Notes on Interaction Nets and Linear Logic: Theory and Applications

Eric Schmid

March 13, 2025

1 Introduction

Linear Logic, introduced by J.-Y. Girard in 1987, is a substructural logic that treats logical reasoning as a resource-sensitive process. Unlike classical logic, linear logic controls contraction and weakening, ensuring that assumptions are used exactly once unless explicitly allowed. This yields a finer-grained "proofs-as-programs" correspondence where proofs correspond to programs and cut-elimination corresponds to program execution. Interaction Nets, proposed by Y. Lafont, are a graph-rewriting model of computation inspired by the proof structures of linear logic. In fact, interaction nets generalize Girard's proof nets of linear logic (Lafont, Interaction Nets, 1989). They provide a *visual and parallel* paradigm for computation, emphasizing local interaction (rewriting) between *agents* (graph nodes) and a built-in notion of deterministic concurrency.

In these notes, we develop the theory of linear logic and interaction nets in a formