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Delta'$*

(6) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \downarrow_m^k A_k), \Delta'$ if

- *$\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \mathsf{shift}(b))$*
- *$\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : A_k), \Delta'$.*

(7) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \exists t_m.B_m), \Delta'$ if

- *$\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle A_m^i, b \rangle)$*
- *There exists some $R : A_m^1 \leftrightarrow A_m^2$ such that $\eta, t_m \hookrightarrow R \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : B_m), \Delta'$*

(8) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \&\{\ell : A_m^\ell\}_{\ell \in L}), \Delta'$ if

- *For each $j \in L$, $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.j(b)) \sim \mathcal{F}_2, \mathsf{thread}(b, a.j(b)) :: (b : A_m^j), \Delta'$.*

That is, the configurations resulting from taking any fixed branch of the choice and evaluating are equivalent.

(9) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : B_m \multimap C_m), \Delta'$ if

- *Whenever $\eta \; ; \cdot \vdash \mathcal{F}_1|_C, \mathcal{F}_1' \sim \mathcal{F}_2|_C, \mathcal{F}_2' :: (b : B_m)$, it also holds that $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c \rangle) \sim \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c \rangle) :: (c : C_m), \Delta'$*

(10) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \uparrow_k^m A_k), \Delta'$ if

- *$\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\mathsf{shift}(b)) \sim \mathcal{F}_2, \mathsf{thread}(b, a.\mathsf{shift}(b)) :: (b : A_k), \Delta'$*

(11) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : \forall t_m.B_m), \Delta'$ if

- *For any choice of A^1_m, A^2_m, and $R : A^1_m \leftrightarrow A^2_m$, it holds that* $\eta, t_m \hookrightarrow R \,;\, \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\langle A^1_m, b\rangle) \sim \mathcal{F}_2, \mathsf{thread}(b, a.\langle A^2_m, b\rangle) :: (b : B_m), \Delta'$

Interestingly, the cases other than $\exists$, $\forall$, and type variables can all be captured by two generic rules, one for positive types and one for negative types:

- $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A^+_m), \Delta'$ if $\mathcal{F}_i = \mathcal{F}'_i, !_m\mathsf{cell}(a, V)$, there is some Γ such that $\Gamma \vdash !_m\mathsf{cell}(a, V) :: (a : A^+_m)$, and $\eta \,;\, C?(\mathcal{F}_1, \mathcal{F}'_1) \sim C?(\mathcal{F}_2, \mathcal{F}'_2) :: \Gamma, \Delta'$.
- $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A^-_m), \Delta'$ if whenever $\Gamma, (a : A^-_m) \vdash \mathsf{thread}(c, a.V) :: (c : C_k)$ and $\mathcal{F}'_1, \mathcal{F}'_2$ are such that $\eta \,;\, \cdot \vdash \mathcal{F}_1|_C, \mathcal{F}'_1 \sim \mathcal{F}_2|_C, \mathcal{F}'_2 :: \Gamma, (a : A^-_m)$, it is also the case that $\eta \,;\, \cdot \vdash \mathcal{F}_1, \mathcal{F}'_1, \mathsf{thread}(c, a.V) \sim \mathcal{F}_2, \mathcal{F}'_2, \mathsf{thread}(c, a.V) :: (c : C_k), \Delta'$

While the compactness of this definition is convenient for presentation, for ease of use (and to handle quantifiers), we prefer the expanded definition where each type is addressed separately. It would, however, be an interesting item of future work to see if there is a natural way to extend this to quantifiers as well — certainly the case for positive types has the flavor of an existential (there is some Γ...), while the case for negative types is more universal.

This definition matches the intuition described above — at each step, we perform a one-level observation of the configuration. For positive types, this consists of directly looking at a cell in each configuration, comparing the values stored within, and then making some number of recursive calls to the definition. For negative types, we instead need to add on to the configuration in order to probe them further, and after evaluation, we are left with new final configurations to check for equivalence.

Note that this definition, like that for observational equivalence, relies critically on the computation relation $\mapsto$. Similarly, it relies on some concept of observation, though here it is more difficult to make this generic. In the definition we give above, we have allowed values to be observed directly, while continuations can only be observed by providing input to them and examining the output, following the same intuition as the function $\mathcal{O}_{ext}$ used in defining extensional equivalence observationally. An interesting avenue of future work would be to define a notion of observation over which logical equivalence can be parameterized, and then to relate this to the notion of observation we used in defining observational equivalence, in order to generically prove, for example, that logical and observational equivalence coincide for a wide range of choices of observation function and computation relation.

5.5.3 Open Configurations

Our definition for logical equivalence extends easily to open configurations, which depend on some input, as well. We define

$$\Xi \,;\, \Gamma \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$$

to hold if, for any $\mathcal{F}'_1, \mathcal{F}'_2$ final such that

$$\Xi \,;\, \cdot \vDash \mathcal{F}'_1 \sim \mathcal{F}'_2 :: \Gamma,$$

it also holds that

$$\Xi \,;\, \cdot \vDash \mathcal{C}_1, \mathcal{F}'_1 \sim \mathcal{C}_2, \mathcal{F}'_2 :: \Delta.$$

That is, if $\mathcal{C}_1, \mathcal{C}_2$ are provided with related final configurations for their input Γ, they yield (after evaluation) related final configurations at Δ. This will then allow us to consider the equivalence (or non-equivalence) of a wider range of configurations.

5.5.4 Ill-typed configurations

An interesting observation about these definitions of equivalence is that neither requires that the configurations under consideration are well-typed. While logical equivalence makes use of types as a measure to ensure that the relation is well-defined, in principle, two configurations can be equivalent at some context Δ without actually being well-typed at Δ. This follows the general idea of *semantic typing* [27], where a system of typing is defined based on correct behavior of programs with respect to the given type. Logical equivalence is a form of semantic type, in this sense — we might say that a configuration that is logically equivalent to itself at some type Δ is semantically well-typed, in that, when run in a context interacting according to the protocols specified by Δ, nothing will go wrong.

This allows for some interesting possibilities, especially in combination with modified computation rules that allow some particular ill-typed configurations to maintain progress. For instance, a rule which reads from a linear cell without consuming it could allow for the sharing of such cells between multiple users despite their linearity, perhaps as a memory footprint optimization.

5.5.5 Counterexample for looking at single variables

If we choose to define logical equivalence by looking at single variables rather than contexts, some issues arise with substructurality.

Suppose we define (for this section only)

$$\eta \;;\; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$$

if $\eta \;;\; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m)$ for each $a : A_m$ in Δ. We can then define this single-variable equivalence in much the same way as we defined equivalence originally — it is analogous to the case where Δ', the remainder of the context we are examining, is empty. However, one of our rules, for equivalence at product types, makes use of a non-singleton context in its definition, and so we would need to modify this case. The most natural definition in this context is to say that $\eta \;;\; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : B_m \otimes C_m)$ if $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle b, c\rangle)$ and $\eta \;;\; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (b : B_m)$ and $\eta \;;\; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (c : C_m)$. The critical difference here is that we are separately comparing $\mathcal{F}_1$ and $\mathcal{F}_2$ for equivalence at $b : B_m$ and $c : C_m$, and so the same linear cell may get used both in checking equivalence at $b : B_m$ and at $c : C_m$.[10] For a simple example, suppose that we have some $\mathcal{C}[x]$ providing some $x : A_m$ (writing $\mathcal{C}[b]$ to represent $[b/x](\mathcal{C}[x])$) and take $\mathcal{C}_1 = \mathcal{C}[b], \mathcal{C}[c], !_m\mathsf{cell}(a, \langle b, c\rangle)$ and $\mathcal{C}_2 = \mathcal{C}[d], !_m\mathsf{cell}(a, \langle d, d\rangle)$. Now, if m is linear (or affine), these two configurations are observationally distinct. A process that attempts to read from a and then from its two components will succeed on $\mathcal{C}_1$, potentially producing some output, while when run on $\mathcal{C}_2$, the cell at address d will be consumed the first time it is read from, causing the

[10] Restricting to only well-typed configurations prevents this problem, but rules out some interesting examples.

process to get stuck upon trying to read from d a second time, producing no output. This process therefore allows us to distinguish the two configurations, which would be logically equivalent under the definition presented in this section. The definition that we take, in terms of whole contexts, avoids this problem, because linear cells, once read from as part of the equivalence checking threadedure, are removed from the configurations that then get tested for equivalence at the remainder of the context.

5.5.6 Results on Logical Equivalence

The key result that we would like to show is that logical and observational equivalence coincide for well-typed configurations.[11] In particular, the formulation of logical equivalence that we present in section 5.5 defines the same relation as extensional observational equivalence, in a way that we will formalize in this section. We will follow a standard approach to showing this type of result, beginning by proving what is often called the parametricity or abstraction theorem [89] for logical equivalence — that all well-typed configurations are logically equivalent to themselves. We then show that logical equivalence respects observational equivalence, and it follows quickly that observational equivalence implies logical equivalence.

For the converse direction, we will define what it means for a relation to be a consistent congruence, and then show that observational equivalence is the coarsest such congruence (relative to a choice of observation). We can then show that logical equivalence implies observational equivalence by proving that logical equivalence is a consistent congruence, and that it is therefore contained in the coarsest such congruence.

Since, in this section, we are working only with extensional observational equivalence, we will write $\Gamma \vDash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$ to denote extensional observational equivalence.

We will now begin by showing some general properties of logical equivalence that will be useful throughout the proofs.

General Properties of Logical Equivalence

Our first general property, inversion, allows us to conclude from the fact that a pair of configurations are equivalent at some $(a : A_m)$ that they both contain a corresponding cell at address a — a minor detail, but which is necessary for technical reasons.

Lemma 12 (Inversion). *If $\eta \; ; \; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m), \Delta'$, then $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, D_i)$ for $i \in \{1, 2\}$.*

Proof. For positive types A_m^+, this is immediate from the definition of logical equivalence. For negative types A_m^-, we note that equivalence requires that each $\mathcal{F}_i$, augmented with a process attempting to read from a, is able to reach a final configuration. As such, $\mathcal{F}_i$ must contain a filled cell at address a — otherwise, the process attempting to read from a would block, and since no other processes exist in this configuration, it would be unable to reach a final state. □

[11] Ideally, we would like to have a simple condition specifying when, even for ill-typed configurations, logical and observational equivalence coincide, but it is not entirely clear what such a condition looks like.

We then establish that logical equivalence is a partial equivalence relation (that is, it is symmetric and transitive). A limited form of reflexivity, for well-typed configurations, will come from parametricity (Theorem 30). We also establish that logical equivalence is closed under converse reduction, which will allow us to more easily connect the definitions of logical equivalence for final and non-final configurations.

Lemma 13 (Symmetry)**.** *Suppose* $\Xi \,;\, \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$*. Then also* $\Xi \,;\, \cdot \vdash \mathcal{C}_2 \sim \mathcal{C}_1 :: \Delta$*.*

Proof. By induction on Δ. In all but the type variable case, the definition is symmetric, and in the type variable case, we take the inverse of the given relation. □

Lemma 14 (Transitivity)**.** *Suppose* $\Xi \,;\, \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ *and* $\Xi \,;\, \cdot \vdash \mathcal{C}_2 \sim \mathcal{C}_3 :: \Delta$*. Then also* $\Xi \,;\, \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_3 :: \Delta$*.*

Proof. By induction on Δ. In the type variable case, we compose the two given relations, while the other cases are immediate. □

Lemma 15 (Closure under Converse Reduction)**.** *Suppose* $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$*, and that* $\mathcal{C}_1' \mapsto \mathcal{C}_1$ *and* $\mathcal{C}_2' \mapsto \mathcal{C}_2$*. Then:*

- $\Xi \,;\, \Gamma \vdash \mathcal{C}_1' \sim \mathcal{C}_2 :: \Delta$.
- $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_2' :: \Delta$.

Proof. This is almost immediate — since reduction is confluent, the fact that $\mathcal{C}_1' \mapsto \mathcal{C}_1$ means that if $\mathcal{C}_1 \mapsto^* \mathcal{F}_1$ final, then also $\mathcal{C}_1' \mapsto^* \mathcal{F}_1$, and likewise for $\mathcal{C}_2$. The comparison for logical equivalence then takes place on these final configurations, which are the same in all three listed equivalences of configurations. □

The next several properties that we examine, Configuration Extension, Weakening, and Contraction, deal with adding (or removing) irrelevant parts of either the configurations being compared for equivalence, or the context at which they are being compared. Naturally, if two configurations are equivalent, they should remain equivalent upon the addition of equal things. Likewise, if two configurations are equivalent at a given context, they should also be equivalent at any sub-context, or at the same context after global renamings. The contraction lemma additionally allows for a cell to be duplicated in the configurations and its corresponding type to be duplicated in the contexts, while retaining equality. This can be thought of as a realization of the logical rule of contraction in the context of equivalence.

Lemma 16 (Configuration Extension)**.** *If* $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$*, then for any* $\mathcal{F}_1'$*,* $\mathcal{F}_2'$ *such that* $\mathcal{F}_i, \mathcal{F}_i'$ *is well-formed for each* $i \in \{1, 2\}$*, it also holds that* $\eta \,;\, \cdot \vdash \mathcal{F}_1, \mathcal{F}_1' \sim \mathcal{F}_2, \mathcal{F}_2' :: \Delta$*.*

Proof. By induction over the derivation of $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$. In each case other than t_m and $(\cdot)$, we apply the inductive hypothesis and continue. In the case of $(\cdot)$, the result is immediate — any two final configurations are equivalent at the empty context. In the case of t_m, we rely on admissibility of $\eta(t_m)$, which, among other things, ensures that if $(a, \mathcal{C}_1)\ \eta(t_m)\ (a, \mathcal{C}_2)$, then also $(a, (\mathcal{C}_1, \mathcal{F}_1'))\ \eta(t_m)\ (a, (\mathcal{C}_2, \mathcal{F}_2'))$ (see Lemma 11). □

Lemma 17 (Weakening)**.** *Suppose* $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta_1, \Delta_2$*. Then also* $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta_1$*.*
Likewise, if $\eta \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta_1, \Delta_2$*, then* $\eta \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta_1$*.*

Proof. By induction on Δ_1.

If Δ_1 is empty, then the result is immediate.

In all other cases, $\Delta_1 = (a : A_m), \Delta_1'$ for some A_m, and the result follows by applying the definition of equivalence at A_m, applying the inductive hypothesis to the resulting conclusion to remove Δ_2, and then applying the definition of equivalence again to rebuild an equivalence at Δ_1.

For non-final configurations, we evaluate to reach a final configuration, apply the result for final configurations, and then use (repeated) closure under converse reduction (Lemma 15). □

Lemma 18 (Contraction). *Suppose $\eta \ ; \ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m), \Delta'$, so, by Inversion, for $i \in \{1, 2\}$, $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, D_i)$. Then:*

- *If b is a fresh symbol and $\mathsf{C} \in \sigma(m)$, then also*
  $\eta \ ; \cdot \vdash \mathcal{F}_1, !_m\mathsf{cell}(b, D_1) \sim \mathcal{F}_2, !_m\mathsf{cell}(b, D_2) :: (a : A_m), (b : A_m), \Delta'$.
- *If b is a fresh symbol and $C \notin \sigma(m)$, then*
  $\eta \ ; \cdot \vdash \mathcal{F}_1', !_m\mathsf{cell}(b, D_1) \sim \mathcal{F}_2', !_m\mathsf{cell}(b, D_2) :: (b : A_m), \Delta'$.
- *Additionally, if $\mathsf{C} \in \sigma(m)$, then*
  $\eta \ ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m), \Delta'$ iff $\eta \ ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m), (a : A_m), \Delta'$.

Proof. The first point follows from lemma 16, using that $\mathsf{C} \in \sigma(m)$ to get that $\mathcal{F}_i, !_m\mathsf{cell}(b, D_i)$ is well-formed for each $i \in \{1, 2\}$.

For the second point, we distinguish cases for the type A_m.

If A_m is a positive type other than an existential, we apply the definition of equivalence at that type to get that $\eta \ ; \ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta_V, \Delta'$ for some Δ_V depending on the cell contents D_1, D_2. This is then exactly what we need to apply the definition of equivalence again (now at c instead of a), giving the desired result.

If $A_m = \exists t_m.B_m$, then we have that there exists some R such that

$$\eta, t_m \hookrightarrow R \ ; \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: (b : B_m), \Delta'.$$

As in the other positive cases, applying the definition of equivalence again gives the result.

If $A_m = \&\{\ell : A_m^\ell\}_{\ell \in L}$, then for each $\ell \in L$, we have that

$$\eta \ ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\ell(b)) \sim \mathcal{F}_2, \mathsf{thread}(b, a.\ell(b)) :: (b : A_m^\ell), \Delta'.$$

Since these configurations $\mathcal{F}_i, \mathsf{thread}(b, a.\ell(b))$ can be evaluated to final configurations, we get that the cells $\mathsf{cell}(a, D_i)$ are of the form $\mathsf{cell}(a, \{j(y) \Rightarrow P_i^j\}_{j \in L})$ (otherwise, the configuration would be stuck). We then have that $\mathcal{F}_i, \mathsf{thread}(b, a.\ell(b)) \mapsto \mathcal{F}_i', \mathsf{thread}(b, P_i^\ell[b/y])$, and (since equivalence is closed under forwards reduction by definition) that

$$\eta \ ; \cdot \vDash \mathcal{F}_1', \mathsf{thread}(b, P_1^\ell[b/y]) \sim \mathcal{F}_2', \mathsf{thread}(b, P_2^\ell[b/y]) :: (b : A_m^\ell), \Delta'.$$

Inverting these steps, using c in place of a, we get the desired result. This makes use of closure under converse reduction and the definition of equivalence at $\&$.

We do not show here the other cases for negative types, but they are similar, applying the definition of equivalence at that type, noting that we must be able to take a step, so the cell data D_i has a suitable form, and then observing that after the step, the cell at address a no longer appears, and so replacing a with c yields the same resulting configuration, allowing us to invert the initial steps.

For the third point, we proceed by induction on $\Delta = (a : A_m)$, viewed as a context, under the multiset ordering. In fact, this will show more generally that if $\mathsf{C} \in \sigma(\Gamma)$, then

$$\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Delta' \qquad \text{iff} \qquad \eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Gamma, \Delta',$$

and similarly for general, non-final configurations as well.

If $\Gamma = (\cdot)$, then we are immediately done.

If $\Gamma = (a : t_m), \Gamma'$, then $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Delta'$ if and only if we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
2. $(a, (\mathcal{F}_1^1, \mathcal{F}_1^2)) \; \eta(t_m) \; (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$
3. $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma', \Delta'$.

Since $\mathsf{C} \in \sigma(\Gamma)$, in particular, $\mathsf{C} \in \sigma(m)$, and so $\mathcal{F}_i^1$ must be empty and $\mathcal{F}_i^2, \mathcal{F}_i^3 = \mathcal{F}_i$. We now apply the inductive hypothesis at Γ', giving that

$$\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma', \Delta' \qquad \text{iff} \qquad \eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma', \Gamma', \Delta'.$$

Now, we note that the same "split" of $\mathcal{F}_i$ into parts remains possible, and indeed, we can do this split twice, as equivalence at a only requires the contractible part of $\mathcal{F}_i$. Tracing through the definition of equivalence at $(a : t_m)$ twice, we get that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma', \Gamma', \Delta'$ iff $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Gamma, \Delta'$.

We show $\oplus$ and $\multimap$ as illustrative positive and negative cases, respectively. The remaining cases are similar to one shown.

Suppose $\Gamma = (a : \oplus_{j \in J} A_m^j), \Gamma'$. Then, $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Delta'$ iff there is some $\ell \in J$ such that $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \ell(b))$ and $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (b : A_m^\ell), \Gamma', \Delta'$. Applying the inductive hypothesis at $(b : A_m^\ell), \Gamma'$, we get that

$$\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (b : A_m^\ell), \Gamma', \Delta' \qquad \text{iff} \qquad \eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (b : A_m^\ell), (b : A_m^\ell), \Gamma', \Gamma', \Delta'.$$

Applying the definition of equivalence at $\oplus$ types twice gives the desired result.

If $\Gamma = (a : B_m \multimap C_m), \Gamma'$, then whenever $\eta \; ; \cdot \vdash \mathcal{F}_1|_C, \mathcal{F}_1' \sim \mathcal{F}_2|_C, \mathcal{F}_2' :: (b : B_m)$, it also holds that $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c\rangle) \sim \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c\rangle) :: (c : C_m), \Gamma', \Delta'$. Applying the inductive hypothesis at $(c : C_m), \Gamma'$, we get that this is true if and only if

$$\eta \; ; \cdot \vDash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c\rangle) \sim \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c\rangle) :: (c : C_m), (c : C_m), \Gamma', \Gamma', \Delta'.$$

Again, unfolding the definition of equivalence at $\multimap$ types twice gives the desired result. □

The next two results, on Joining and Splitting, are intuitive, but quite technical in their details. Joining states, at a high level, that given two pairs of equivalent configurations, $\mathcal{C}_1 \sim \mathcal{C}_2$ and $\mathcal{D}_1 \sim \mathcal{D}_2$ we should be able to also say that $\mathcal{C}_1, \mathcal{D}_1$ is equivalent to $\mathcal{C}_2, \mathcal{D}_2$. This is not quite true, of course — one obvious problematic case involves $\mathcal{C}_i$ and $\mathcal{D}_i$ providing the same address, so they cannot even be joined directly. We account for this by allowing some overlap between $\mathcal{C}_i$ and $\mathcal{D}_i$, but then also need to ensure that this overlap admits contraction, so that it can be used by both $\mathcal{C}_i$ and $\mathcal{D}_i$ without cells being illegally read multiple times. Splitting is an inverse to Joining, but similarly has some technical constraints. In particular, while we can join non-final configurations, there are such configurations that cannot be validly split — consider, for instance, a configuration containing a cut, where the left-hand side provides some b, and the right-hand side provides c. There is no way to split this configuration (consisting of only a single process) so that one part provides b and the other provides c, without first evaluating it. The details of equivalence, of course, complicate this slightly, but the core problem remains the same. It is also technically difficult to work with an arbitrary shared portion between the two configurations when splitting (in that it is often unclear which portion should be shared), and so we will require the entire contractible portion of the configuration to be split should be considered shared between the two sides, even if this is not strictly necessary. These two lemmas will be key to much of what follows.

We will often find it useful in these lemmas to refer to the contractible and non-contractible parts of configurations, writing $\mathcal{C} = \mathcal{C}|_C, \mathcal{C}|_{\neg C}$ for this split. Here, $\mathcal{C}|_C$ is the largest subset $\mathcal{D}$ of $\mathcal{C}$ for which $\mathsf{C} \in \sigma(\mathcal{D})$, and $\mathcal{C}|_{\neg C}$ is the remainder of $\mathcal{C}$, capturing the intuitive notion of what pieces of $\mathcal{C}$ may or may not be duplicated.

Lemma 19 (Joining)**.** *Suppose that we have contexts Δ_1, Δ_2, final configurations $\mathcal{F}_i^j$ for $i = 1, 2$ and $j = 1, 2, 3$, and a family η of admissible relations. Suppose also that the following conditions hold:*

(1) $\mathsf{C} \in \sigma(\mathcal{F}_i^2)$.

(2) $\eta \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$.

(3) $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$.

(4) Δ_1, Δ_2 *is well-formed — i.e., the two contexts share no symbols.*

Then, $\eta \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_1, \Delta_2$.

Likewise, the same result holds replacing final configurations $\mathcal{F}_i^j$ everywhere with configurations $\mathcal{C}_i^j$ that need not be final.

Proof. By induction on Δ_1, Δ_2. Note that we prove the results for final configurations and general configurations by mutual induction. The result for general configurations at a particular Δ_1, Δ_2 relies on the result for final configurations at the same context, but the result for final contexts invokes the result for general configurations at a smaller context, and so we are still assured of well-foundedness of the argument. Technically, this is an induction on (Δ_1, Δ_2, j), where $j = 0$ for the claim about final configurations and $j = 1$ for the claim about general configurations.

We first consider general configurations.

Since $\eta \; ; \cdot \vDash \mathcal{C}_1^1, \mathcal{C}_1^2 \sim \mathcal{C}_2^1, \mathcal{C}_2^2 :: \Delta_1$, by definition, $\mathcal{C}_i^1, \mathcal{C}_i^2$ evaluate to final configurations $\mathcal{F}_i^a$ with $\eta \; ; \cdot \vDash \mathcal{F}_1^a \sim \mathcal{F}_2^a :: \Delta_1$. Likewise, $\mathcal{C}_i^2, \mathcal{C}_i^3 \mapsto^* \mathcal{F}_i^b$ with $\eta \; ; \cdot \vDash \mathcal{F}_1^b \sim \mathcal{F}_2^b :: \Delta_2$. Since we reach a final configuration in each case, critically, $\mathcal{C}_i^2$ cannot have read from any cell produced by either $\mathcal{C}_i^1$ or $\mathcal{C}_i^3$ — if it did, on one side or the other we would be left with a thread blocked waiting for a nonexistent cell.[12] We may therefore step $\mathcal{C}_i^2$ to $\mathcal{F}_i^2$ final, independently of the other

[12] This also relies on symbols generated by a cut always being fresh — if $\mathcal{C}_i^1$ and $\mathcal{C}_i^3$ were allowed to generate cells with the same address, and $\mathcal{C}_i^2$ to depend on the cell at this address, this would fail.

$\mathcal{C}_i^j$, and then once that is complete, step the other $\mathcal{C}_i^j$ to $\mathcal{F}_i^j$ final. Moreover, $\mathcal{F}_i^a = \mathcal{F}_i^1, \mathcal{F}_i^2$ and $\mathcal{F}_i^b = \mathcal{F}_i^2, \mathcal{F}_i^3$. Applying the inductive hypothesis (for final configurations) yields the desired result almost immediately.

If both Δ_1 and Δ_2 are empty, then the result is immediate, because equivalence at an empty context is trivial.

Otherwise, Δ_1, Δ_2 contains at least one entry. We assume without loss of generality that this entry is in Δ_1 (as the definition is symmetric).

If $\Delta_1 = (a : t_m), \Delta_1'$ for a type variable t_m, then we can conclude from (2) that we can split $\mathcal{F}_i^1, \mathcal{F}_i^2 = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that

- $\mathcal{G}_i^2 = \mathcal{F}_i^1|_C, \mathcal{F}_i^2$ (the contractible part of $\mathcal{F}_i^1, \mathcal{F}_i^2$)
- $(a, (\mathcal{G}_1^1, \mathcal{G}_1^2))\ \eta(t_m)\ (a, (\mathcal{G}_2^1, \mathcal{G}_2^2))$.
- $\eta\ ;\ \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_1'$.

Note that this means that $\mathcal{G}_i^1, \mathcal{G}_i^3 = \mathcal{F}_i^1|_{\neg C}$. Applying the inductive hypothesis to Δ_1', Δ_2, with the shared portion being $\mathcal{F}_i^2$, we conclude that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_1^1|_C, \mathcal{G}_1^3, \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^1|_C, \mathcal{G}_2^3, \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_1', \Delta_2.$$

As such, we can split $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ into $\mathcal{G}_i^1$, $(\mathcal{F}_i^1|_C, \mathcal{F}_i^2, \mathcal{F}_i^3|_C)$, and $\mathcal{G}_i^3, \mathcal{F}_i^3|_{\neg C}$. Now, using the definition of equivalence at type variables, and noting that $\mathcal{F}_i^1|_C, \mathcal{F}_i^2, \mathcal{F}_i^3|_C = (\mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3)|_C$, it will suffice to show that $(a, (\mathcal{G}_1^1, \mathcal{G}_1^2, \mathcal{F}_1^3|_C))\ \eta(t_m)\ (a, (\mathcal{G}_2^1, \mathcal{G}_2^2, \mathcal{F}_2^3|_C))$, the second condition in this definition. This, however, follows immediately from Lemma 11, as $\mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{F}_i^3|_C$ is necessarily well-formed (otherwise, Δ_1, Δ_2 would not be well-formed).

Suppose $\Delta_1 = (A : B_m \otimes C_m), \Delta_1'$. Now, from (2) and the definition of equivalence at $\otimes$ types, we can conclude that $\mathcal{F}_i^1, \mathcal{F}_i^2$ contains $!_m\mathsf{cell}(a, \langle b, c\rangle)$. We distinguish several cases, depending on whether $\mathsf{C} \in \sigma(m)$ or not.

If $C \notin \sigma(m)$, then we know that the cell at address a must be in $\mathcal{F}_i^1$ for $i = 1, 2$. Write $\mathcal{F}_i^1 = \mathcal{G}_i^1, !_m\mathsf{cell}(a, \langle b, c\rangle)$. We now know that $\eta\ ;\ \cdot \vdash \mathcal{G}_1^1, \mathcal{F}_1^2 \sim \mathcal{G}_2^1, \mathcal{F}_2^2 :: (b : B_m), (c : C_m), \Delta_1'$. Applying the inductive hypothesis to $(b : B_m), (c : C_m), \Delta_1', \Delta_2$, we conclude that

$$\eta\ ;\ \cdot \vdash \mathcal{G}_1^1, \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{G}_2^1, \mathcal{F}_2^2, \mathcal{F}_2^3 :: (b : B_m), (c : C_m), \Delta_1', \Delta_2.$$

We can then apply the definition of equivalence at $\otimes$ types to conclude the desired result.

If $\mathsf{C} \in \sigma(m)$, then we know that $\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (b : B_m), (c : C_m), \Delta_1'$. Applying the inductive hypothesis to $(b : B_m), (c : C_m), \Delta_1', \Delta_2$, we conclude that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathcal{F}_2^3 :: (b : B_m), (c : C_m), \Delta_1', \Delta_2.$$

We can then apply the definition of equivalence at $\otimes$ types to conclude the desired result.

The other cases for $\Delta_1 = (A : A_m^+), \Delta_1'$ where A_m^+ is a positive type are similar.

Suppose $\Delta_1 = (A : B_m \multimap C_m), \Delta_1'$. Let $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ for $i = 1, 2$. We wish to show that $\eta\ ;\ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (A : B_m \multimap C_m), \Delta_1', \Delta_2$. Suppose we have $\mathcal{F}_i'$ such that $\eta\ ;\ \cdot \vdash \mathcal{F}_1|_C, \mathcal{F}_1' \sim \mathcal{F}_2|_C, \mathcal{F}_2' :: (b : B_m)$.

Let $\mathcal{G}_i' = \mathcal{F}_i' \cup (\mathcal{F}_i|_C \cap \mathcal{F}_i^3)$. Then, $\eta\ ;\ \cdot \vdash \mathcal{F}_1^1|_C, \mathcal{F}_1^2|_C, \mathcal{G}_1' \sim \mathcal{F}_2^1|_C, \mathcal{F}_2^2|_C, \mathcal{G}_2' :: (b : B_m)$, since $\mathcal{F}_i|_C = \mathcal{F}_i^1|_C, \mathcal{F}_i^2|_C, \mathcal{F}_i^3|_C$. We therefore know by definition of equivalence at $\multimap$ types that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathcal{G}_1', \mathsf{thread}(c, a.\langle b, c\rangle) \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathcal{G}_2', \mathsf{thread}(c, a.\langle b, c\rangle) :: (c : C_m), \Delta_1'.$$

Applying the inductive hypothesis (for general configurations) to $(c : C_m), \Delta_1', \Delta_2$, taking the shared portion to be $\mathcal{F}_i^2 \cup (\mathcal{F}_i|_C \cap \mathcal{F}_i^3)$, we get that

$$\eta \ ; \cdot \vdash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c\rangle) \sim \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c\rangle) :: (c : C_m), \Delta_1', \Delta_2.$$

The definition of equivalence at $\multimap$ types then gives that $\eta \ ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta_1, \Delta_2$, as desired.

The remaining cases for negative types are similar (if simpler). □

Lemma 20 (Splitting). *If $\eta \ ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta_1, \Delta_2$ then it is possible to write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ for $i = 1, 2$, such that:*
(1) $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
(2) $\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$
(3) $\eta \ ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$

Proof. By induction on Δ_1.

If Δ_1 is empty, then the result is immediate, letting $\mathcal{F}_i^2 = \mathcal{F}_i|_C$ and $\mathcal{F}_i^3$ the remainder of $\mathcal{F}_i$.

If $\Delta_1 = (a : t_m), \Delta_1'$ for some type variable t_m, then, by definition, $\mathcal{F}_i = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that

- $\mathcal{G}_i^2 = \mathcal{G}_i|_C$
- $(a, (\mathcal{G}_1^1, \mathcal{G}_1^2)) \ \eta(t_m) \ (a, (\mathcal{G}_2^1, \mathcal{G}_2^2))$
- $\eta \ ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_1', \Delta_2$

Now, applying the inductive hypothesis to Δ_1', Δ_2, we get a split of $\mathcal{G}_i^2, \mathcal{G}_i^3 = \mathcal{G}_i^4, \mathcal{G}_i^5, \mathcal{G}_i^6$. Since $\mathcal{G}_i^2 = \mathcal{F}_i|_C$, we also know that $\mathcal{G}_i^5 = \mathcal{G}_i^2$, and so this yields also a split of $\mathcal{G}_i^3$ into $\mathcal{G}_i^4, \mathcal{G}_i^6$, satisfying $\eta \ ; \cdot \vdash \mathcal{G}_1^4, \mathcal{G}_1^2 \sim \mathcal{G}_2^4, \mathcal{G}_2^2 :: \Delta_1'$ and $\eta \ ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^6 \sim \mathcal{G}_2^2, \mathcal{G}_2^6 :: \Delta_2$.

Taking $\mathcal{F}_i^1 = \mathcal{G}_i^1, \mathcal{G}_i^4$, $\mathcal{F}_i^2 = \mathcal{G}_i^2$, and $\mathcal{F}_i^3 = \mathcal{G}_i^6$, we can easily see that the desired result holds — to show that $\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (a : t_m), \Delta_1'$, we split this as $\mathcal{G}_i^1, \mathcal{F}_i^2, \mathcal{G}_i^4$.

Positive Cases Suppose $\Delta_1 = (a : \oplus\{\ell : A_m^\ell\}_{\ell \in L}), \Delta_1'$. Then, by definition, there is $j \in L$ such that $\mathcal{F}_i = \mathcal{G}_i', !_m\mathsf{cell}(a, j(b))$. We consider two cases, depending on whether $\mathsf{C} \in \sigma(m)$ or not.

If $\mathsf{C} \in \sigma(m)$, then we also know by definition that $\eta \ ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (b : A_m^j), \Delta_1', \Delta_2$. Applying the inductive hypothesis, we get a split of $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that

(1) $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
(2) $\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (b : A_m^j), \Delta_1'$
(3) $\eta \ ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$

It now will suffice to show that $\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$ as well, for which it suffices to show that $!_m\mathsf{cell}(a, j(b))$ occurs in $\mathcal{F}_i^1$ or $\mathcal{F}_i^2$ for $i = 1, 2$. This is immediate, as $\mathsf{C} \in \sigma(m)$, so necessarily, this cell occurs in $\mathcal{F}_i^2$.

If $C \notin \sigma(m)$, then we also know that $\eta \ ; \cdot \vdash \mathcal{G}_1 \sim \mathcal{G}_2 :: (b : A_m^j), \Delta_1', \Delta_2$. Again, we apply the inductive hypothesis to get a split of $\mathcal{G}_i = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$. Now, we take $\mathcal{F}_i^1 = \mathcal{G}_i^1, !_m\mathsf{cell}(a, j(b))$, and $\mathcal{F}_i^j = \mathcal{G}_i^j$ for $j = 2, 3$. Since $C \notin \sigma(m)$, and $\mathcal{F}_i = \mathcal{G}_i, !_m\mathsf{cell}(a, j(b))$, we can conclude that $\mathcal{G}_i|_C = \mathcal{G}_i^2 = \mathcal{F}_i^2 = \mathcal{F}_i|_C$. We can then also easily conclude that $\eta \ ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$. It remains to show that $\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$. This is almost immediate by definition —

we know that $\mathcal{F}_i^1, \mathcal{F}_i^2 = \mathcal{G}_i^1, \mathcal{G}_i^2, !_m\mathsf{cell}(a, j(b))$, and the necessary equivalence at $(b : A_m^j)$ comes from the inductive hypothesis.

The other cases for $\Delta_1 = (a : A_m^+), \Delta_1'$ are similar.

Negative Cases Suppose $\Delta_1 = (a : \&\{\ell : A_m^\ell\}_{\ell \in L}), \Delta_1'$. Then, by definition, for any choice of $j \in L$ and fresh symbol b, we have that

$$\eta \, ; \cdot \vdash \mathcal{F}_1, \mathsf{thread}(b, a.j(b)) \sim \mathcal{F}_2, \mathsf{thread}(b, a.j(b)) :: (b : A_m^j), \Delta_1', \Delta_2.$$

This means that $\mathcal{F}_i, \mathsf{thread}(b, a.j(b)) \mapsto^* \mathcal{G}_i$ final with $\eta \, ; \cdot \vdash \mathcal{G}_1 \sim \mathcal{G}_2 :: (b : A_m^j), \Delta_1', \Delta_2$, and we may therefore apply the inductive hypothesis to split $\mathcal{G}_i = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that

(1) $\mathcal{G}_i^2 = \mathcal{G}_i|_C$
(2) $\eta \, ; \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_1^2 \sim \mathcal{G}_2^1, \mathcal{G}_2^2 :: (b : A_m^j), \Delta_1'$
(3) $\eta \, ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_2$

Now, we note that $\mathcal{F}_i$ and $\mathcal{G}_i$ consist of three types of cells:

- Cells that occur both in $\mathcal{F}_i$ and $\mathcal{G}_i$
- Cells that occurred in $\mathcal{F}_i$, but were consumed by the running process that yields $\mathcal{G}_i$. Note that these cells necessarily are not contractible, and that they do not appear in any $\mathcal{G}_i^j$ (because they do not appear in $\mathcal{G}_i$)
- Cells newly created for $\mathcal{G}_i$ by the process. These cells do not occur in $\mathcal{F}_i$, and may occur in any of the $\mathcal{G}_i^j$. However, since b is chosen fresh, and any new cells created by the process will also be at fresh addresses, Δ_2 cannot depend on them, so we may assume without loss of generality that these cells occur only in $\mathcal{G}_i^1$ and $\mathcal{G}_i^2$.

Choose $\mathcal{F}_i^3 = \mathcal{F}_i \cap \mathcal{G}_i^3$[13] and $\mathcal{F}_i^2 = \mathcal{F}_i|_C$. We know that $\eta \, ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_2$, and $\mathcal{F}_i^2, \mathcal{F}_i^3$ is nearly the same as $\mathcal{G}_i^2, \mathcal{G}_i^3$, differing only in that $\mathcal{G}_i^2$ may be larger than $\mathcal{F}_i^2$. However, since Δ_2 cannot depend on any of these new cells, we may also conclude that

$$\eta \, ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2.$$

It now only remains to define $\mathcal{F}_i^1 = (\mathcal{F}_i \cap \mathcal{G}_i^1) \cup (\mathcal{F}_i \setminus \mathcal{G}_i)$.[14] The first term consists of the portion of $\mathcal{G}_i^1$ that already occurred in $\mathcal{F}_i$, while the second term is the portion of $\mathcal{F}_i$ that was consumed to produce the remainder of $\mathcal{G}_i^1$ (and possibly some of $\mathcal{G}_i^2$). We now claim that $\eta \, ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathsf{thread}(b, a.j(b)) \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathsf{thread}(b, a.j(b)) :: (b : A_m^j), \Delta_1'$. This follows from closure of logical equivalence under converse reduction (Lemma 15) if we can show that $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(b, a.j(b)) \mapsto^* \mathcal{G}_i^1, \mathcal{G}_i^2$. This is almost immediate — we need only show that the running process can never read from any cells in $\mathcal{F}_i^3$, but if it were to read from such a cell (which, not being in $\mathcal{G}_i^2$, is necessarily not contractible), this would consume the cell, preventing it from occurring in $\mathcal{G}_i^3$, contradicting that it was in $\mathcal{F}_i^3$ to begin with. As such, the process only reads from cells that either occur in $\mathcal{F}_i^1$ or $\mathcal{F}_i^2$, or cells that

[13] Based on our assumption above, this is, without loss of generality, just $\mathcal{G}_i^3$

[14] This can be written more simply as $\mathcal{F}_i \setminus \mathcal{G}_i^2 \setminus \mathcal{G}_i^3$, but this obscures the meaning of the two different pieces of $\mathcal{F}_i^1$.

it newly creates, and so indeed $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(b, a.j(b))$ reaches a final state, containing the cells that occur both in $\mathcal{F}_i^1, \mathcal{F}_i^2$ and in $\mathcal{G}_i$, as well as the cells that were newly created by the process (and not also consumed). This gives precisely $\mathcal{G}_i^1, \mathcal{G}_i^2$ — $\mathcal{G}_i^1$ differs from $\mathcal{F}_i^1$ only in the addition of some new cells created by the running process, and likewise for $\mathcal{G}_i^2$. Since $\eta ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathsf{thread}(b, a.j(b)) \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathsf{thread}(b, a.j(b)) :: (b : A_m^j), \Delta_1'$ holds, we also get, by definition, that $\eta ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (a : \&\{\ell : A_m^\ell\}_{\ell \in L}), \Delta_1'$, concluding the desired result.

Suppose $\Delta_1 = (a : B_m \multimap C_m), \Delta_1'$. Then, by definition, whenever we have $\mathcal{F}_1', \mathcal{F}_2'$ such that $\eta ; \cdot \vdash \mathcal{F}_1|_C, \mathcal{F}_1' \sim \mathcal{F}_2|_C, \mathcal{F}_2' :: (b : B_m)$, it also holds that

$$\eta ; \cdot \vdash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c\rangle) \sim \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c\rangle) :: (c : C_m), \Delta_1', \Delta_2.$$

This means that $\mathcal{F}_i, \mathcal{F}_i', \mathsf{thread}(b, a.j(b)) \mapsto^* \mathcal{G}_i$ final with $\eta ; \cdot \vdash \mathcal{G}_1 \sim \mathcal{G}_2 :: (c : C_m), \Delta_1', \Delta_2$, and we may therefore apply the inductive hypothesis to split $\mathcal{G}_i = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that

(1) $\mathcal{G}_i^2 = \mathcal{G}_i|_C$
(2) $\eta ; \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_1^2 \sim \mathcal{G}_2^1, \mathcal{G}_2^2 :: (c : C_m), \Delta_1'$
(3) $\eta ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_2$

As in the case of $\&$, we observe that there are several types of cells in $\mathcal{G}_i$ and $\mathcal{F}_i$, the only difference being that some cells in $\mathcal{G}_i$ may have come from $\mathcal{F}_i'$. We note that the cells in $\mathcal{F}_i'$ can, without loss of generality, be placed in $\mathcal{G}_i^1, \mathcal{G}_i^2$, for much the same reason that any cells generated by the running process can be placed in this part of the configuration.

Now, take $\mathcal{F}_i^2 = \mathcal{F}_i|_C$ and $\mathcal{F}_i^3 = \mathcal{F}_i \cap \mathcal{G}_i^3$ (or just $\mathcal{F}_i^3 = \mathcal{G}_i^3$). As in the case for $\&$ types, we can see that any portion of $\mathcal{G}_i^2$ not in $\mathcal{F}_i^2$ is unnecessary for Δ_2, and so we still have that $\eta ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$.

Finally, take $\mathcal{F}_i^1 = (\mathcal{F}_i \cap \mathcal{G}_i^1) \cup (\mathcal{F}_i \setminus \mathcal{G}_i)$. Again, as in the case of $\&$ types, the first portion of this consists of the part of $\mathcal{G}_i^1$ that already occurred in $\mathcal{F}_i$ (the remainder comes from $\mathcal{F}_i'$ and new cells generated by the running process), and the second portion consists of cells that are consumed by the running process while generating $\mathcal{G}_i^1$. We now note that

$$\mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i', \mathsf{thread}(c, a.\langle b, c\rangle) \mapsto^* \mathcal{G}_i^1, \mathcal{G}_i^2,$$

and closure of logical equivalence under converse reduction (Lemma 15) gives us the desired result, using that $\mathcal{F}_i'$ were arbitrarily chosen and the definition of equivalence at $\multimap$ types.

The remaining cases for negative types are similar. □

While the next two lemmas on type extension and reuse of contractible data are more similar in spirit to the earlier results on weakening and contraction, specifying how we can modify the context at which configurations are equivalent, they rely on joining and splitting.

Lemma 21 (Type Extension). *If $\Xi ; \Gamma \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$, and Θ is a context which shares no symbols with Γ, Δ, $\mathcal{C}_1$, or $\mathcal{C}_2$, then also $\Xi ; \Gamma, \Theta \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta, \Theta$.*

Proof. Suppose $\Xi ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Theta$.

By Splitting (Lemma 20), we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathcal{F}_i^2 = \mathcal{F}_i|_C$

2. $\Xi \,;\, \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Gamma$
3. $\Xi \,;\, \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Theta$

Now, by assumption, this means that $\Xi \,;\, \cdot \vDash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathcal{C}_1 \sim \mathcal{F}_2^1, \mathcal{F}_2^2, \mathcal{C}_2 :: \Delta$. Joining (Lemma 19) with the shared portion being $\mathcal{F}_i^2$, and using that Δ and Θ share no symbols, then gives us that $\Xi \,;\, \cdot \vDash \mathcal{F}_1, \mathcal{C}_1 \sim \mathcal{F}_2, \mathcal{C}_2 :: \Delta, \Theta$. As such, $\Xi \,;\, \Gamma, \Theta \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta, \Theta$. □

Lemma 22 (Reuse)**.** *Suppose* $\Xi \,;\, \Gamma_1, \Gamma_2 \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$. *Suppose also that* $\mathsf{C} \in \sigma(\Gamma_2)$ *and that* Δ, Γ_2 *is well-formed. Then,* $\Xi \,;\, \Gamma_1, \Gamma_2 \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta, \Gamma_2$.

Proof. Suppose $\Xi \,;\, \cdot \vDash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma_1, \Gamma_2$. By splitting, we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
2. $\Xi \,;\, \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Gamma_1$
3. $\Xi \,;\, \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma_2$.

Since $\mathsf{C} \in \sigma(\Gamma_2)$, we note that $\mathcal{F}_i^2, \mathcal{F}_i^3$ must be entirely contractible, and so, in particular, $\mathcal{F}_i^3$ must be empty.

By definition and our initial assumption, $\Xi \,;\, \cdot \vDash \mathcal{F}_1, \mathcal{C}_1 \sim \mathcal{F}_2, \mathcal{C}_2 :: \Delta$. Now, applying joining with the shared portion being $\mathcal{F}_i^2$, we get that $\Xi \,;\, \cdot \vDash \mathcal{F}_1, \mathcal{C}_1 \sim \mathcal{F}_2, \mathcal{C}_2 :: \Delta, \Gamma_2$, as desired. □

The next three results are used to handle type variables, and in particular, to show that logical equivalence itself is a sensible basis for defining equality at a type variable.

Lemma 23 (Compositionality for Closed Configurations)**.** *Suppose that for some* A_m*,* R *is defined by*

$$(a, \mathcal{C}_1)\ R\ (a, \mathcal{C}_2) \qquad \textit{if and only if} \qquad \eta \,;\, \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: (a : A_m).$$

Then,

$$\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta[A_m/t_m] \qquad \textit{if and only if} \qquad \eta, t_m \hookrightarrow R \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta,$$

and similarly,

$$\eta \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta[A_m/t_m] \qquad \textit{if and only if} \qquad \eta, t_m \hookrightarrow R \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta.$$

Proof. We first consider the case of general (not necessarily final) configurations $\mathcal{C}$. By definition, $\eta \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta[A_m/t_m]$ if and only if there are $\mathcal{F}_1, \mathcal{F}_2$ final such that $\mathcal{C}_i \mapsto^* \mathcal{F}_i$ and $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta[A_m/t_m]$. Similarly, $\eta, t_m \hookrightarrow R \,;\, \cdot \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ if and only if there are $\mathcal{F}'_1, \mathcal{F}'_2$ final such that $\mathcal{C}_i \mapsto^* \mathcal{F}'_i$ and $\eta, t_m \hookrightarrow R \,;\, \cdot \vdash \mathcal{F}'_1 \sim \mathcal{F}'_2 :: \Delta$. Confluence means that if $\mathcal{C}_i \mapsto^* \mathcal{F}_i$ and $\mathcal{C}_i \mapsto^* \mathcal{F}'_i$ final, then $\mathcal{F}_i$ and $\mathcal{F}'_i$ are equal up to renaming, and so the result for general configurations $\mathcal{C}$ reduces to proving the result for final configurations $\mathcal{F}$.

For final configurations $\mathcal{F}$, we proceed by induction on Δ.

If $\Delta = (\cdot)$, then the result is immediate, as we have no proof obligation.

Suppose $\Delta = (a : t_m), \Delta'$ and that $\eta, t_m \hookrightarrow R \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : t_m), \Delta'$. By definition, then, we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that

- $\mathcal{F}_i^2 = \mathcal{F}_i|_C$ for $i = 1, 2$

- $(a, (\mathcal{F}_1^1, \mathcal{F}_1^2))\ R\ (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$, or $\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (a : A_m)$.
- $\eta, t_m \hookrightarrow R\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$.

Applying the induction hypothesis to Δ', we can conclude that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'[A_m/t_m].$$

Then, by Joining (Lemma 19), we get that $\eta\ ;\ \cdot \vdash \mathcal{F}_1 :: (a : A_m), \Delta'[A_m/t_m]$, which is exactly the desired result.

Now, suppose that $\Delta = (a : t_m), \Delta'$ and that $\eta\ ;\ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m), \Delta'[A_m/t_m]$. By lemma 20, we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that

- $\mathcal{F}_i^2 = \mathcal{F}_i|_C$ for $i = 1, 2$
- $\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (a : A_m)$, or, equivalently, $(a, (\mathcal{F}_1^1, \mathcal{F}_1^2))\ R\ (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$.
- $\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'[A_m/t_m]$.

Applying the inductive hypothesis to Δ', we get that $\eta, t_m \hookrightarrow R\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$. The result then follows immediately from the definition for equivalence at type variables.

In all remaining cases, we can just recurse, applying the inductive hypothesis as necessary. □

Lemma 24 (Compositionality for Open Configurations)**.** *Suppose that for some A_m, R is defined by $(a, \mathcal{C}_1)\ R\ (a, \mathcal{C}_2)$ if and only if $\eta\ ;\ \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: (a : A_m)$.*

Then, $\eta\ ;\ \Gamma[A_m/t_m] \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta[A_m/t_m]$ if and only if $\eta, t_m \hookrightarrow R\ ;\ \Gamma \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$.

Proof. Suppose $\eta\ ;\ \cdot \vDash \mathcal{C}_1' \sim \mathcal{C}_2' :: \Gamma[A_m/t_m]$. By Lemma 23, this is true if and only if $\eta, t_m \hookrightarrow R\ ;\ \cdot \vdash \mathcal{C}_1' \sim \mathcal{C}_2' :: \Gamma$.

Now, again by Lemma 23, we have that $\eta\ ;\ \cdot \vDash \mathcal{C}_1, \mathcal{C}_1' \sim \mathcal{C}_2, \mathcal{C}_2' :: \Delta[A_m/t_m]$ if and only if $\eta, t_m \hookrightarrow R\ ;\ \cdot \vDash \mathcal{C}_1, \mathcal{C}_1' \sim \mathcal{C}_2, \mathcal{C}_2' :: \Delta$. Since $\mathcal{C}_1', \mathcal{C}_2'$ were arbitrary equivalent configurations, this gives exactly the desired result. □

Lemma 25 (Extension of Type Variable Context)**.** *Suppose that t_m does not occur free in Δ, and that $R : A_m \leftrightarrow B_m$. Then, $\eta\ ;\ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$ if and only if $\eta, t_m \hookrightarrow R\ ;\ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$.*

Proof. Since t_m does not occur free in Δ, and R is only used in cases where t_m occurs free in Δ, R is never used in such a proof of equivalence, and so can be freely removed (or added) without affecting the correctness of such a proof. □

Parametricity

We now set out to prove parametricity. Since we are given a well-typed configuration, we know that there is a typing derivation for it. By providing a logical equivalence version of each typing rule, we can then build a derivation of equivalence with the same structure as the typing derivation, and so we begin by proving these variants of each typing rule.

For the positive left rules, where we have several different cases to distinguish, depending on the value of α and whether the principal formula admits contraction or not, the following generic lemma will be useful. Unfortunately, existentials behave differently enough from the other positive types[15] that we cannot include them:

Lemma 26. *Let A_m^+ be a positive type which is not an existential type, and $\eta : \Xi$. Suppose $\eta ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : A_m^+)$.*

Inversion tells us that $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, V)$ and that there is Δ_V (dependent on A_m^+) such that $\Xi ; \Delta_V \vDash !_m\mathsf{cell}(a, V) :: (a : A_m^+)$. For instance, if $A_m^+ = \oplus_{j \in J} A_m^j$, then V has the form $\ell(b)$ and Δ_V is $b : A_m^\ell$.

Then also $\eta ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Gamma, (a : A_m^+)^\alpha, \Delta_V$, where α may only be 1 if $\mathsf{C} \in \sigma(m)$.

Proof. By definition of logical equivalence (at positive types), we get that $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, V)$ and that $\eta ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Gamma, \Delta_V$.

If $\alpha = 0$, we note that we already have $\eta ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Gamma, \Delta_V$, which is exactly the desired result.

If $\alpha = 1$, then we necessarily are in the case where $\mathsf{C} \in \sigma(m)$. Applying the third part of contraction (Lemma 18) to our initial hypothesis, we also get that

$$\eta ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : A_m^+), (a : A_m^+).$$

By the definition of equivalence at A_m^+, this means that (among other things)

$$\eta ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : A_m^+), \Delta_V,$$

which is exactly the desired result when $\alpha = 1$. □

We now go on to prove a sample of these logical equivalence versions of our typing rules, beginning with the judgmental rules of cut and identity, and then showing some example positive and negative cases.

Lemma 27 (cut)**.** *Suppose $\Gamma, \Delta \geq m \geq r$, and that $\mathsf{C} \in \sigma(\Gamma)$.*

If

$$\Xi ; \Gamma, \Delta \vDash \mathsf{thread}(a, P_1) \sim \mathsf{thread}(a, P_2) :: (a : A_m)$$

and

$$\Xi ; \Gamma, \Delta', a : A_m \vDash \mathsf{thread}(c, Q_1) \sim \mathsf{thread}(c, Q_2) :: (c : C_r)$$

then it also holds that

$$\Xi ; \Gamma, \Delta, \Delta' \vDash \mathsf{thread}(c, x \leftarrow P_1[x/a] ; Q_1[x/a]) \sim \mathsf{thread}(c, x \leftarrow P_2[x/a] ; Q_2[x/a]) :: (c : C_r)$$

Proof. Fix some $\xi_1, \xi_2 : \Xi$ and $\eta : \xi_1 \leftrightarrow \xi_2$ (we will often write $\Xi : \eta$ for this, avoiding the need to handle ξ_1 and ξ_2 explicitly where they are not needed). By type extension (Lemma 21) with Δ' applied to the first hypothesis, we get that

$$\eta ; \Gamma, \Delta, \Delta' \vDash \mathsf{thread}(a, P_1) \sim \mathsf{thread}(a, P_2) :: \Delta', (a : A_m)$$

[15] Our existentials are positive, in that their left rule is invertible, but the need to introduce a new type variable/admissible relation to work with them makes them not fit into the context of this lemma

Now, applying reuse (Lemma 22) with Γ, we have

$$\eta \,;\, \Gamma, \Delta, \Delta' \vDash \mathsf{thread}(a, P_1) \sim \mathsf{thread}(a, P_2) :: \Gamma, \Delta', (a : A_m)$$

Let $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, \Delta, \Delta'$. Applying the definition of equivalence for open configurations, we get that

$$\eta \,;\, \cdot \vdash \mathcal{F}_1, \mathsf{thread}(a, P_1) \sim \mathcal{F}_2, \mathsf{thread}(a, P_2) :: \Gamma, \Delta', (a : A_m).$$

As such, $\mathcal{F}_i, \mathsf{thread}(a, P_i) \mapsto^* \mathcal{F}_i'$ for some final $\mathcal{F}_i'$ with $\eta \,;\, \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Gamma, \Delta, (a : A_m)$.

Again applying the definition of equivalence for open configurations, this time to our second hypothesis, we get

$$\eta \,;\, \cdot \vDash \mathcal{F}_1', \mathsf{thread}(c, Q_1) \sim \mathcal{F}_2', \mathsf{thread}(c, Q_2) :: (c : C_r).$$

Now, we note that $\mathcal{F}_i, \mathsf{thread}(c, x \leftarrow P_i[x/a] \,;\, Q_i[x/a]) \mapsto \mathcal{F}_i, \mathsf{thread}(a, P_1), \mathsf{thread}(c, Q_i)$, and that $\mathcal{F}_i, \mathsf{thread}(a, P_i), \mathsf{thread}(c, Q_i) \mapsto^* \mathcal{F}_i', \mathsf{thread}(c, Q_i)$, and so closure of logical equivalence under converse reduction gives the desired result. □

Lemma 28 (id). *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then,

$$\Xi \,;\, \Gamma, a : A_m \vDash \mathsf{thread}(c, c \leftarrow a) \sim \mathsf{thread}(c, c \leftarrow a) :: (c : A_m)$$

Proof. Fix some $\eta : \Xi$, and let $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, a : A_m$. By Inversion (Lemma 12), we can write $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, D_i)$ for some cell data D_i. As such, we may conclude that $\mathcal{F}_i, \mathsf{thread}(c, c \leftarrow a) \mapsto C_m?(\mathcal{F}_i, \mathcal{F}_i'), !_m\mathsf{cell}(c, D_i)$.

If $\mathsf{C} \in \sigma(m)$, then the first case of contraction (Lemma 18) gives us that

$$\eta \,;\, \cdot \vdash \mathcal{F}_1, !_m\mathsf{cell}(c, D_1) \sim \mathcal{F}_2, !_m\mathsf{cell}(c, D_2) :: (a : A_m), (c : A_m).$$

After weakening (Lemma 17) to remove $(a : A_m)$, closure under converse reduction gives the result.

If $C \notin \sigma(m)$, then the second case of contraction gives us that

$$\eta \,;\, \cdot \vdash \mathcal{F}_1', \mathsf{cell}(c, D_1) \sim \mathcal{F}_2', \mathsf{cell}(c, D_2) :: (c : A_m),$$

as desired. □

Lemma 29 ($\downarrow L_\alpha$). *Suppose m, k are modes with $k \leq m$, and that $\alpha \in \{0, 1\}$, and that if $\alpha = 1$ then $\mathsf{C} \in \sigma(k)$. If*

$$\Xi \,;\, \Gamma, (b : {\downarrow}_k^m A_m)^\alpha, a : A_m \vDash \mathsf{thread}(c, Q_1) \sim \mathsf{thread}(c, Q_2) :: (c : C_r)$$

then also

$$\Xi \,;\, \Gamma, b : {\downarrow}_k^m A_m \vDash \mathsf{thread}(c, \mathsf{case}\ b\ (y \Rightarrow Q_1[y/a])) \sim \mathsf{thread}(c, \mathsf{case}\ b\ (y \Rightarrow Q_2[y/a])) :: (c : C_r)$$

Proof. Fix $\eta : \Xi$, and suppose that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, b : \downarrow_k^m A_m$.

By definition of equivalence at $\downarrow_k^m A_m$, we can write $\mathcal{F}_i = \mathcal{F}_i', !_k\mathsf{cell}(b, \mathsf{shift}(a))$, and then, applying Lemma 26, we get that

$$\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Gamma, (b : \downarrow_k^m A_m)^\alpha, (a : A_m)$$

Applying the definition of equivalence for open configurations, we see that

$$\eta \; ; \cdot \vDash C_m?(\mathcal{F}_1, \mathcal{F}_1'), \mathsf{thread}(c, Q_1) \sim C_m?(\mathcal{F}_2, \mathcal{F}_2'), \mathsf{thread}(c, Q_2) :: (c : C_r)$$

Now, observe that $\mathcal{F}_i, \mathsf{thread}(c, \mathsf{case}\ b\ (y \Rightarrow Q_i[y/a])) \mapsto C_m?(\mathcal{F}_i, \mathcal{F}_i'), \mathsf{thread}(c, Q_i)$, and so the result follows from closure under converse reduction. □

Lemma 30 ($\downarrow R^0$)**.** *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then,

$$\Xi \; ; \Gamma, (a : A_m) \vDash \mathsf{thread}(c_k.\mathsf{shift}(a_m)) \sim \mathsf{thread}(c_k.\mathsf{shift}(a_m)) :: (c : \downarrow_k^m A_m).$$

Proof. Fix $\eta : \Xi$, and let $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : A_m)$.

Observe that $\mathcal{F}_i, \mathsf{thread}(c_k.\mathsf{shift}(a_m)) \mapsto \mathcal{F}_i, !_k\mathsf{cell}(c_k, \mathsf{shift}(a_m))$, and so by closure under converse reduction, it will suffice to show that

$$\eta \; ; \cdot \vdash \mathcal{F}_1, !_k\mathsf{cell}(c_k, \mathsf{shift}(a_m)) \sim \mathcal{F}_2, !_k\mathsf{cell}(c_k, \mathsf{shift}(a_m)) :: (c : \downarrow_k^m A_m).$$

We already have that $\mathcal{F}_i, !_k\mathsf{cell}(c_k, \mathsf{shift}(a_m))$ contains a suitable cell, and so it only remains to show that

$$\eta \; ; \cdot \vdash \mathcal{F}_1, C_k?(!_k\mathsf{cell}(c_k, \mathsf{shift}(a_m)), \cdot) \sim \mathcal{F}_2, C_k?(!_k\mathsf{cell}(c_k, \mathsf{shift}(a_m)), \cdot) :: (a : A_m).$$

By weakening (Lemma 17), we get that

$$\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: (a : A_m),$$

and we can then (in the case where $\mathsf{C} \in \sigma(k)$) apply Lemma 16 to get the desired result. □

Lemma 31 ($\multimap L^0$)**.** *Suppose* $\mathsf{W} \in \sigma(\Gamma)$. *Then,*

$$\Xi \; ; \Gamma, w : A_m, x : A_m \multimap B_m \vDash \mathsf{thread}(y, x.\langle w, y\rangle) \sim \mathsf{thread}(y, x.\langle w, y\rangle) :: (y : B_m)$$

Proof. Fix $\eta : \Xi$ and suppose that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, w : A_m, x : A_m \multimap B_m$. Applying Splitting (Lemma 20), we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
2. $\eta \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 \sim \mathcal{F}_2^1, \mathcal{F}_2^2 :: (w : A_m)$
3. $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma, (x : A_m \multimap B_m)$

Now, note that $(\mathcal{F}_i^2, \mathcal{F}_i^3)|_C = \mathcal{F}_i^2$, so, taking $\mathcal{F}_i'$ in the definition of equivalence at $\multimap$ types to be $\mathcal{F}_i^1$, we can conclude that

$$\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, x.\langle w, y\rangle) \sim \mathcal{F}_2, \mathsf{thread}(y, x.\langle w, y\rangle) :: (y : B_m), \Gamma.$$

Weakening away Γ then gives the desired result. □

Lemma 32 ($\multimap R$)**.** *Suppose* $\Xi \,;\, \Gamma, w : A_m \vdash \mathsf{thread}(y, P_1) \sim \mathsf{thread}(y, P_2) :: (y : B_m)$*. Then,*

$\Xi \,;\, \Gamma \vdash \mathsf{thread}(x, \mathsf{case}\ x\ (\langle w, y\rangle \Rightarrow P_1)) \sim \mathsf{thread}(x, \mathsf{case}\ x\ (\langle w, y\rangle \Rightarrow P_2)) :: (x : A_m \multimap B_m)$

Proof. Fix $\eta : \Xi$, and suppose that $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma$.

Now, let $\eta \,;\, \cdot \vdash \mathcal{F}'_1 \sim \mathcal{F}'_2 :: (w : A_m)$. By Joining (Lemma 19),

$$\eta \,;\, \cdot \vdash \mathcal{F}_1, \mathcal{F}'_1 \sim \mathcal{F}_2, \mathcal{F}'_2 :: \Gamma, w : A_m,$$

and so by assumption,

$$\eta \,;\, \cdot \vdash \mathcal{F}_1, \mathcal{F}'_1, \mathsf{thread}(y, P_1) \sim \mathcal{F}_2, \mathcal{F}'_2, \mathsf{thread}(y, P_2) :: (y : B_m).$$

We then note that $\mathcal{C}_i, \mathcal{F}'_i, \mathsf{thread}(x, \mathsf{case}\ x\ (\langle w, y\rangle \Rightarrow P_i)) \mapsto^* \mathcal{F}_i, \mathcal{F}'_i, \mathsf{thread}(y, P_i)$, and so the result follows from (repeated) closure under converse reduction (Lemma 23). □

Lemma 33 ($\exists L_\alpha$)**.** *Suppose* $\Xi \vdash C_r$ *and*

$$\Xi, t_m \,;\, \Gamma, (a : \exists t_m.A_m)^\alpha, b : A_m \vdash \mathsf{thread}(c, Q_1) \sim \mathsf{thread}(c, Q_2) :: (c : C_r),$$

with $\alpha = 1$ *only if* $\mathsf{C} \in \sigma(m)$*.*

Then also

$$\begin{aligned}\Xi \,;\, \Gamma, a : \exists t_m.A_m \vdash\ & \mathsf{thread}(c, \mathsf{case}\ x\ (\langle t_m, y\rangle \Rightarrow Q_1[y/b])) \sim \\ & \mathsf{thread}(c, \mathsf{case}\ x\ (\langle t_m, y\rangle \Rightarrow Q_2[y/b])) :: (c : C_r).\end{aligned}$$

Proof. Fix $\eta : \Xi$, and suppose that $\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, a : \exists t_m.A_m$. By definition, then, there is some $R : B^1_m \leftrightarrow B^2_m$ such that $\mathcal{F}_i = \mathcal{F}'_i, !_m\mathsf{cell}(a, \langle B^i_m, b\rangle)$, and

$$\eta, t_m \hookrightarrow R \,;\, \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}'_1) \sim C_m?(\mathcal{F}_2, \mathcal{F}'_2) :: \Gamma, (b : A_m).$$

If $\alpha = 0$, using the definition of logical equivalence for open configurations and our first hypothesis, we get that

$$\eta, t_m \hookrightarrow R \,;\, \cdot \vDash C_m?(\mathcal{F}_1, \mathcal{F}'_1), \mathsf{thread}(c, Q_1) \sim C_m?(\mathcal{F}_2, \mathcal{F}'_2), \mathsf{thread}(c, Q_2) :: (c : C_r)$$

If $\alpha = 1$, then necessarily $\mathsf{C} \in \sigma(m)$, and so by applying the third part of contraction (Lemma 18) to our initial supposition about the $\mathcal{F}_i$, we get that

$$\eta \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : \exists t_m.A_m), (a : \exists t_m.A_m)$$

Applying the definition of equivalence at $\exists t_m.A_m$, we get that there is some $R' : C^1_m \leftrightarrow C^2_m$ such that

$$\eta, t_m \hookrightarrow R' \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (a : \exists t_m.A_m), (b : A_m).$$

Again, we can combine this with our first hypothesis to get that

$$\eta, t_m \hookrightarrow R' \,;\, \cdot \vDash \mathcal{F}_1, \mathsf{thread}(c, Q_1) \sim \mathcal{F}_2, \mathsf{thread}(c, Q_2) :: (c : C_r)$$

In either case, $\mathcal{F}_i, \mathsf{thread}(c, \mathsf{case}\ x\ (\langle t_m, y\rangle \Rightarrow Q_i[y/b])) \mapsto C_m?(\mathcal{F}_i, \mathcal{F}'_i), \mathsf{thread}(c, Q_i)$, and so by closure under converse reduction, we have the result. □

Lemma 34 ($\exists R^0$). *Suppose* $\Xi \vdash B_m$ *and* $\Xi, t_m \vdash A_m$*, and that* $\mathsf{W} \in \sigma(\Gamma)$*. Then,*

$$\Xi \; ; \Gamma, a : A_m[B_m/t_m] \vdash \mathsf{thread}(c, c.\langle B_m, a\rangle) \sim \mathsf{thread}(c, c.\langle B_m, a\rangle) :: (c : \exists t_m.A_m)$$

Proof. Fix $\eta : \Xi$ and let $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, a : A_m[B_m/t_m]$. By compositionality for closed configurations (Lemma 23), if R is defined by $(a, \mathcal{C}_1) \; R \; (a, \mathcal{C}_2)$ when $\eta \; ; \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: (a : B_m)$, we get that

$$\eta, t_m \hookrightarrow R \; ; \cdot \vdash \mathcal{F}_1, \sim \mathcal{F}_2 :: \Gamma, a : A_m$$

as well.

Observe that $\mathcal{F}_i, \mathsf{thread}(c, c.\langle B_m, a\rangle) \mapsto \mathcal{F}_i, !_m\mathsf{cell}(c, \langle B_m, a\rangle)$, which we may therefore consider instead (by closure under converse reduction). By the definition of equivalence at existential types, we need to find $R : B_m \leftrightarrow B_m$ such that

$$\eta, t_m \hookrightarrow R \; ; \cdot \vdash \mathcal{F}_1, C_m?(!_m\mathsf{cell}(c, \langle B_m, a\rangle, \cdot) \sim \mathcal{F}_2, C_m?(!_m\mathsf{cell}(c, \langle B_m, a\rangle, \cdot) :: (a : A_m),$$

but as we have seen, with the aid of configuration extension (Lemma 16) in the case where m admits contraction, defining R by logical equivalence at B_m gives exactly the needed condition. □

Lemma 35 ($\forall L^0$). *Suppose* $\Xi \vdash B_m$ *and* $\mathsf{W} \in \sigma(\Gamma)$*. Then,*

$$\Xi \; ; \Gamma, a : \forall t_m.A_m \vdash \mathsf{thread}(c, a.\langle B_m, c\rangle) \sim \mathsf{thread}(c, a.\langle B_m, c\rangle) :: (c : A_m[B_m/t_m])$$

Proof. Fix $\eta : \Xi$ and suppose that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, a : \forall t_m.A_m$.

By definition of equivalence at universally-quantified types, then, we have that whenever $R : B^1_m \leftrightarrow B^2_m$, it holds that

$$\eta, t_m \hookrightarrow R \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(c, a.\langle B^1_m, c\rangle) \sim \mathcal{F}_2, \mathsf{thread}(c, a.\langle B^2_m, c\rangle) :: \Gamma, (c : A_m)$$

By compositionality for closed configurations (lemma 23), we then get (using R defined by logical equivalence at B_m, and $B^1_m = B^2_m = B_m$) that

$$\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(c, a.\langle B_m, c\rangle) \sim \mathcal{F}_2, \mathsf{thread}(c, a.\langle B_m, c\rangle) :: \Gamma, (c : A_m[B_m/t_m])$$

which is exactly the desired result. □

Lemma 36 ($\forall R$). *If*

$$\Xi, t_m \; ; \Gamma \vdash \mathsf{thread}(a, Q_1) \sim \mathsf{thread}(a, Q_2) :: (a : A_m)$$

then

$$\begin{aligned}\Xi \; ; \Gamma \vdash \;\; & \mathsf{thread}(c, \mathsf{case}\; c \; (\langle t_m, y\rangle \Rightarrow Q_1[y/c])) \sim \\ & \mathsf{thread}(c, \mathsf{case}\; c \; (\langle t_m, y\rangle \Rightarrow Q_2[y/c])) :: (c : \forall t_m.A_m).\end{aligned}$$

Proof. Fix $\eta : \Xi$ and suppose that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma$.

We observe that $\mathcal{F}_i, \mathsf{thread}(c, \mathsf{case}\; c \; (\langle t_m, y\rangle \Rightarrow Q_i[y/c])) \mapsto \mathcal{F}_i, !_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_i[y/c])$, so by closure under converse reduction, we only need to show these final configurations to be equivalent at $(c : \forall t_m.A_m)$. Fix some $R : B^1_m \leftrightarrow B^2_m$. Using that t_m does not occur free in

Γ (since Ξ, t_m is well-formed, and $\eta \; ; \; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma$), we can extend the relation η using Lemma 25, and get that

$$\eta, t_m \hookrightarrow R \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma$$

Combining this with our hypothesis, using the definition of equivalence for open configurations, we get that

$$\eta, t_m \hookrightarrow R \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(a, Q_1) \sim \mathcal{F}_2, \mathsf{thread}(a, Q_2) :: (a : A_m)$$

We now note that

$$\begin{aligned}&\mathcal{F}_i, \mathsf{thread}(c, \mathsf{case}\ c\ (\langle t_m, y\rangle \Rightarrow Q_i[y/c])), \mathsf{thread}(a, c.\langle B^i_m, a\rangle) \mapsto^* \\ &\mathcal{F}_i, C_m?(!_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_i[y/c]), \cdot), \mathsf{thread}(a, Q_i).\end{aligned}$$

If m admits contraction, we can extend the configuration $\mathcal{F}_i, \mathsf{thread}(a, Q_i)$ by adding the cell $!_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_i[y/c])$, giving us that

$$\begin{aligned}\eta, t_m \hookrightarrow R \; ; \cdot \vDash \quad &\mathcal{F}_1, C_m?(!_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_1[y/c]), \cdot), \mathsf{thread}(a, Q_1) \sim \\ &\mathcal{F}_2, C_m?(!_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_2[y/c]), \cdot), \mathsf{thread}(a, Q_2) :: (a : A_m)\end{aligned}$$

Closure under converse reduction then gives us that also

$$\begin{aligned}\eta, t_m \hookrightarrow R \; ; \cdot \vDash \quad &\mathcal{F}_1, !_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_1[y/c]), \mathsf{thread}(a, Q_1) \sim \\ &\mathcal{F}_2, !_m\mathsf{cell}(c, \langle t_m, y\rangle \Rightarrow Q_2[y/c]), \mathsf{thread}(a, Q_2) :: (a : A_m)\end{aligned}$$

Now, since R was chosen arbitrarily, we may apply the definition of equivalence at $\forall t_m.A_m$, and use closure under converse reduction one last time to give the result. □

In addition to the rules for typing processes, we also need logical equivalence versions of the rules for typing configurations. The empty configuration rule follows immediately from our definitions.

Lemma 37 (Empty Configuration)**.** *For any* Γ *and* Ξ*, we have* $\Xi \; ; \Gamma \vdash (\cdot) \sim (\cdot) :: \Gamma$.

For singleton configurations, we rely on closure under converse reduction to type configurations with a single cell, based on the threads that write those cells. The thread rule of configuration typing becomes an identity in the setting of these rules for equivalence. Configuration join is also straightforward, relying on the definition of equivalence for open configurations in a similar manner to the rule for the empty configuration.

Lemma 38 (Configuration join)**.** *Suppose* $\Xi \; ; \Gamma_1 \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Gamma_2$ *and* $\Xi \; ; \Gamma_2 \vDash \mathcal{C}'_1 \sim \mathcal{C}'_2 :: \Gamma_3$.

Then, $\Xi \; ; \Gamma_1 \vDash \mathcal{C}_1, \mathcal{C}'_1 \sim \mathcal{C}_2, \mathcal{C}'_2 :: \Gamma_3$.

Proof. Let $\Xi \; ; \cdot \vDash \mathcal{D}_1 \sim \mathcal{D}_2 :: \Gamma_1$.

By definition, $\Xi \sim \cdot \vDash \mathcal{D}_1, \mathcal{C}_1 \sim \mathcal{D}_2, \mathcal{C}_2 :: \Gamma_2$.

Again by definition, $\Xi \sim \cdot \vDash \mathcal{D}_1, \mathcal{C}_1, \mathcal{C}'_1 \sim \mathcal{D}_2, \mathcal{C}_2, \mathcal{C}'_2 :: \Gamma_3$, from which the result follows. □

Theorem 30 (Parametricity)**.** *If* $\Xi \; ; \Gamma \vdash \mathcal{C} :: \Delta$*, then also* $\Xi \; ; \Gamma \vdash \mathcal{C} \sim \mathcal{C} :: \Delta$.

Proof. By induction on the typing derivation, replacing each typing rule with the corresponding equivalence lemma. □

With parametricity, along with our earlier results that $\sim$ is transitive and symmetric, we have that $\sim$ is a partial equivalence relation, and, when restricted to well-typed configurations, even an equivalence relation. This is interesting in its own right, but also serves as a key component of the relation between logical and observational equivalence.

We now continue towards this overall goal, first showing that logical equivalence implies observational equivalence, and then showing the reverse direction, giving that these two equivalences coincide (at least in the setting of well-typed configurations — there are some simple counterexamples for ill-typed configurations, and some more technical work would be required to find the weakest constraints under which the two equivalences agree).

Logical Equivalence Implies Observational Equivalence

We begin by showing that logical equivalence implies observational equivalence. We define consistency of relations (with respect to a notion of observation, which we fix in this section to be extensional observation, but note that other options are possible), as well as congruence. We then demonstrate that observational equivalence is the coarsest relation satisfying these conditions, after which it suffices to show that logical equivalence is a consistent congruence, meaning that it must be a refinement of observational equivalence.

Definition 17. *We say that a family of relations $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \; R \; \mathcal{C}_2 :: \Delta$ is* consistent *if, whenever $\cdot \,;\, \cdot \vdash \mathcal{C}_1 \; R \; \mathcal{C}_2 :: \Delta$, there are $\mathcal{F}_1, \mathcal{F}_2$ final such that $\mathcal{C}_i \mapsto \mathcal{F}_i$ and $\mathcal{O}(\mathcal{F}_1, \Delta)$ is equal to $\mathcal{O}(\mathcal{F}_2, \Delta)$ up to renaming.*

We say that such a family is a congruence *if, whenever $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \; R \; \mathcal{C}_2 :: \Delta$ and $\mathcal{C}$ is such that $\Xi' \,;\, \Gamma' \vdash \mathcal{C}, \mathcal{C}_i :: \Delta'$ (for some Ξ', Γ', Δ'), we also have that*

$$\Xi' \,;\, \Gamma' \vdash \mathcal{C}, \mathcal{C}_1 \; R \; \mathcal{C}, \mathcal{C}_2 :: \Delta'.$$

Lemma 39. *Observational equivalence $\cong$ is the coarsest consistent congruence.*

Proof. It is immediate from its definition that observational equivalence is consistent, taking the observation context $\mathcal{C}'$ to be empty.

Now, suppose that $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$ and $\Xi' \,;\, \Gamma' \vdash \mathcal{C}, \mathcal{C}_i :: \Delta'$ for $i = 1, 2$. Let $\mathcal{C}'$ be such that $\Xi'' \,;\, \Gamma'' \vDash \mathcal{C}', \mathcal{C}, \mathcal{C}_i :: \Delta''$ for $i = 1, 2$. Then, taking the observation context $\mathcal{C}', \mathcal{C}$ and using the fact that $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$, we conclude that observational equivalence is a congruence.

Finally, suppose that R is a consistent congruence and that $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \; R \; \mathcal{C}_2 :: \Delta$. Suppose also that $\mathcal{C}$ is such that $\cdot \,;\, \cdot \vdash \mathcal{C}, \mathcal{C}_i :: \Delta'$. Since R is a congruence, we have that

$$\cdot \,;\, \cdot \vdash \mathcal{C}, \mathcal{C}_1 \; R \; \mathcal{C}, \mathcal{C}_2 :: \Delta'.$$

Since R is consistent, we can conclude that $\mathcal{C}, \mathcal{C}_i \mapsto \mathcal{F}_i$ final with $\mathcal{O}(\mathcal{F}_1, \Delta') = \mathcal{O}(\mathcal{F}_2, \Delta')$. This then gives us exactly that $\Xi \,;\, \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$. □

Lemma 40 (Logical Equivalence is Consistent)**.** *Logical Equivalence is a consistent family of relations.*

Proof. Suppose that $\cdot \,;\, \cdot \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$. By definition, there are $\mathcal{F}_1, \mathcal{F}_2$ final such that $\mathcal{C}_i \mapsto \mathcal{F}_i$ and $\cdot \,;\, \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$.

We now proceed by induction on Δ, using the multiset ordering of the types in Δ, seeking to show that $\mathcal{O}(\mathcal{F}_1, \Delta) = \mathcal{O}(\mathcal{F}_2, \Delta)$.

If $\Delta = (\cdot)$, then these observations are both empty as well, and so are equal.

As $\cdot\ ;\ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$, we cannot be in the case where $\Delta = a : t_m, \Delta'$, as there are no free type variables in Δ.

Observable cases If $\Delta = a : A_m^+, \Delta'$ for some positive A_m^+ other than $\exists t_m.B_m$, then in each case, we can write $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, V)$, and $\cdot\ ;\ \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Delta_V, \Delta'$, where Δ_V is a context smaller than $a : A_m^+$ in the multiset order, consisting of the (typed) addresses in the value V. Applying the inductive hypothesis, we get that

$$\mathcal{O}(C_m?(\mathcal{F}_1, \mathcal{F}_1'), (\Delta_V, \Delta')) = \mathcal{O}(C_m?(\mathcal{F}_2, \mathcal{F}_2'), (\Delta_V, \Delta')).$$

We now note that the observation $\mathcal{O}(\mathcal{F}_i, \Delta)$ can be written as

$$\{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V) \cup \mathcal{O}(\mathcal{F}_i, \Delta').$$

It will therefore suffice to show that

$$\{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V) \cup \mathcal{O}(\mathcal{F}_i, \Delta') = \{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), (\Delta_V, \Delta'))$$

By definition, $\mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), (\Delta_V, \Delta) = \mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), \Delta_V) \cup \mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), \Delta)$. We will first show that

$$\{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i, \Delta_V) = \{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V),$$

handling the first two terms of the left-hand side, and then that

$$\mathcal{O}(\mathcal{F}_i, \Delta') = \mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), \Delta'),$$

concluding the proof.

If $C \notin \sigma(m)$, then $C_m?(\mathcal{F}_i, \mathcal{F}_i') = \mathcal{F}_i'$, and so $\mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), \Delta_V) = \mathcal{O}(\mathcal{F}_i', \Delta_V)$ immediately. Otherwise, we consider two cases for $\mathcal{O}(\mathcal{F}_i, \Delta_V)$. If this observation does not contain $!_m\mathsf{cell}(a, V)$, then it must be exactly the same as $\mathcal{O}(\mathcal{F}_i', \Delta_V)$, as the only difference between $\mathcal{F}_i$ and $\mathcal{F}_i'$ is this single cell, which, if not observed, cannot affect the remainder of the observation. If it does contain $!_m\mathsf{cell}(a, V)$, there is only a single case of the definition of observation that this cell can come from, and so we can write $\mathcal{O}(\mathcal{F}_i, \Delta_V) = \{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V) \cup S$ for some $S \subseteq \mathcal{O}(\mathcal{F}_i', \Delta_V)$, as the only possible differences between the observation at $\mathcal{F}_i$ and $\mathcal{F}_i'$ must come from or after the additional observation of $!_m\mathsf{cell}(a, V)$ — prior to observing the cell at address a, observation of both configurations proceeds identically. This then means that $\mathcal{O}(\mathcal{F}_i, \Delta_V) = \{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V)$, and so also, as desired, we have

$$\{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i, \Delta_V) = \{!_m\mathsf{cell}(a, V)\} \cup \mathcal{O}(\mathcal{F}_i', \Delta_V).$$

If $\mathsf{C} \in \sigma(m)$, then $C_m?(\mathcal{F}_i, \mathcal{F}_i') = \mathcal{F}_i$, and so $\mathcal{O}(C_m?(\mathcal{F}_i, \mathcal{F}_i'), \Delta') = \mathcal{O}(\mathcal{F}_i, \Delta')$. Otherwise, we need to show that $\mathcal{O}(\mathcal{F}_i, \Delta') = \mathcal{O}(\mathcal{F}_i', \Delta')$. In this case, $\cdot\ ;\ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta_V, \Delta'$, and so by weakening (Lemma 17), also $\cdot\ ;\ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta'$. Since logical equivalence is a partial equivalence relation, we can also conclude that $\cdot\ ;\ \cdot \vdash \mathcal{F}_i' \sim \mathcal{F}_i' :: \Delta'$ (for $i = 1, 2$). Then, applying Lemma 16 to extend the configuration, we get that $\cdot\ ;\ \cdot \vdash \mathcal{F}_i' \sim \mathcal{F}_i :: \Delta'$. The inductive hypothesis then gives us exactly that $\mathcal{O}(\mathcal{F}_i', \Delta') = \mathcal{O}(\mathcal{F}_i, \Delta')$ in this case as well.

Non-observable cases If $\Delta = a : A_m^-, \Delta'$, or $\Delta = \exists t_m.B_m, \Delta'$, we note that $\mathcal{O}(\mathcal{F}_i, \Delta) = \mathcal{O}(\mathcal{F}_i, \Delta')$, as neither negative types nor existentials are observable under our definition. It will therefore suffice to show that $\mathcal{O}(\mathcal{F}_1, \Delta') = \mathcal{O}(\mathcal{F}_2, \Delta')$. By weakening (Lemma 17), we get that $\cdot \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta'$, and applying our inductive hypothesis gives the result immediately. □

Lemma 41 (Logical Equivalence is a Congruence)**.** *Logical Equivalence is a congruence.*

Proof. Suppose $\Xi \; ; \Gamma \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$ and $\mathcal{C}$ is such that $\Xi' \; ; \Gamma' \vDash \mathcal{C}, \mathcal{C}_i :: \Delta'$.

Now, we note that we can write $\mathcal{C} = \mathcal{C}_a, \mathcal{C}_b$ such that $\Xi' \; ; \Gamma' \vDash \mathcal{C}_a :: \Gamma$ and $\Xi' \; ; \Delta \vDash \mathcal{C}_b :: \Delta'$.

By parametricity, we get that $\mathcal{C}_a, \mathcal{C}_b$ are each logically equivalent to themselves. We can then use configuration join (Lemma 38) to combine $\mathcal{C}_a, \mathcal{C}_i, \mathcal{C}_b$ and conclude the desired result. □

Theorem 31. *If* $\Xi \; ; \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$*, then also* $\Xi \; ; \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$.

Proof. This follows immediately from the previous three results, as observational equivalence is the coarsest consistent congruence, and logical equivalence is a consistent congruence. □

Observational Equivalence Implies Logical Equivalence

In order to show that observational equivalence implies logical equivalence, we will first need to show that logical equivalence respects observational equivalence. We are already given a form of this for the type variable case, from the definition of admissible relations (Definition 14), but need to ensure that it extends to all cases of logical equivalence. This, along with parametricity, is then sufficient to give the result almost immediately (See Theorem 32).

Lemma 42 (Logical Equivalence Respects Observational Equivalence)**.** *Suppose that:*

- $\Xi \; ; \Gamma \vdash \mathcal{C}_1' \cong \mathcal{C}_1 :: \Delta.$
- $\Xi \; ; \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta.$
- $\Xi \; ; \Gamma \vdash \mathcal{C}_2 \cong \mathcal{C}_2' :: \Delta.$

Then also $\Xi \; ; \Gamma \vdash \mathcal{C}_1' \sim \mathcal{C}_2' :: \Delta.$

Proof. Suppose $\Xi \; ; \cdot \vdash \mathcal{C}_3 \sim \mathcal{C}_4 :: \Gamma$. By definition, then, $\Xi \; ; \cdot \vdash \mathcal{C}_1, \mathcal{C}_3 \sim \mathcal{C}_2, \mathcal{C}_4 :: \Delta$. Since $\cong$ is a congruence, we also have that $\Xi \; ; \cdot \vdash \mathcal{C}_1', \mathcal{C}_3 \cong \mathcal{C}_1, \mathcal{C}_3 :: \Delta$ and $\Xi \; ; \cdot \vdash \mathcal{C}_2, \mathcal{C}_4 \cong \mathcal{C}_2', \mathcal{C}_4 :: \Delta$.

By definition, $\mathcal{C}_1, \mathcal{C}_3 \mapsto^* \mathcal{F}_1$ and $\mathcal{C}_2, \mathcal{C}_4 \mapsto^* \mathcal{F}_2$ final, with $\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta$. Since $\mathcal{C}_1, \mathcal{C}_3 \cong \mathcal{C}_1', \mathcal{C}_3$, and the former reduces to $\mathcal{F}_1$ final, the latter must also reduce to some $\mathcal{F}_1'$ final for which $\mathcal{F}_1 \cong \mathcal{F}_1'$. Since logical equivalence is closed under converse reduction, it will suffice to show that $\Xi \; ; \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta$, giving that $\Xi \; ; \cdot \vDash \mathcal{C}_1', \mathcal{C}_3 \sim \mathcal{C}_2', \mathcal{C}_4 :: \Delta$.

Now, choose $\eta : \Xi$, and proceed by induction on Δ.

If $\Delta = (\cdot)$, the result is immediate, as any two configurations are logically equivalent at the empty context.

If $\Delta = (a : t_m), \Delta'$, then, by definition, we have that $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ with $\mathcal{F}_i^2 = \mathcal{F}_i^C$, $(a, (\mathcal{F}_1^1, \mathcal{F}_1^2)) \; \eta(t_m) \; (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$, and $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$.

Suppose that we can write $\mathcal{F}_i' = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that $\eta \; ; \cdot \vdash \mathcal{G}_i^1, \mathcal{G}_i^2 \cong \mathcal{F}_i^1, \mathcal{F}_i^2 :: \xi_i(t_m)$ and $\eta \; ; \cdot \vdash \mathcal{G}_i^2, \mathcal{G}_i^3 \cong \mathcal{F}_i^2, \mathcal{F}_i^3 :: \Delta'$.

Applying the inductive hypothesis to Δ', we get that $\eta \; ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 \sim \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta'$, and as admissible relations respect observational equivalence, $(a, (\mathcal{G}_1^1, \mathcal{G}_1^2)) \; \eta(t_m) \; (a, (\mathcal{G}_2^1, \mathcal{G}_2^2))$. The definition of logical equivalence at variable types then gives us that $\eta\ ;\ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta$, concluding this case.

If $\Delta = (a : A_m^+), \Delta'$ for some positive type A_m^+, in each case, we write $\mathcal{F}_i = !_m\mathsf{cell}(a, V), \mathcal{G}_i$, and similarly $\mathcal{F}_i' = !_m\mathsf{cell}(a, V), \mathcal{G}_i'$ — this must be possible by the definitions of $\sim$ and $\cong$, respectively. We then apply the inductive hypothesis to the $\mathcal{G}_i$ and $\mathcal{G}_i'$, at Δ_V, Δ', where Δ_V contains some components of A_m^+ (e.g., in the case where $V_i = \langle b, c\rangle$, we would have that $\Delta_V = (b : B_m), (c : C_m)$). As a context, Δ_V is smaller (in the multiset ordering) than $(a : A_m^+)$, and so this use of the inductive hypothesis is allowable, giving us that $\eta\ ;\ \cdot \vdash \mathcal{G}_1' \sim \mathcal{G}_2' :: \Delta_V, \Delta'$. Using the definition of $\sim$ at type A_m^+ again, we recover that $\eta\ ;\ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta$, as desired.

If $\Delta = (a : A_m^-), \Delta'$ for some negative type A_m^-, we consider a suitable process $\mathsf{thread}(c, a.V)$ which attempts to read from address a. The definition of logical equivalence gives us that $\eta\ ;\ \cdot \vDash \mathcal{F}_1, \mathsf{thread}(c, a.V) \sim \mathcal{F}_2, \mathsf{thread}(c, a.V) :: (c : C_m), \Delta'$, where C_m is some component of A_m^- (e.g., in the case where $V = \langle b, c\rangle$ and $A_m^- = B_m \multimap D_m$, C_m would be D_m). Since $\eta\ ;\ \cdot \vdash \mathcal{F}_1' \cong \mathcal{F}_1 :: \Delta$ and observational equivalence is a congruence, we have also that $\eta\ ;\ \cdot \vdash \mathcal{F}_1', \mathsf{thread}(c, a.V) \cong \mathcal{F}_1, \mathsf{thread}(c, a.V) :: (c : C_m), \Delta'$ (and a similar result for $\mathcal{F}_2$ and $\mathcal{F}_2'$). Applying the inductive hypothesis to $\mathcal{F}_i, \mathsf{thread}(c, a.V)$ at type $(c : C_m), \Delta'$, we get that $\eta\ ;\ \cdot \vDash \mathcal{F}_1', \mathsf{thread}(c, a.V) \sim \mathcal{F}_2', \mathsf{thread}(c, a.V) :: (c : C_m), \Delta'$. This use of the inductive hypothesis is valid, as the result for general configurations at a given Δ relies only on the result for final configurations at that same Δ. Finally, applying the definition of logical equivalence at type A_m^-, we get that $\eta\ ;\ \cdot \vdash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Delta$, as desired. □

Theorem 32. *If* $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$*, then also* $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta$.

Proof. Since $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_2 :: \Delta$, we know in particular that $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 :: \Delta$. By Parametricity (Theorem 30), we then know that $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 \sim \mathcal{C}_1 :: \Delta$, and since observational equivalence is reflexive, we also know that $\Xi\ ;\ \Gamma \vdash \mathcal{C}_1 \cong \mathcal{C}_1 :: \Delta$. The result then follows from Lemma 42, applied to these three known equivalences. □

With both directions of the relation between observational and logical equivalence, we have demonstrated that logical equivalence, as presented in section 5.5, and (extensional) observational equivalence coincide. We also conjecture that a more general result is possible, by parameterizing logical equivalence also with respect to a notion of observation, and showing that the observational and logical equivalences associated with a given observation coincide, but leave this to future work.

Example 13 (Optimizing away indirections)**.** *As a concrete motivating example of an optimization whose soundness would be useful to prove via our machinery of equivalence, we pointed near the beginning of this chapter to an implementation of a natural-deduction-based adjoint language [47], which compiles to a variant of our shared-memory language. This compilation process generates unnecessary indirections of the form* $x \leftarrow P[x/a]\ ;\ a \leftarrow x$ *and* $x \leftarrow (x \leftarrow c)\ ;\ P[x/c]$*, which, when run, cause an additional allocation and unnecessary copy/move step. As an optimization, terms of this form are simplified to just* P*, removing the extra indirection step. Note that the soundness of this optimization is not a given — in a language with support for comparing symbols (addresses) for equality, it may be possible to distinguish whether an additional copy/move operation has taken place by a suitable comparison of addresses.*

We will show now that this optimization is sound, in the sense that the unoptimized and optimized programs are logically (and therefore also observationally) equivalent, in the extensional sense. This has two parts, for the two different kinds of cuts that we can optimize out.

Suppose that P is a process term, and that $\Xi \; ; \Gamma_C, \Gamma_1 \vdash P :: (a : A_m)$. This also means that for any choice of Γ_2, we have

$$\Xi \; ; \Gamma_C, \Gamma_1, \Gamma_2 \vDash \mathsf{thread}(a, P), \mathsf{cell}(a, \_) :: \Gamma_C, \Gamma_2, (a : A_m).$$

Write $\Gamma = \Gamma_C, \Gamma_1, \Gamma_2$ and $\Delta = \Gamma_C, \Gamma_2$ — the details of these contexts will not be relevant in this example, other than that they appear in the typing rule for threads. We will also leave empty cells implicit in the remainder of this example for space reasons, and to focus attention on the more essential details.

Now, we wish to show that $\mathsf{thread}(a, P)$ *is logically equivalent to its indirect counterpart* $\mathsf{thread}(a, x \leftarrow P[x/a] \; ; \; a \leftarrow x)$. *We observe that the latter reduces in one step to the configuration* $\mathsf{thread}(a', P[a'/a]), \mathsf{thread}(a, a \leftarrow a')$ *(with a' a fresh symbol), and so by backwards closure, it will suffice to show that*

$$\Xi \; ; \Gamma \vDash \mathsf{thread}(a, P) \sim \mathsf{thread}(a', P[a'/a]), \mathsf{thread}(a, a \leftarrow a') :: \Delta, (a : A_m).$$

By parametricity (Theorem 30), we have that

$$\Xi \; ; \Gamma \vDash \mathsf{thread}(a, P) \sim \mathsf{thread}(a, P) :: \Delta, (a : A_m)$$

Suppose that $\Xi \; ; \cdot \vDash \mathcal{F}_1' \sim \mathcal{F}_2' :: \Gamma$. Then, we may conclude that $\mathcal{F}_i'$, $\mathsf{thread}(a, P) \mapsto^* \mathcal{F}_i$ *final for which*

$$\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta, (a : A_m).$$

By inversion (Lemma 12), we can write $\mathcal{F}_i = \mathcal{F}_i'', !_m\mathsf{cell}(a, D_i)$ for some data D_i. Now, since $\mathsf{thread}(a, P)$ *and* $\mathsf{thread}(a', P[a'/a])$ *are equal up to renaming (in particular, the renaming that takes a to a' and is otherwise the identity), and we identify configurations that are renamings of each other, we have also that* $\mathcal{F}_2', \mathsf{thread}(a', P[a'/a]) \mapsto^* \mathcal{F}_2'', !_m\mathsf{cell}(a', D_2)$. *This also means that*

$$\mathcal{F}_2', \mathsf{thread}(a', P[a'/a]), \mathsf{thread}(a, a \leftarrow a') \mapsto^* \mathcal{F}_2'', C_m?(!_m\mathsf{cell}(a', D_2), \cdot), !_m\mathsf{cell}(a, D_2).$$

In the case where m does not admit contraction, this is exactly equal to $\mathcal{F}_2$. Thus, since $\mathcal{F}_1', \mathsf{thread}(a, P) \mapsto^* \mathcal{F}_1$ *and* $\mathcal{F}_2', \mathsf{thread}(a', P[a'/a]), \mathsf{thread}(a, a \leftarrow a') \mapsto^* \mathcal{F}_2$, *and we know that $\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta, (a : A_m)$, we have exactly the desired result (by the definition of equivalence for open configurations).*

The case where m does admit contraction is slightly more involved — it will suffice to show that

$$\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2'', !_m\mathsf{cell}(a', D_2), !_m\mathsf{cell}(a, D_2) :: \Delta, (a : A_m),$$

at which point the same reasoning used in the case where m does not admit contraction may be applied. Since $\mathcal{F}_2 = \mathcal{F}_2'', !_m\mathsf{cell}(a, D_2)$, applying our configuration extension lemma (Lemma 16) to add $!_m\mathsf{cell}(a', D_2)$ gives the above.

For the second type of indirection, we suppose that Q is a well-typed process term, with $\Xi \, ; \Gamma_C, \Gamma_1, (c : C_k) \vdash Q :: (a : A_m)$. As in the first case, we may take any Γ_2, and get that

$$\Xi \, ; \Gamma_C, \Gamma_1, \Gamma_2, (c : C_k) \vDash \mathsf{thread}(a, Q) :: \Gamma_C, \Gamma_2, (a : A_m).$$

We again take $\Gamma = \Gamma_C, \Gamma_1, \Gamma_2$ and $\Delta = \Gamma_C, \Gamma_2$.

Now, our goal will be to to show that $\mathsf{thread}(a, Q)$ and $\mathsf{thread}(a, x \leftarrow (x \leftarrow c) \, ; Q[x/c])$ are logically equivalent. By backwards closure, it will suffice to show that

$$\Xi \, ; \Gamma, (c : C_k) \vDash \mathsf{thread}(a, Q) \sim \mathsf{thread}(c', c' \leftarrow c), \mathsf{thread}(Q[c'/c]) :: \Delta, (a : A_m),$$

with c' a fresh symbol.

Suppose that $\Xi \, ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma, (c : C_k)$. We wish to show that

$$\Xi \, ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(a, Q) \sim \mathcal{F}_2, \mathsf{thread}(c', c' \leftarrow c), \mathsf{thread}(Q[c'/c]) :: \Delta, (a : A_m).$$

We begin by applying parametricity to get that

$$\Xi \, ; \Gamma, (c : C_k) \vDash \mathsf{thread}(a, Q) \sim \mathsf{thread}(a, Q) :: \Delta, (a : A_m).$$

Applying the definition of equivalence for open configurations, using that $\mathcal{F}_1$ and $\mathcal{F}_2$ are logically equivalent at $\Gamma, (c : C_k)$, we get that

$$\Xi \, ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(a, Q) \sim \mathcal{F}_2, \mathsf{thread}(a, Q) :: \Delta, (a : A_m).$$

By transitivity, it will suffice to show that

$$\Xi \, ; \cdot \vDash \mathcal{F}_2, \mathsf{thread}(a, Q) \sim \mathcal{F}_2, \mathsf{thread}(c', c' \leftarrow c), \mathsf{thread}(Q[c'/c]) :: \Delta, (a : A_m).$$

We now apply inversion to $\mathcal{F}_i$, getting that $\mathcal{F}_i = \mathcal{F}_i', !_k\mathsf{cell}(c, D_i)$. This lets us conclude that

$$\mathcal{F}_2, \mathsf{thread}(c', c' \leftarrow c), \mathsf{thread}(a, Q[c'/c]) \mapsto \mathcal{F}_2', C_k?(!_k\mathsf{cell}(c, D_2), \cdot), !_k\mathsf{cell}(c', D_2), \mathsf{thread}(a, Q[c'/c]).$$

In the case where k does not admit contraction, this is a renaming (with c renamed to c') of $\mathcal{F}_2, \mathsf{thread}(a, Q)$, using the fact that c' was chosen fresh, and so is logically equivalent to it. If k does admit contraction, we instead apply a renaming to $\mathcal{F}_2, \mathsf{thread}(a, Q)$, renaming c to c', and apply the configuration extension lemma to add the additional cell $!_k\mathsf{cell}(c, D_2)$ to one side, giving us that

$$\Xi \, ; \cdot \vDash \mathcal{F}_2, \mathsf{thread}(a, Q) \sim \mathcal{F}_2, \mathsf{thread}(c', c' \leftarrow c), \mathsf{thread}(Q[c'/c]) :: \Delta, (a : A_m),$$

which then yields the desired result.

We therefore have that the optimization of eliminating cuts with identities as one premise is sound (with respect to our notion of equivalence).

5.6 Mode-dependent Equivalence

Thus far, we have examined several different ways to define equivalence for (shared-memory) programs in the adjoint setting. However, all of the equivalences we presented were uniform, in the sense that they behave the same at all modes, other than some details surrounding whether a given cell may be read from more than once. The fact that we can treat modes differently in pure adjoint logic (most obviously, by giving them different structural properties, but also by differently restricting what connectives and rules are available) suggests that we may be able to do the same in the context of programming and equivalence. For equivalence in particular, we might wish to say that what it means for two terms to be equivalent at mode m is not the same as what it means for terms to be equivalent at mode k, for instance with mode k having a proof-irrelevant notion of equivalence while mode m is extensional. Certainly if we examine only a single mode, we can assign many different notions of equivalence to programs, but it is less obvious how this works in a case with multiple modes. We needed to constrain structural properties so that if $k \leq m$, then $\sigma(k) \subseteq \sigma(m)$ in order to ensure that we could eliminate cuts in the pure logic, and so the question arises of what, if any, constraints need to be placed on per-mode equivalences in order to have a sensible overall system, and, for that matter, what it means to have a sensible system to begin with.

Just as in the pure logic, we wanted to ensure that we could combine rules, structural properties, and such at different modes into a coherent adjoint logic, here, we would like to combine equivalences at different modes into a coherent equivalence defined over the whole mode structure. That is, if we have (potentially partial) equivalence relations $=_k$ at each mode k, we should be able to combine them into some overall (partial) equivalence relation $=$ which can compare programs that may make use of different modes, rather than just a single one, but which is still compatible with the $=_k$ in that if P and Q are programs entirely at mode k, then $P =_k Q$ if and only if $P = Q$. Some other properties we might want $=$ to inherit from the $=_k$ include congruence, consistency (with a similarly mode-dependent notion of observation), and parametricity. We will take the last of these as our primary guide, attempting to prove parametricity in a general setting with potentially different equivalences per mode. As we will see, this provides some obvious candidates for constraints on how the various $=_k$ relate, which we can then explore further to find a simple set of conditions under which the $=_k$ can be coherently combined.

5.6.1 A strongly isolating equivalence

In order to extend per-mode equivalences into an overall equivalence for multi-mode programs, we need to handle the boundary between modes. While $=_k$ may be well-defined for programs purely at mode k, a program at mode k may still contain some components of type $\downarrow_k^m A_m$ or $\uparrow_\ell^k A_\ell$, for which $=_k$ can say little directly. Instead, we need to transition to using $=_m$ or $=_\ell$ to compare the components lying underneath these shifts.

We already have some machinery for allowing another relation to take over in determining equivalence for some portion of a program, used for handling quantified types, which we will seek to reuse for handling shifts as well. It is far from obvious that this is a correct choice, and we expect that much further work could be done on exploring possible ways to combine equivalences, perhaps motivated by concrete applications. However, we find this to be an interesting approach to take, both because the needed machinery already exists, making it a worthwhile avenue to explore, and because shifts, in a sense, form an abstraction boundary, similar to that provided by quantified types. This analogy is stronger in one direction than the other (in that lower modes may depend on higher modes, but not vice versa), but following it nevertheless leads to an interesting system.

At a high level, our approach for checking equivalence at a type A_k will be to replace any shift types $\downarrow_k^m A_m$ or $\uparrow_\ell^k A_\ell$ with fresh type variables t_k in A_k, yielding a new type A'_k, depending on several type variables. We then will check equality using $=_k$ at A'_k, instantiating each type variable with a relation defined based on the shifted type it replaces — intuitively, these relations should do as little as possible before checking $=_m$ for the underlying mode m of the shift.

In order for this approach to work, we need the relations $=_k$ to work not just for closed types, but also open ones. In particular, we need to ensure that $\eta \; ; \cdot \vdash \mathcal{F}_1 =_k \mathcal{F}_2 :: (a : t_k)$ exactly when $(a, \mathcal{F}_1) \; \eta(t_k) \; (a, \mathcal{F}_2)$, so that $=_k$ cannot do any more at type variables than query the collection η of relations instantiating those type variables. Given this extension of $=_k$, we can define an overall relation $=$ at all modes.

We say $\eta \; ; \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2 :: (a : A_k), \Delta$ if there are $\mathcal{F}_i^j$ for $i = 1, 2$ and $j = 1, 2, 3$ such that:

- $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3,$
- $\mathsf{C} \in \sigma(\mathcal{F}_i^2)$ for $i = 1, 2$
- $\eta, \hat{\eta}(A_k, \eta) \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 =_k \mathcal{F}_2^1, \mathcal{F}_2^2 :: (a : t(A_k))$
- $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta$

and $\eta \; ; \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2 :: (\cdot)$ is always true.

We already have given a definition of $=_k$, but we still need to define the type transformation $A_k \mapsto t(A_k)$ and the mapping $\hat{\eta}(A_k, \eta)$ of type variables to admissible relations:

$$\begin{array}{rcl} t(\oplus\{\ell : A_k^\ell\}_{\ell \in L}) & = & \oplus\{\ell : t(A_k^\ell)\}_{\ell \in L} \\ t(\&\{\ell : A_k^\ell\}_{\ell \in L}) & = & \&\{\ell : t(A_k^\ell)\}_{\ell \in L} \\ & \ldots & \\ t(\downarrow_k^m A_m) & = & t_{A_m} \\ t(\uparrow_\ell^k A_\ell) & = & t_{A_\ell} \end{array}$$

where we assume that each t_{A_m} is equal to t_{B_ℓ} iff $A_m = B_\ell$, and each t_{A_m} is not equal to any other type variables.

We can define $\hat{\eta}(A_k, \eta)$ similarly:

$$\begin{array}{rcl} \hat{\eta}(\oplus\{\ell : A_k^\ell\}_{\ell \in L}, \eta) & = & \otimes_{\ell \in L} \hat{\eta}(A_k^\ell, \eta) \\ \hat{\eta}(\&\{\ell : A_k^\ell\}_{\ell \in L}, \eta) & = & \otimes_{\ell \in L} \hat{\eta}(A_k^\ell, \eta) \\ & \ldots & \\ \hat{\eta}(\downarrow_k^m A_m, \eta) & = & t_{A_m} \hookrightarrow R_{\downarrow_k}(A_m, \eta) \\ \hat{\eta}(\uparrow_\ell^k A_\ell, \eta) & = & t_{A_\ell} \hookrightarrow R_{\uparrow^k}(A_\ell, \eta) \end{array}$$

Note that this needs to take the original mapping η as an argument in order to properly include η when defining the new relations $R_{\downarrow_k}$ and $R_{\uparrow^k}$.

We define $(a, \mathcal{F}_1) \; R_{\downarrow_k}(A_m, \eta) \; (a, \mathcal{F}_2)$ to be true iff $\mathcal{F}_i = \mathcal{F}'_i, !_k\mathsf{cell}(a, \mathsf{shift}(b))$ and

$$\eta, \eta(A_m) \; ; \cdot \vdash C_k?(\mathcal{F}_1, \mathcal{F}'_1) =_m C_k?(\mathcal{F}_2, \mathcal{F}'_2) :: (b : t(A_m)).$$

That is, downshifts are a positive type, are directly observable, and allow for direct comparison of the data they point to with $=_m$.

We also define $(a, \mathcal{F}_1)\ R_{\uparrow^k}(A_\ell, \eta)\ (a, \mathcal{F}_2)$ to be true iff

$$\eta, \eta(A_\ell)\ ;\ \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\mathsf{shift}(b)) =_\ell \mathcal{F}_2, \mathsf{thread}(b, a.\mathsf{shift}(b)) :: (b : t(A_\ell)).$$

Unlike downshifts, upshifts are negatively typed, and do not allow direct observation, instead being compared at mode ℓ after queried with a shift. Note that this requires that $=_\ell$ can be used to test equivalence of non-final configurations as well as final configurations. We assume for now that equivalence of non-final configurations, as in the settings of logical and observational equivalence, is determined by evaluating to a final configuration, where $=_\ell$ can be defined more directly.

This can be extended to more general configurations and to equality at open types in the usual way:

Definition 18. *We say that $\eta\ ;\ \cdot \vDash \mathcal{C}_1 = \mathcal{C}_2 :: \Delta$ if each $\mathcal{C}_i$ evaluates to a final configuration $\mathcal{F}_i$ for which $\eta\ ;\ \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2 :: \Delta$.*

Similarly, we say that $\eta\ ;\ \Gamma \vDash \mathcal{C}_1 = \mathcal{C}_2 :: \Delta$ if whenever $\eta\ ;\ \cdot \vDash \mathcal{F}'_1 = \mathcal{F}'_2 :: \Gamma$ for final $\mathcal{F}'_1, \mathcal{F}'_2$, also $\eta\ ;\ \cdot \vDash \mathcal{C}_1, \mathcal{F}'_1 = \mathcal{C}_2, \mathcal{F}'_2 :: \Delta$.

Parametricity for Mixed Equivalence

With this definition of $=$, we now would like to establish parametricity for $=$ from parametricity for the given $=_k$ relations, in order to show that we can lift parametricity at each base equivalence into the result for an overall, combined equivalence. We follow the same approach as for $\sim$, proving semantic typing rules based on $=$, which we can then use in the same structure as a given typing derivation to prove that a well-typed configuration is equivalent to itself. Most of these rules live entirely within a single mode, and so follow immediately from parametricity for the $=_k$ relations. The five rules we need to establish are for certain cuts (those that are used to prove a result at mode k using a cut formula at mode m, although our proof will work as well for single-mode cuts), as well as the four shift rules (five, if the $\alpha = 0$ and $\alpha = 1$ cases of $\downarrow L_\alpha$ are considered distinct).

In order to prove our new typing rules, we first need to establish some of the same properties that we used for $\sim$.

Lemma 43 (Joining for $=$)**.** *Suppose that Δ_1 and Δ_2 are contexts such that Δ_1, Δ_2 is well-formed, and $\mathcal{F}_i^j$ are final configurations for $i = 1, 2$ and $j = 1, 2, 3$ such that $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ is well-formed. Fix some family η of admissible relations, and suppose further that the following hold:*

1. *$\mathsf{C} \in \sigma(\mathcal{F}_i^2)$ for $i = 1, 2$.*
2. *$\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 = \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$*
3. *$\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$.*

Then, $\eta\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^1, \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_1, \Delta_2$. Likewise, the same result holds replacing final configurations $\mathcal{F}_i^j$ everywhere with general configurations $\mathcal{F}_i^j$.

Proof. The result for general configurations will follow from that for final configurations by reducing until a final configuration is reached, applying the result, and then taking the same reduction steps in reverse. A key factor in this being possible is that $\mathcal{C}_i^2$ cannot ever read from a cell provided by $\mathcal{C}_i^1$ or $\mathcal{C}_i^3$, because if it were to do so, it would get stuck in the absence of that portion, and one of the preconditions of the lemma would fail.

We proceed by induction on the context Δ_1, Δ_2. If Δ_1 is empty, then we can take a derivation of $\eta \ ; \ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$ and add the configurations $\mathcal{F}_i^1$ to each step, always being passed to be checked against the remaining context as an element is peeled off. Otherwise, suppose that $\Delta_1 = (a : A_m), \Delta_1'$. Since $\eta \ ; \ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 = \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$, we can find a split of $\mathcal{F}_i^1, \mathcal{F}_i^2$ into $\mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$, where:

- $\mathsf{C} \in \sigma(\mathcal{G}_i^2)$ for $i = 1, 2$
- $\eta, \hat{\eta}(A_m, \eta) \ ; \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_1^2 =_m \mathcal{G}_2^1, \mathcal{G}_2^2 :: (a : t(A_m))$
- $\eta \ ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 = \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_1'$.

Let $\mathcal{G}_i^4 = (\mathcal{G}_i^2, \mathcal{G}_i^3) \cap \mathcal{F}_i^1$, $\mathcal{G}_i^5 = (\mathcal{G}_i^2, \mathcal{G}_i^3) \cap \mathcal{F}_i^2$, and $\mathcal{G}_i^6 = (\mathcal{G}_i^1, \mathcal{G}_i^2) \cap \mathcal{F}_i^2$ Applying the inductive hypothesis to Δ_1', Δ_2, with the shared portion being $\mathcal{G}_i^5$, we get that

$$\eta \ ; \cdot \vdash \mathcal{G}_i^4, \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{G}_i^4, \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_1', \Delta_2$$

Now, applying the definition of $=$, taking the shared portion to be $\mathcal{G}_i^6$, we get the desired result. □

Lemma 44 (Splitting for $=$)**.** *Suppose $\eta \ ; \ \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Delta_1, \Delta_2$. Then, it is possible to write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ for $i = 1, 2$ such that*

1. *$\mathcal{F}_i^2 = \mathcal{F}_i|_C$ for $i = 1, 2$*
2. *$\eta \ ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 = \mathcal{F}_2^1, \mathcal{F}_2^2 :: \Delta_1$*
3. *$\eta \ ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta_2$.*

Proof. By induction on Δ_1. If Δ_1 is empty, then the result is immediate, taking $\mathcal{F}_i^3 = \mathcal{F}_i|_{\neg C}$.

If $\Delta_1 = (a : A_m), \Delta_1'$, then, by definition of $=$, we can write $\mathcal{F}_i = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{G}_i^3$ such that

- $\mathsf{C} \in \sigma(\mathcal{G}_i^2)$ for $i = 1, 2$
- $\eta, \hat{\eta}(A_m, \eta) \ ; \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_1^2 =_m \mathcal{G}_2^1, \mathcal{G}_2^2 :: (a : t(A_m))$
- $\eta \ ; \cdot \vdash \mathcal{G}_1^2, \mathcal{G}_1^3 = \mathcal{G}_2^2, \mathcal{G}_2^3 :: \Delta_1', \Delta_2$.

Applying the inductive hypothesis on Δ_1', we get a split of $\mathcal{G}_i^2, \mathcal{G}_i^3$ into $\mathcal{G}_i^4, \mathcal{G}_i^5, \mathcal{G}_i^6$ such that

- $\mathsf{C} \in \sigma(\mathcal{G}_i^5)$ for $i = 1, 2$
- $\eta \ ; \cdot \vdash \mathcal{G}_1^4, \mathcal{G}_1^5 = \mathcal{G}_2^4, \mathcal{G}_2^5 :: \Delta_1'$
- $\eta \ ; \cdot \vdash \mathcal{G}_1^5, \mathcal{G}_1^6 = \mathcal{G}_2^5, \mathcal{G}_2^6 :: \Delta_2$

Let $\mathcal{G}_i^7 = \mathcal{G}_i^2 \cap \mathcal{G}_i^6$ and $\mathcal{G}_i^8 = \mathcal{G}_i^3 \cap \mathcal{G}_i^6$. Now, we apply the definition of $=$ again, with the shared portion being $(\mathcal{G}_i^4, \mathcal{G}_i^5) \cap \mathcal{G}_i^2$, to get that

$$\eta \ ; \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_i^7, \mathcal{G}_1^4, \mathcal{G}_1^5 = \mathcal{G}_2^1, \mathcal{G}_2^7, \mathcal{G}_2^4, \mathcal{G}_2^5 :: \Delta_1.$$

Now, if we take:

- $\mathcal{F}_i^1 = \mathcal{G}_i^1, \mathcal{G}_i^4$
- $\mathcal{F}_i^2 = \mathcal{G}_i^5, \mathcal{G}_i^7$

- $\mathcal{F}_i^3 = \mathcal{G}_i^8$

we have the desired result. The latter two conditions can be checked by tracing through the definitions of each $\mathcal{G}_i^j$ — for example, $\mathcal{G}_i^7, \mathcal{G}_i^8 = \mathcal{G}_i^6$, and so $\mathcal{F}_i^2, \mathcal{F}_i^3 = \mathcal{G}_i^5, \mathcal{G}_i^6$. For the first condition, we are given that $\mathsf{C} \in \sigma(\mathcal{G}_i^5)$, and $\mathcal{G}_i^7$ is a subcontext of $\mathcal{G}_i^2$, which we are also given is contractible. □

These next two lemmas can then be proven from splitting and joining, in exactly the same manner as their counterparts for $\sim$.

Lemma 45 (Type Extension for $=$)**.** *Suppose* $\Xi \; ; \; \Gamma \vdash \mathcal{F}_1 = \mathcal{F}_2 :: \Delta$ *and that* Θ *is a context which shares no symbols with* $\Gamma, \Delta, \mathcal{C}_1, \mathcal{C}_2$*. Then also* $\Xi \; ; \Gamma, \Theta \vdash \mathcal{F}_1 = \mathcal{F}_2 :: \Delta, \Theta$.

Lemma 46 (Reuse for $=$)**.** *Suppose* $\Xi \; ; \; \Gamma_1, \Gamma_2 \vDash \mathcal{C}_1 = \mathcal{C}_2 :: \Delta$*. Suppose also that* $\mathsf{C} \in \sigma(\Gamma_2)$*, and that* Δ, Γ_2 *is well-formed. Then,* $\Xi \; ; \Gamma_1, \Gamma_2 \vDash \mathcal{C}_1 \sim \mathcal{C}_2 :: \Delta, \Gamma_2$.

With these general results established, we can now move on to the typing rules.

Lemma 47 (Cross-mode cut)**.** *Suppose* $\mathsf{C} \in \sigma(\Gamma_2)$.

If

$$\Xi \; ; \Gamma_1, \Gamma_2 \vDash \mathsf{thread}(y, P_1) = \mathsf{thread}(y, P_2) :: (y : A_m)$$

and

$$\Xi \; ; \Gamma_2, \Gamma_3, (y : A_m) \vDash \mathsf{thread}(z, Q_1) = \mathsf{thread}(z, Q_2) :: (z : C_k)$$

then $\Xi \; ; \Gamma_1, \Gamma_2, \Gamma_3 \vDash \mathsf{thread}(x, y \leftarrow P_1 \; ; \; Q_1) = \mathsf{thread}(x, y \leftarrow P_2 \; ; \; Q_2) :: (z : C_k)$.

Proof. Suppose $\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim \mathcal{F}_2 :: \Gamma_1, \Gamma_2, \Gamma_3$.

By Lemma 45 and our first assumption,

$$\Xi \; ; \Gamma_1, \Gamma_2, \Gamma_3 \vDash \mathsf{thread}(y, P_1) = \mathsf{thread}(y, P_2) :: \Gamma_3, (y : A_m).$$

Then, Lemma 46 allows us to conclude (as $\mathsf{C} \in \sigma(\Gamma_2)$) that

$$\Xi \; ; \Gamma_1, \Gamma_2, \Gamma_3 \vDash \mathsf{thread}(y, P_1) = \mathsf{thread}(y, P_2) :: \Gamma_2, \Gamma_3, (y : A_m).$$

It follows that $\Xi \; ; \; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, P_1) = \mathcal{F}_2, \mathsf{thread}(y, P_2) :: \Gamma_2, \Gamma_3, (y : A_m)$. From the definition of equivalence, and now using our second assumption, we may conclude that

$$\Xi \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, P_1), \mathsf{thread}(z, Q_1) = \mathcal{F}_2, \mathsf{thread}(y, P_2), \mathsf{thread}(z, Q_2) :: (z : C_k).$$

Closure under converse reduction (which follows almost immediately from the definition of $=$ on non-final configurations, as it did for $\sim$) then gives the desired result. □

Lemma 48 ($\downarrow R^0$)**.** *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then $\Xi \; ; \Gamma, (y : A_m) \vDash \mathsf{thread}(x, x.\mathsf{shift}(y)) = \mathsf{thread}(x, x.\mathsf{shift}(y)) :: (x : \downarrow_k^m A_m)$.

Proof. Suppose $\Xi \; ; \cdot \vDash \mathcal{F}_1 = \mathcal{F}_2 :: \Gamma, (y : A_m)$. Then, by definition, we have that

$$\mathcal{F}_i, \mathsf{thread}(x, x.\mathsf{shift}(y)) \mapsto^* \mathcal{F}_i, !_k\mathsf{cell}(x, \mathsf{shift}(y)),$$

and we wish to show that $\Xi \; ; \; \cdot \vdash \mathcal{F}_1, !_k\mathsf{cell}(x, \mathsf{shift}(y)) = \mathcal{F}_2, !_k\mathsf{cell}(x, \mathsf{shift}(y)) :: (x : \downarrow_k^m A_m)$. This follows almost immediately, however, from the definitions of $R_{\downarrow_k}(A_m)$ and $=$. □

Lemma 49 ($\downarrow L_\alpha$). *If*

$$\Xi \; ; \Gamma, (x : \downarrow_k^m A_m)^\alpha, y : A_m \vDash \mathsf{thread}(z, Q_1) = \mathsf{thread}(z, Q_2) :: (z : C_r),$$

then

$$\begin{aligned}\Xi \; ; \Gamma, x : \downarrow_k^m A_m \vDash & \mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_1)) = \\ & \mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_2)) :: (z : C_r).\end{aligned}$$

Proof. Let $\eta : \Xi$ be arbitrary.

Suppose $\eta \; ; \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2 :: \Gamma, x : \downarrow_k^m A_m$. By definition, we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ with $\mathsf{C} \in \sigma(\mathcal{F}_i^2)$, and

$$\eta, t_{A_m} \hookrightarrow R_{\downarrow_k}(A_m, \eta) \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 = \mathcal{F}_2^1, \mathcal{F}_2^2 :: (x : t_{A_m})$$

and

$$\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma.$$

From this, we can conclude that $\mathcal{F}_i^1, \mathcal{F}_i^2 = !_k \mathsf{cell}(x, \mathsf{shift}(y)), \mathcal{F}_i'$ for which

$$\eta, \eta(A_m) \; ; \cdot \vdash C_k?((\mathcal{F}_1^1, \mathcal{F}_1^2), \mathcal{F}_1') =_m C_k?((\mathcal{F}_2^1, \mathcal{F}_2^2), \mathcal{F}_2') :: (y : t(A_m)).$$

Combining configurations again, this means that

$$\mathcal{F}_i, \mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_i)) \mapsto C_k?((\mathcal{F}_i^1, \mathcal{F}_i^2), \mathcal{F}_i'), \mathcal{F}_i^3, \mathsf{thread}(z, Q_i),$$

and so it will suffice to show that

$$\eta \; ; \cdot \vdash C_k?((\mathcal{F}_1^1, \mathcal{F}_1^2), \mathcal{F}_1'), \mathcal{F}_1^3, \mathsf{thread}(z, Q_i) \sim C_k?((\mathcal{F}_2^1, \mathcal{F}_2^2), \mathcal{F}_2'), \mathsf{thread}(z, Q_i) :: (z : C_r).$$

If $\alpha = 0$, we can apply the definition of $=$ again to get that

$$\eta \; ; \cdot \vdash C_k?((\mathcal{F}_1^1, \mathcal{F}_1^2), \mathcal{F}_1'), \mathcal{F}_1^3 = C_k?((\mathcal{F}_2^1, \mathcal{F}_2^2), \mathcal{F}_2'), \mathcal{F}_2^3 :: \Gamma, (y : A_m),$$

with the shared portion being $\mathcal{F}_i^2$ — note that if k is not contractible, then $\mathcal{F}_i'$ still contains all of $\mathcal{F}_i^2$, as the cell that was removed from $\mathcal{F}_i^1, \mathcal{F}_i^2$ to give $\mathcal{F}_i'$ was at mode k. Combining this with the initial hypothesis then gives the result.

If $\alpha = 1$, we know that $\mathsf{C} \in \sigma(k)$ (and, since $m \geq k$, that $\mathsf{C} \in \sigma(m)$, as well). We therefore know that $\eta, \eta(A_m) \; ; \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 =_m \mathcal{F}_2^1, \mathcal{F}_2^2 :: (y : t(A_m))$. By definition of $=$, we can conclude that $\eta \; ; \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2; : \Gamma, (y : A_m)$. Now, since k is contractible, we may assume that $\mathcal{F}_i^1$ was empty, and so $\eta, t_{A_m} \hookrightarrow R_{\downarrow_k}(A_m, \eta) \; ; \cdot \vdash \mathcal{F}_1^2 =_k \mathcal{F}_2^2 :: (x : t_{A_m})$. Combining this with the above, using the definition of $=$, we get that $\eta \; ; \cdot \vdash \mathcal{F}_1 = \mathcal{F}_2 :: \Gamma, (x : \downarrow_k^m A_m)^\alpha, (y : A_m)$, which, when combined with our initial hypothesis, gives the result. □

Lemma 50 ($\uparrow R$). *If*

$$\Xi \; ; \Gamma \vDash \mathsf{thread}(x, P_1) = \mathsf{thread}(x, P_2) :: (x : A_k)$$

then

$$\Xi \; ; \Gamma \vDash \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_1)) = \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_2)) :: (y : \uparrow_k^m A_k).$$

Proof. Suppose $\Xi\ ;\ \cdot \vDash \mathcal{F}_1 = \mathcal{F}_2 :: \Gamma$. Consider $\mathcal{F}_i, \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_i))$. This configuration immediately steps to $\mathcal{F}_i, !_m\mathsf{cell}(y, (\mathsf{shift}(x) \Rightarrow P_i))$. On probing each such configuration with the process $\mathsf{thread}(z, y.\mathsf{shift}(z))$, as specified by $R_{\uparrow^k}(A_k)$, we can step to $\mathcal{F}_i, C_m?(!_m\mathsf{cell}(y, (\mathsf{shift}(x) \Rightarrow P_i)), \cdot), \mathsf{thread}(x, P_i)$. The result then follows from our hypothesis, the definition of $=$ at A_k, and the definition of $R_{\uparrow^k}(A_k)$. □

Lemma 51 ($\uparrow L^0$)**.** *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then,

$$\Xi\ ;\ \Gamma, (x : \uparrow_k^m A_k) \vDash \mathsf{thread}(y, x.\mathsf{shift}(y)) = \mathsf{thread}(y, x.\mathsf{shift}(y)) :: (y : A_k).$$

Proof. Suppose $\Xi\ ;\ \cdot \vDash \mathcal{F}_1 = \mathcal{F}_2 :: \Gamma, (x : \uparrow_k^m A_k)$.

By definition of $=$, we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathsf{C} \in \sigma(\mathcal{F}_i^2)$
2. $\eta, \hat{\eta}(A_k, \eta)\ ;\ \cdot \vdash \mathcal{F}_1^1, \mathcal{F}_1^2 =_m \mathcal{F}_2^1, \mathcal{F}_2^2 :: (x : t_{A_k})$.
3. $\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma$.

Now, by definition of $=_m$ at a type variable t_{A_k} and the definitions of $\hat{\eta}$ and $R_{\uparrow^m}(A_k, \eta)$, we get that

$$\eta, \hat{\eta}(A_k, \eta)\ ;\ \cdot \vDash \mathcal{F}_1^1, \mathcal{F}_1^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) =_k \mathcal{F}_2^1, \mathcal{F}_2^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) :: (y : t(A_k)).$$

Combining this again with $\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 = \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma$ using the definition of $=$ we get that $\eta\ ;\ \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, x.\mathsf{shift}(y)) = \mathcal{F}_2, \mathsf{thread}(y, x.\mathsf{shift}(y)) :: (y : A_k)$. □

We can conclude, overall, that if we define equivalence in this way, with strong separation at mode boundaries via transitioning from one relation $=_k$ to another $=_m$, that the resulting equivalence is sensible, satisfying parametricity.

A question still remains, however, of how this relates to observational equivalence. One natural idea is that if each $=_k$ arises from a choice of observation at mode k, then the relation $=$ defined from these $=_k$ should similarly correspond to the notion of observation which first checks the mode of an object, and then observes it according to its mode, with shifts being handled extensionally. This is not something we explore in detail, but we conjecture that it should hold, with at most minor further restrictions (notably, it is probably necessary that the $=_k$ are admissible relations with respect to observation at mode k).

5.6.2 Upwards-Closed Sets of Modes and Erasure

While the previous section deals very generally with the idea of an equivalence that treats modes differently, its key drawback is that the modes are, in essence, entirely separated from each other. While modes k and m may have different notions of equivalence, all information at mode k is treated with the mode-k equivalence, and likewise for mode m, even if that information at mode m lies under a shift $\downarrow_k^m$ to mode k. Intuitively, we might expect that any information that a process at mode k can depend on — that is, which is at a mode m with $m \geq k$ — should somehow be observable by the standards of mode k, and so that, for instance, if k treats equivalence extensionally and m treats equivalence in a proof-irrelevant fashion, there may be configurations which are equivalent from the perspective of mode m, but which are distinct from the perspective of mode k. A process at mode k which makes use of these m-equivalent configurations may then nevertheless be able to perform some tests that distinguish them, and so, from the perspective of observational equivalence, takes advantage of the observation at mode k being more precise than that at mode m. A full exploration of this idea is beyond the scope of this thesis, but we will look at a particularly useful example, in which modes either treat equality extensionally or proof-irrelevantly.

For our example, we will work with a set $\mathcal{M}$ of modes, a subset of which, $\mathcal{E}$, have extensional equivalence, while the remainder are treated proof-irrelevantly. A reasonable notion of equivalence in this setting should agree with usual extensional equivalence if $\mathcal{E}$ is all of $\mathcal{M}$, and should agree with proof-irrelevant equivalence if $\mathcal{E}$ is empty. We would similarly expect such a notion of equivalence to agree with extensional equivalence for any process purely at mode $\mathcal{E}$. The main question in defining such an equivalence is how to treat processes or configurations that span multiple modes, some of which may be in $\mathcal{E}$, while some are not. Our intuition above suggests that a process at mode k, which may depend on some data at a mode $m \geq k$, should be able to use the notion of equivalence at mode k when looking at this data it depends on, capturing the idea that a process at mode k may read data at mode m and use it to produce some (distinguishable) result at mode k. In other words, if we are using one type of equivalence and come across a downshift, we should continue using the same equivalence underneath that shift. Upshifts, by contrast, should change the type of equivalence we are using to that of the underlying mode of the shift, reflecting that a process at mode k cannot depend on information at lower modes.

With these constraints in mind, we will set out to define such a notion of equivalence using the techniques we have already explored, and we will observe that if $\mathcal{E}$ is an upwards-closed subset of $\mathcal{M}$ (that is, if $k \leq m$ and $k \in \mathcal{E}$, this implies that also $m \in \mathcal{E}$), we can adapt our existing definition of extensional equivalence to this setting as well.

Definition 19. *Suppose that $\mathcal{E}$ is an upwards-closed subset of $\mathcal{M}$. We will define a type-indexed family of equivalences $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Delta$ in the usual way, by recursion over the type Δ.*

We will also make use of the relation $\eta \; ; \cdot \vDash \mathcal{C}_1 \sim_{\mathcal{E}} \mathcal{C}_2 :: \Delta$, which holds when $\mathcal{C}_i \mapsto^* \mathcal{F}_i$ final such that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Delta$.

Note that the structure of this definition is much the same as that for (logical) extensional equivalence in section 5.5, and will make use of the same general concepts.

(1) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (\cdot)$ *always holds.*

(2) If $m \notin \mathcal{E}$, then $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : A_m), \Delta'$ whenever we can write each $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

- *$\mathcal{F}_i^2 = \mathcal{F}_i \mid_C$ for $i = 1, 2$.*
- *$\eta \; ; \cdot \vdash \mathcal{F}_i^1, \mathcal{F}_i^2 \sim \mathcal{F}_i^1, \mathcal{F}_i^2 :: (a : A_m)$ for $i = 1, 2$ — that is, each $\mathcal{F}_i^1, \mathcal{F}_i^2$ is extensionally equivalent to itself at $(a : A_m)$ — this ensures that these portions of the configuration are well-formed.*
- *$\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim_{\mathcal{E}} \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$.*

For the remaining cases, we will assume $m \in \mathcal{E}$.

(3) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : t_m), \Delta'$ *if we can write each $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:*

- $\mathcal{F}_i^2 = \mathcal{F}_i \mid_C$ *for* $i = 1, 2$.
- $(a, (\mathcal{F}_1^1, \mathcal{F}_1^2)) \; \eta(t_m) \; (a, (\mathcal{F}_2^1, \mathcal{F}_2^2))$.
- $\eta \; ; \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim_{\mathcal{E}} \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Delta'$.

(4) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \oplus\{\ell : A_m^\ell\}_{\ell \in L}), \Delta'$ *if there is some* j *in* L *such that*

- *each* $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, j(b))$
- $\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim_{\mathcal{E}} C_m(\mathcal{F}_2, \mathcal{F}_2') :: (b : A_m^j), \Delta'$

(5) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : B_m \otimes C_m), \Delta'$ *if*

- $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle b, c \rangle)$
- $\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim_{\mathcal{E}} C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : B_m), (c : C_m), \Delta'$

(6) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \mathbf{1}_m), \Delta'$ *if*

- $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle \rangle)$
- $\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim_{\mathcal{E}} C_m?(\mathcal{F}_2, \mathcal{F}_2') :: \Delta'$.

(7) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \downarrow_m^\ell A_\ell), \Delta'$ *if*

- $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \mathsf{shift}(b))$
- $\eta \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim_{\mathcal{E}} C_m?(\mathcal{F}_2, \mathcal{F}_2')$

Note that because $\mathcal{E}$ *is upwards-closed,* $m \in \mathcal{E}$*, and* $m \leq \ell$*, we must also have that* $\ell \in \mathcal{E}$*, and so this definition continues to use the same equivalence when moving through the downshift, as desired.*

(8) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \exists t_m . B_m), \Delta'$ *if*

- $\mathcal{F}_i = \mathcal{F}_i', !_m\mathsf{cell}(a, \langle A_m^i, b \rangle)$
- *There exists some* $R : A_m^1 \leftrightarrow A_m^2$ *such that* $\eta, t_m \hookrightarrow R \; ; \cdot \vdash C_m?(\mathcal{F}_1, \mathcal{F}_1') \sim_{\mathcal{E}} C_m?(\mathcal{F}_2, \mathcal{F}_2') :: (b : B_m), \Delta'$

(9) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \&\{\ell : A_m^\ell\}_{\ell \in L}), \Delta'$ *if*

- *For each* $j \in L$*, it holds that* $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.j(b)) \sim_{\mathcal{E}} \mathcal{F}_2, \mathsf{thread}(b, a.j(b)) :: (b : A_m^j), \Delta'$

(10) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : B_m \multimap C_m), \Delta'$ *if*

- *Whenever* $\eta \; ; \cdot \vdash \mathcal{F}_1 \mid_C, \mathcal{F}_1' \sim_{\mathcal{E}} \mathcal{F}_2 \mid_C, \mathcal{F}_2' :: (b : B_m)$*, it also holds that* $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathcal{F}_1', \mathsf{thread}(c, a.\langle b, c \rangle) \sim_{\mathcal{E}} \mathcal{F}_2, \mathcal{F}_2', \mathsf{thread}(c, a.\langle b, c \rangle) :: (c : C_m), \Delta'$

(11) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \uparrow_k^m A_k), \Delta'$ *if*

- $\eta \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\mathsf{shift}(b)) \sim_{\mathcal{E}} \mathcal{F}_2, \mathsf{thread}(b, a.\mathsf{shift}(b)) :: (b : A_k), \Delta'$

While we do not explicitly switch equivalences here, we have switched from trying to test equivalence at mode m *to trying to test equivalence at mode* k*, and so the next step that examines* A_k *will first check if* $k \in \mathcal{E}$ *before continuing, thus switching to proof-irrelevant equivalence if necessary.*

(12) $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: (a : \forall t_m . B_m), \Delta'$ *if*

- *For any choice of* A_m^1*,* A_m^2*, and* $R : A_m^1 \leftrightarrow A_m^2$*, it holds that* $\eta, t_m \hookrightarrow R \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(b, a.\langle A_m^1, b \rangle) \sim_{\mathcal{E}} \mathcal{F}_2, \mathsf{thread}(b, a.\langle A_m^2, b \rangle) :: (b : B_m), \Delta'$

It is easy to see by comparing this definition with that for our usual extensional equivalence $\sim$ that if $\mathcal{E} = \mathcal{M}$, the two agree, as they only differ in the case where $m \notin \mathcal{E}$. That it agrees with proof-irrelevant equivalence when $\mathcal{E}$ is empty is somewhat less obvious — we would like to say that $\eta \; ; \cdot \vdash \mathcal{F}_1 \sim_\emptyset \mathcal{F}_2 :: \Delta$ if and only if $\eta \; ; \cdot \vdash \mathcal{F}_i \sim \mathcal{F}_i :: \Delta$ (for $i = 1, 2$), but we only have that $\mathcal{F}_i$ can be split up into segments where each segment is extensionally equivalent to itself at one element of Δ. However, the joining and splitting lemmas (lemmas 19 and 20) for extensional equivalence allow us to convert between these two different presentations and conclude that we do indeed get a proof-irrelevant equivalence when $\mathcal{E}$ is empty.

We can also easily extend this definition to open configurations in the usual way, defining $\Xi \; ; \Gamma \vDash \mathcal{C}_1 \sim_\mathcal{E} \mathcal{C}_2 :: \Delta$ to be true if whenever $\Xi \; ; \cdot \vDash \mathcal{F}_1' \sim_\mathcal{E} \mathcal{F}_2' :: \Gamma$, it also holds that $\Xi \; ; \cdot \vDash \mathcal{C}_1, \mathcal{F}_1' \sim_\mathcal{E} \mathcal{C}_2, \mathcal{F}_2' :: \Delta$.

We now would like to show that this definition satisfies parametricity, and so is a reasonable choice of equivalence. Most of the lemmas we use in proving parametricity (including the many general lemmas not dealing with particular typing rules) for $\sim$ hold for $\sim_\mathcal{E}$ as well — for modes in $\mathcal{E}$, with the same proof, and for modes not in $\mathcal{E}$, using the corresponding lemma for $\sim$ itself. The cases we need to examine, therefore, are those involving multiple modes, which could possibly cross into or out of $\mathcal{E}$, namely the rules for shifts and that for cut.

Lemma 52 (Cross-mode cut)**.** *Suppose* $\mathsf{C} \in \sigma(\Gamma_2)$ *and* $\Gamma_1, \Gamma_2 \geq m \geq k$.

If

$$\Xi \; ; \Gamma_1, \Gamma_2 \vDash \mathsf{thread}(y, P_1) \sim_\mathcal{E} \mathsf{thread}(y, P_2) :: (y : A_m)$$

and

$$\Xi \; ; \Gamma_2, \Gamma_3, (y : A_m) \vDash \mathsf{thread}(z, Q_1) \sim_\mathcal{E} \mathsf{thread}(z, Q_2) :: (z : C_k)$$

then $\Xi \; ; \Gamma_1, \Gamma_2, \Gamma_3 \vDash \mathsf{thread}(z, y \leftarrow P_1 \; ; Q_1) \sim_\mathcal{E} \mathsf{thread}(z, y \leftarrow P_2 \; ; Q_2) :: (z : C_k)$.

Proof. Suppose $\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim_\mathcal{E} \mathcal{F}_2 :: \Gamma_1, \Gamma_2, \Gamma_3$. Then, we observe that

$$\mathcal{F}_i, \mathsf{thread}(z, y \leftarrow P_i \; ; Q_i) \mapsto \mathcal{F}_i, \mathsf{thread}(y, P_i), \mathsf{thread}(z, Q_i),$$

and so, by hypothesis and by the closure of $\sim_\mathcal{E}$ under converse reduction, it suffices to show that

$$\Xi \; ; \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, P_1) \sim_\mathcal{E} \mathcal{F}_2, \mathsf{thread}(y, P_2) :: \Gamma_2, \Gamma_3, (y : A_m).$$

This then follows from the initial hypothesis, applying type extension (Lemma 21) to add Γ_3 to both sides, and then reuse (Lemma 22) to add Γ_2 to the right-hand side. □

Lemma 53 ($\downarrow R^0$)**.** *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then $\Xi \; ; \Gamma, (y : A_m) \vDash \mathsf{thread}(x, x.\mathsf{shift}(y)) \sim_\mathcal{E} \mathsf{thread}(x, x.\mathsf{shift}(y)) :: (x : \downarrow_k^m A_m)$.

Proof. Suppose $\Xi \; ; \cdot \vdash \mathcal{F}_1 \sim_\mathcal{E} \mathcal{F}_2 :: \Gamma, (y : A_m)$. We also see that

$$\mathcal{F}_i, \mathsf{thread}(x, x.\mathsf{shift}(y)) \mapsto^* \mathcal{F}_i, !_k\mathsf{cell}(x, \mathsf{shift}(y)),$$

and would like to show that

$$\Xi \; ; \cdot \vdash \mathcal{F}_1, !_k\mathsf{cell}(x, \mathsf{shift}(y)) \sim_\mathcal{E} \mathcal{F}_2, !_k\mathsf{cell}(x, \mathsf{shift}(y)) :: (x : \downarrow_k^m A_m).$$

If $k \in \mathcal{E}$, then as for $\sim$, the result is nearly immediate from the definition of $\sim_{\mathcal{E}}$ at downshift types, potentially using an extension lemma if k admits contraction. If $k \notin \mathcal{E}$, we first note that we can easily show $\Xi \,;\, \cdot \vdash \mathcal{F}_i \sim \mathcal{F}_i :: \Gamma, (y : A_m)$ by transitivity and symmetry of $\sim$. It then remains to show that $\Xi \,;\, \cdot \vdash \mathcal{F}_i, !_k\mathsf{cell}(x, \mathsf{shift}(y)) \sim \mathcal{F}_i, !_k\mathsf{cell}(x, \mathsf{shift}(y)) :: (x : {\downarrow}_k^m A_m)$, which is again almost immediate from the definition. □

Lemma 54 (${\downarrow}L_\alpha$). *If*

$$\Xi \,;\, \Gamma, (x : {\downarrow}_k^m A_m)^\alpha, y : A_m \vDash \mathsf{thread}(z, Q_1) \sim_{\mathcal{E}} \mathsf{thread}(z, Q_2) :: (z : C_r),$$

then

$$\begin{aligned}\Xi \,;\, \Gamma, x : {\downarrow}_k^m A_m \vDash &\mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_1)) \sim_{\mathcal{E}} \\ &\mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_2)) :: (z : C_r).\end{aligned}$$

Proof. Suppose $\Xi \,;\, \cdot \vDash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Gamma, x : {\downarrow}_k^m A_m$, and consider $\mathcal{F}_i, \mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_i))$.

Regardless of whether $k \in \mathcal{E}$ or not, we can conclude that $\mathcal{F}_i = \mathcal{F}'_i, !_k\mathsf{cell}(x, \mathsf{shift}(w))$, and that $\Xi \,;\, \cdot \vdash C_k?(\mathcal{F}_i, \mathcal{F}'_i) \sim_{\mathcal{E}} C_k?(\mathcal{F}_j, \mathcal{F}'_j) :: \Gamma, (w : A_m)$ for either $i = 1$ and $j = 2$ if $k \in \mathcal{E}$, or $(i, j) \in \{(1, 1), (2, 2)\}$ if $k \notin \mathcal{E}$. In either case, we have sufficient information to say that $\mathcal{F}_i, \mathsf{thread}(z, \mathsf{case}\ x\ (\mathsf{shift}(y) \Rightarrow Q_i)) \mapsto C_k?(\mathcal{F}_i, \mathcal{F}'_i), \mathsf{thread}(z, Q_i[w/y])$ for both $i = 1$ and $i = 2$.

We now consider whether $r \in \mathcal{E}$ or not. If $r \in \mathcal{E}$, then, since $\mathcal{E}$ is upwards-closed, we also have that $k, m \in \mathcal{E}$. Using Lemma 26 (adapted to $\sim_{\mathcal{E}}$, which is possible since $k, m \in \mathcal{E}$), we get that $\Xi \,;\, \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Gamma, (x : {\downarrow}_k^m A_m)^\alpha, (w : A_m)$. Renaming w to y then gives us that $\Xi \,;\, \cdot \vdash \mathcal{F}_1, \mathsf{thread}(z, Q_1) \sim_{\mathcal{E}} \mathcal{F}_2, \mathsf{thread}(z, Q_2) :: (z : C_r)$, and closure under converse reduction completes this case.

If $r \notin \mathcal{E}$, it will suffice to show that $\Xi \,;\, \cdot \vdash \mathcal{F}_i \sim \mathcal{F}_i :: \Gamma, (x : {\downarrow}_k^m A_m)^\alpha, w : A_m$, and our initial assumption, along with closure under converse reduction and a renaming of w to y, will then give the desired result. We know that $\Xi \,;\, \cdot \vdash \mathcal{F}_i \sim \mathcal{F}_i :: \Gamma, (x : {\downarrow}_k^m A_m)$, and applying Lemma 26 (after fixing some η) gives us that $\Xi \,;\, \cdot \vdash \mathcal{F}_i \sim \mathcal{F}_i :: \Gamma, (x : {\downarrow}_k^m A_m)^\alpha, (w : A_m)$, as desired. □

Lemma 55 (${\uparrow}R$). *If*

$$\Xi \,;\, \Gamma \vDash \mathsf{thread}(x, P_1) \sim_{\mathcal{E}} \mathsf{thread}(x, P_2) :: (x : A_k)$$

then

$$\Xi \,;\, \Gamma \vDash \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_1)) \sim_{\mathcal{E}} \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_2)) :: (y : {\uparrow}_k^m A_k).$$

Proof. Suppose $\Xi \,;\, \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Gamma$, and fix $\eta : \Xi$.

If $k \in \mathcal{E}$, then also $m \in \mathcal{E}$, and so this proof can proceed as for $\sim$.

Suppose therefore that $k \notin \mathcal{E}$. By definition, we can then conclude that

$$\eta \,;\, \cdot \vDash \mathcal{F}_i, \mathsf{thread}(x, P_i) \sim \mathcal{F}_i, \mathsf{thread}(x, P_i) :: (x : A_k).$$

Consider the configuration $\mathcal{F}_i, \mathsf{thread}(y, \mathsf{case}\ y\ (\mathsf{shift}(x) \Rightarrow P_i))$. This reduces in one step to $\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i)$, and so it will suffice to show that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_1, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_1) \sim_{\mathcal{E}} \mathcal{F}_2, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_2) :: (y : \uparrow_k^m A_k).$$

If $m \in \mathcal{E}$, so this shift crosses the boundary of $\mathcal{E}$, then we need to show that

$$\begin{aligned}\eta\ ;\ \cdot \vDash &\mathcal{F}_1, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_1), \mathsf{thread}(z, y.\mathsf{shift}(z)) \sim_{\mathcal{E}}\\ &\mathcal{F}_2, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_2), \mathsf{thread}(z, y.\mathsf{shift}(z)) :: (z : A_k).\end{aligned}$$

Observe that $\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, y.\mathsf{shift}(z))$ reduces in one step to the configuration $\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, P_i[z/x])$. If we attempt to show that

$$\begin{aligned}\eta\ ;\ \cdot \vdash &\mathcal{F}_1, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_1), \mathsf{thread}(z, P_1[z/x]) \sim_{\mathcal{E}}\\ &\mathcal{F}_2, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_2), \mathsf{thread}(z, P_2[z/x]) :: (z : A_k)\end{aligned}$$

we find that we need to have

$$\begin{aligned}\eta\ ;\ \cdot \vdash &\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, P_i[z/x]) \sim\\ &\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, P_i[z/x]) :: (z : A_k)\end{aligned}$$

This follows from $\eta\ ;\ \cdot \vdash \mathcal{F}_i, \mathsf{thread}(x, P_i) \sim \mathcal{F}_i, \mathsf{thread}(x, P_i) :: (x : A_k)$ by renaming and configuration extension, concluding this case.

If $m \notin \mathcal{E}$, then we need to show that

$$\eta\ ;\ \cdot \vdash \mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i) \sim \mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i)) :: (y : \uparrow_k^m A_k).$$

Following the definition of $\sim$ at $\uparrow_k^m A_k$, we need to show that

$$\begin{aligned}\eta\ ;\ \cdot \vDash &\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, y.\mathsf{shift}(z)) \sim\\ &\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, y.\mathsf{shift}(z)) :: (z : A_k).\end{aligned}$$

As for the case where $m \in \mathcal{E}$, we observe that $\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, y.\mathsf{shift}(z))$ reduces in one step to $\mathcal{F}_i, !_m\mathsf{cell}(y, \mathsf{shift}(x) \Rightarrow P_i), \mathsf{thread}(z, P_i[z/x])$. Using symmetry and transitivity of $\sim$, we get that $\eta\ ;\ \cdot \vdash \mathcal{F}_i, \mathsf{thread}(x, P_i) \sim \mathcal{F}_i, \mathsf{thread}(x, P_i) :: (x : A_k)$. Renaming and configuration extension then give the desired result. □

Lemma 56 ($\uparrow L^0$). *Suppose* $\mathsf{W} \in \sigma(\Gamma)$.

Then,

$$\Xi\ ;\ \Gamma, (x : \uparrow_k^m A_k) \vDash \mathsf{thread}(y, x.\mathsf{shift}(y)) \sim_{\mathcal{E}} \mathsf{thread}(y, x.\mathsf{shift}(y)) :: (y : A_k).$$

Proof. Suppose $\Xi\ ;\ \cdot \vdash \mathcal{F}_1 \sim_{\mathcal{E}} \mathcal{F}_2 :: \Gamma, (x : \uparrow_k^m A_k)$, and fix $\eta : \Xi$.

If $k \in \mathcal{E}$, note that $m \in \mathcal{E}$ as well, and so the proof can continue as for $\sim$.

If $k \notin \mathcal{E}$ but $m \in \mathcal{E}$, we have by definition of equivalence at $\uparrow_k^m A_k$ that

$$\eta\ ;\ \cdot \vDash \mathcal{F}_1, \mathsf{thread}(y, x.\mathsf{shift}(y)) \sim_{\mathcal{E}} \mathcal{F}_2, \mathsf{thread}(y, x.\mathsf{shift}(y)) :: \Gamma, (y : A_k),$$

which is sufficient to give our result.

If neither k nor m is in $\mathcal{E}$, then we can write $\mathcal{F}_i = \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i^3$ such that:

1. $\mathcal{F}_i^2 = \mathcal{F}_i|_C$
2. $\eta\ ;\ \cdot \vdash \mathcal{F}_i^1, \mathcal{F}_i^2 \sim \mathcal{F}_i^1, \mathcal{F}_i^2 :: (x : {\uparrow}_k^m A_k)$.
3. $\eta\ ;\ \cdot \vdash \mathcal{F}_1^2, \mathcal{F}_1^3 \sim_{\mathcal{E}} \mathcal{F}_2^2, \mathcal{F}_2^3 :: \Gamma$.

The definition of $\sim$ at ${\uparrow}_k^m A_k$ tells us that

$$\eta\ ;\ \cdot \vDash \mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) \sim \mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) :: (y : A_k).$$

As such, $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) \mapsto^* $ some final $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i'$ for which

$$\eta\ ;\ \cdot \vdash \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i' \sim \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i' :: (y : A_k).$$

Let $\mathcal{G}_i^1 = \mathcal{F}_i^1, \mathcal{F}_i'|_{\neg C}$ and $\mathcal{G}_i^2 = \mathcal{F}_i^2, \mathcal{F}_i'|_C$. By configuration extension (Lemma 16), (adapted for $\sim_{\mathcal{E}}$), we get that $\eta\ ;\ \cdot \vdash \mathcal{G}_1^2, \mathcal{F}_1^3 \sim \mathcal{G}_2^2, \mathcal{F}_2^3 :: \Gamma$, and, using that $k \notin \mathcal{E}$ and applying the definition of equivalence with respect to $\mathcal{E}$ at A_k, we get that

$$\eta\ ;\ \cdot \vdash \mathcal{G}_1^1, \mathcal{G}_1^2, \mathcal{F}_1^3 \sim \mathcal{G}_2^1, \mathcal{G}_2^2, \mathcal{F}_2^3 :: \Gamma, (y : A_k).$$

Now, we observe that since $\mathcal{F}_i^1, \mathcal{F}_i^2, \mathsf{thread}(y, x.\mathsf{shift}(y)) \mapsto^* \mathcal{F}_i^1, \mathcal{F}_i^2, \mathcal{F}_i'$, we also have that $\mathcal{F}_i, \mathsf{thread}(y, x.\mathsf{shift}(y)) \mapsto^* \mathcal{F}_i, \mathcal{F}_i' = \mathcal{G}_i^1, \mathcal{G}_i^2, \mathcal{F}_i^3$, and closure under converse reduction completes this case. □

We conjecture (but again do not develop fully, because it is out of scope), that this definition of equivalence corresponds to a form of observational equivalence where configurations can only be observed (extensionally) at modes in $\mathcal{E}$.

Chapter 6

Conclusion

We set out to support the following claim:

> **Thesis Statement:** *Adjoint logic provides a suitably general framework for combining diverse components of deductive systems, not just in pure logic, but also when applied to both the specification of and reasoning about programming languages.*

In Chapter 2, we defined adjoint logic and the key principle of independence, and provided some first evidence with our three calculi ADJ^I, ADJ^E, and ADJ^F, each of which, despite its generality, can be reasoned about generically to prove the standard results of cut elimination and identity expansion. The relations between these calculi also give us several ways to present logics that can be represented in the adjoint framework, as well as a generic focusing theorem, which can aid in proof search. All this supports that adjoint logic is not just general, but "suitably general", providing enough structure to get useful results, while still allowing us to model a wide range of behavior. Several examples in this section also illustrate that we can model a variety of common logics within the adjoint framework, but we are not limited to only these examples.

For the specification of programming languages, we turn to Chapter 4, where, based on a semi-axiomatic presentation of ADJ^I, we defined two programming languages (or, technically, language frameworks, dependent on the choice of mode structure — the uniformity of the definition of the language in modes means that the two are roughly equivalent for now). At a base level, these languages capture two different forms of asynchronous concurrent computation, via message-passing, or via shared-memory (where synchronous communication makes less sense, in any case), but the use of different modes allows us to also handle communication behaviors such as multicast and cancellation in a logically-grounded manner. As for the proof theory, here we are again able to prove results generically over any instantiation of these languages, giving standard type-safety results as well as a form of confluence, providing further support to the suitability and generality of the adjoint approach in this different domain. We also see here that these concurrent languages are not entirely separate from sequential computation, with sequentiality being recoverable at least for the shared-memory language via scheduling, or via the addition of some new language constructs for blocking, and speculate that we can make use of this, along with the separation between modes, to work with a mixed concurrent-sequential language where concurrent computation is only allowable at certain modes, giving one example of the range of computational behavior that independence between modes allows us to capture.

Finally, in Chapter 5, we address reasoning about programs and programming languages.

Here, we explored what it means for programs to be equivalent in the language we have defined, providing an understanding of equivalence in a setting where some, but not all data is reusable. As with the other sections, we find again that the adjoint framework provides enough structure that we can prove results such as parametricity. We also examine the idea of having different equivalences at different modes, just as in the context of programming languages we speculated about what could be accomplished if different modes have different semantics, or allow different programs to be written, or in the context of proof theory, we allow modes to have different structural properties, or to restrict what connectives are allowed. The results of this are preliminary, but provide support for the idea that mode-dependent equivalences are possible to define coherently in several different ways, with the results depending on how we choose to combine them — a fruitful area for future study.

In each of these applications, we see that the adjoint framework is general, suitable for modeling a wide range of behavior — varied logics and combinations thereof, programming languages where the allowable computations vary by mode (even just in the basic setting where the language is defined uniformly, some of this is given by the structural properties of modes), and equivalences both uniform and mode-dependent. We also see that this generality is not excessive, however, and that we are able to establish useful results generically for these varied instances. Because of this, the different logics, programming languages, or equivalences defined within the adjoint framework can also be combined cleanly, allowing us to work in a programming language that mixes features of two base languages, for instance, while enjoying the benefits of a general type-safety result.

This work is relatively general and foundational, and as such, there are many different directions that future work could take. We will examine here several of those directions, separated by which portion of the work they correspond to: proof theory and logic, programming language specification, and equivalence or reasoning about programs. This is by no means meant to be an exhaustive list, but serves to illustrate some of the more interesting possible directions.

6.1 Proof Theory and Logic

In the direction of proof theory and logic, there are two main directions for future work — first, extending adjoint logic to be able to model a wider range of logics, and second, building towards a dependent adjoint type theory, which could serve as the basis for an adjoint logical framework, generalizing the linear and concurrent logical frameworks LLF [15] and CLF [17, 96, 108].

Currently, adjoint logic as presented in this work only models intuitionistic logics, and independence imposes some further restrictions — for instance, any monad that we model is a *strong* monad, which prevents us from modeling the modal possibility $\diamond$ of S4. It would be interesting to explore whether we can find modifications that enable adjoint logic to model some additional systems, without losing too much in terms of the ability to prove results generically. For a concrete example, *ecumenical logic* [79, 82], which consists of a classical and an intuitionistic portion combined into the same logic, has a distinctly similar feel to modal logics, and to adjoint logic, and there may be a way to handle this. A study of ecumenical logic would likely also give a better understanding of how to define a classical form of adjoint logic, which could then be used to model a classical linear/non-linear logic, for instance. Modal logics other than S4 also present an interesting direction for future exploration — in particular, it would be interesting to explore whether various sorts of temporal logics can be expressed in an adjoint fashion, with modes representing time steps, for instance.

The goal of building a dependent adjoint type theory is fairly clear, being to extend adjoint logic with dependent types (again, in a way that does not lose the benefits of being able to prove results generically), although how to do it is less clear — a notion of equivalence of terms is needed in order to reason about equivalence of dependent types, and our work on equivalence may be a starting point for that, but it seems that further development would be necessary to get a suitable form of equivalence, and once this is done, there remain further questions, such as whether a type dependent on a linear term *consumes* that term or not, and how this should be handled.

6.2 Programming Language Specification

A first question in this direction is to what extent we can define mixed-mode languages, and what properties they enjoy. For instance, an ideal result would say that given two languages based on adjoint logic with disjoint sets of modes, we can define a combined language on the union of their sets of modes, which restricts to each base language when looking only at modes that given language uses. We would also like to ensure that good properties of these base languages can be "lifted" to the overall language — for instance, if each base language has progress, then so does the overall language, and similarly for preservation, confluence, and other such properties. One immediate question, however, is what the order relation should be between two modes from different languages — can we arbitrarily put any language on top of any other, or is there some condition on the semantics of the languages that restricts this further, as with the restriction of structural properties of modes? Once this is addressed, we would also need to determine what the interface between two languages with potentially very different semantics looks like, and what the semantics are for programs that cross over this boundary.

A different aspect of programming languages based on adjoint logic involves providing a more "high-level" version of the language. The two systems we present in Chapter 4 are reminiscent of low-level programming, dealing quite a bit with fine details of memory, sequences of messages, process spawning, and so on. A system with higher-level syntax would be easier to work with and use, and could be translated to this low-level syntax for reasoning or optimization if necessary.

6.3 Program Reasoning

Program reasoning again provides several different directions for future work. Most directly related to adjoint logic, we have yet to find a full characterization of when distinct equivalences can be combined. The examples that we see suggest that with sufficient isolation between modes, there are no restrictions, but that if we want equivalence at a mode k to be able to depend on equivalence of terms at higher mode m, just as (following independence) a term at mode k can depend on terms at mode m, we may need some constraints, such as requiring that proof-irrelevant modes are strictly below extensional modes, as seen in Chapter 5. A first conjecture might be that if $k \leq m$, then equivalence at k must be coarser than equivalence at mode m, able to distinguish fewer processes.

Another direction based on this work, although not so relevant to adjoint logic, would be to further explore the generic definition of observational equivalence proposed in Chapter 5, and to see if it is feasible to define a similarly generic logical equivalence which agrees with observational equivalence, rather than restricting to the extensional case, as we do here.

Bibliography

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming*, pages 69–83. Springer, 2006. 5.1

[2] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992. 2, 2.2, 2.3

[3] Robert Atkey. Observed communication semantics for classical processes. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*, pages 56–82. Springer, 2017. 5.5

[4] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. In *International Conference on Functional Programming (ICFP)*, pages 37:1–37:29. ACM, September 2017. Extended version available as Technical Report CMU-CS-17-106R, June 2017. 4.1

[5] Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency. *arXiv preprint arXiv:2309.00192*, 2023. 5.1, 5.5

[6] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996. 2, 2

[7] Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CLS'94)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge. 1, 2, 2, 3

[8] Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. 2.1.1

[9] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Information and computation*, 155(1-2):3–63, 1999. 5.1

[10] G. E. Blelloch and M. Reid-Miller. Pipeling with futures. *Theory of Computing Systems*, 32:213–239, 1999. 4, 4.4.2

[11] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 365(1–3):227–270, 2007. 1

[12] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Pro-*

ceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010), pages 222–236, Paris, France, August 2010. Springer LNCS 6269. 1, 3, 3.1, 4, 4.1

[13] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792. 5.5, 5.5.1

[14] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types. 4, 4.1

[15] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and computation*, 179(1):19–75, 2002. 1, 6.1

[16] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009. 4.2.4

[17] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003. 1, 6.1

[18] Kaustuv Chaudhuri. Classical and intuitionistic subexponential logics are equally expressive. In *Computer Science Logic*, pages 185–199. Springer LNCS 6247, August 2010. 1, 2

[19] Karl Crary. Higher-order representation of substructural logics. In P.Hudak and S.Weirich, editors, *Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)*, pages 131–142, Baltimore, Maryland, September 2010. ACM. 2.1.1

[20] Ornela Dardha and Jorge A Pérez. Comparing type systems for deadlock freedom. *Journal of Logical and Algebraic Methods in Programming*, 124:100717, 2022. 4.3.3

[21] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015. 2.1.1

[22] Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. Session logical relations for noninterference. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2021. 5.5

[23] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979. 5.5, 5.5

[24] Joëlle Despeyroux, Carlos Olarte, and Elaine Pimentel. Hybrid and subexponential linear logics. *Electronic Notes in Theoretical Computer Science*, 332:95–111, 2017. 1

[25] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Annual Conference on Computer Science Logic (CSL*

2012), pages 228–242, Fontainebleau, France, September 2012. LIPIcs 16. 4.2.1

[26] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. 1.1, 1.3, 3, 4.3.1, 4.4.1, 5.5

[27] Derek Dreyer. The type soundness theorem that you really want to prove (and now you can). The 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018), 2018. 5.5.4

[28] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 71–80. IEEE, 2009. 5.1

[29] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7, 2011. 5.1

[30] M. Fairtlough and M.V. Mendler. An intuitionistic modal logic with application to the formal verification of hardware. In L. Pacholski and J. Tiuryn, editors, *Proceedings of the 8th Workshop on Computer Science Logic (CSL'94)*, pages 354–368, Kazimierz, Poland, September 1994. Springer-Verlag LNCS 933. 2

[31] M. Fairtlough and M.V. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997. 1, 5

[32] Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, 27(04):289–296, 1978. 4.4, 4.4.2

[33] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005. 4.2.6

[34] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969. 2.1.1, 4

[35] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 1, 2, 2

[36] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250. 2

[37] JY Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, PhD thesis, Université Paris VII, 1972. 5.1.1, 5.5, 5.5.1

[38] Marco Giunti and Vasco T Vasconcelos. A linear account of session types in the pi calculus. In *International Conference on Concurrency Theory*, pages 432–446. Springer, 2010. 4.1

[39] Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985. 1.1, 4.4, 4.4.2

[40] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016. 2, 5.4, 5.5

[41] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 4.1

[42] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993. 4

[43] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512. 4

[44] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998. 4.1

[45] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St. Andrews, Scotland, 1998. 2.3

[46] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–141, 2001. 4.1

[47] Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. Adjoint natural deduction (extended version). *arXiv preprint arXiv:2402.01428*, 2024. 5.1, 13

[48] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017. 2

[49] Max Kanovich, Stepan Kuznetsov, Vivek Nigam, and Andre Scedrov. Subexponentials in non-commutative linear logic. *Mathematical Structures in Computer Science*, 29(8): 1217–1249, 2019. 1, 2, 2

[50] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR 2006–Concurrency Theory: 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006. Proceedings 17*, pages 233–247. Springer, 2006. 4.3.3

[51] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In H.-J. Boehm and G. Steele, editors, *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 358–371, St. Petersburg Beach, Florida, USA, January 1996. ACM. 4.1

[52] Neelakantan R Krishnaswami and Derek Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In *Computer Science Logic 2013 (CSL 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. 5.3

[53] James R Larus. *Restructuring symbolic programs for concurrent execution on multiprocessors*. PhD thesis, University of California at Berkeley, 1989. 4.4, 4.4.2

[54] Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001. 2.3, 5.4

[55] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006. 2.3, 5.4

[56] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *International Workshop on Computer Science Logic*, pages 451–465. Springer, 2007. 2.3

[57] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009. 2.3

[58] Daniel R. Licata and Michael Shulman. Adjoint logic with a 2-category of modes. In *International Symposium on Logical Foundations of Computer Science (LFCS)*, pages 219–235. Springer LNCS 9537, January 2016. 1, 2, 2

[59] Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)*, pages 25:1–25:22, Oxford, UK, September 2017. LIPIcs. 1, 2, 2, 4

[60] Robin Milner. *An algebraic definition of simulation between programs*. Citeseer, 1971. 5.4

[61] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978. 4.3.3

[62] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag LNCS 92, 1980. 4.1

[63] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. 4.1

[64] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *International Colloquium on Automata, Languages, and Programming*, pages 685–695. Springer, 1992. 5.4

[65] Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, 2001. 4

[66] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *Proceedings of the 11th International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 129–140, Coimbra, Portugal, sep 2009. ACM. 1, 2

[67] Vivek Nigam, Elaine Pimentel, and Giselle Reis. Specifying proof systems in linear logic with subexponentials. *Electronic Notes in Theoretical Computer Science*, 269:109–123, 2011. 1, 2

[68] Peter O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007. 1

[69] Peter W. O'Hearn. Resources, concurrency, and local reasoning. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, pages 49–67, London, England, August 2004. Springer LNCS. 1

[70] 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), pages 1–10, 2014. 4.3.3

[71] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science: 5th GI-Conference Karlsruhe, March 23–25, 1981*, pages 167–183. Springer, 1981. 5.4

[72] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Termination in session-based concurrency via linear logical relations. In H. Seidl, editor, *22nd European Symposium on Programming*, ESOP'12, pages 539–558, Tallinn, Estonia, March 2012. Springer LNCS 7211. 5.5

[73] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014. 5.5

[74] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. Technical Report CMU-CS-01-116, Department of Computer Science, Carnegie Mellon University, April 2001. 4

[75] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999. 1, 2, 4, 5

[76] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23 (7):199–208, 1988. 2.1.1

[77] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England, April 2015. Springer LNCS 9034. Invited talk. 1, 4

[78] Frank Pfenning and Carsten Schuermann. Twelf user's guide. Technical report, version 1.2. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998. URL `https://www.cs.cmu.edu/~twelf/guide-1-4/twelf.pdf`. 2.1.1

[79] Elaine Pimentel, Luiz Carlos Pereira, and Valeria de Paiva. An ecumenical notion of entailment. *Synthese*, 198(Suppl 22):5391–5413, 2021. 6.1

[80] Andrew M Pitts. Relational properties of domains. *Information and computation*, 127(2): 66–90, 1996. 5.1

[81] Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973. 5.1.1, 5.5

[82] Dag Prawitz. Classical versus intuitionistic logic. *Why is this a Proof? Festschrift for Luiz Carlos Pereira*, pages 15–32, 2015. 6.1

[83] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software (PLACES)*, pages 60–79, Prague,

Czech Republic, April 2019. EPTCS 291. 1.1

[84] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming*, 120:100637, 2021. 1.1

[85] Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 32:e6, 2022. 1.1

[86] Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf. 1.1

[87] Zesen Qian, GA Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–31, 2021. 4.1

[88] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf. 1, 2, 4

[89] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983. 5.1.1, 5.5, 5.5.1, 5.5.6

[90] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Computer Society. 1

[91] Jacques Riguet. Relations binaires, fermetures, correspondances de galois. *Bulletin de la société mathématique de France*, 76:114–155, 1948. 5.3, 5, 25, 26

[92] Jacques Riguet. Quelques propriétés des relations difonctionnelles. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 230:1999–2000, 1950. 27

[93] Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021. 4.1

[94] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001. 4.1

[95] Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):374–399, 2023. 2.1.1

[96] Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195. 1, 6.1

[97] Peter Selinger and Benoıt Valiron. Quantum lambda calculus. *Semantic techniques in quantum computation*, pages 135–172, 2009. 1

[98] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014. 2.3

[99] Siva Somayyajula and Frank Pfenning. Type-based termination for futures. In *7th International Conference on Formal Structures for Computation and Deduction*, 2022. 5.1

[100] Siva Somayyajula and Frank Pfenning. Dependent type refinements for futures. *Electronic Notes in Theoretical Informatics and Computer Science*, 3, 2023. 5.1

[101] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985. 5.1.1, 5.5

[102] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal Of Symbolic Logic*, 32:198–212, 1967. 5.1.1, 5.5

[103] The Coq Development Team. The Coq reference manual – release 8.19.0. `https://coq.inria.fr/doc/V8.19.0/refman`, 2024. 2.1.1

[104] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 350–369, Rome, Italy, March 2013. Springer LNCS 7792. 1, 4

[105] Rob J Van Glabbeek and W Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, 1996. 5.4

[106] André Van Tonder. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004. 1

[107] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52, 1984. 4.4, 4.4.2

[108] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003. 1, 6.1

[109] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust. 2019. 2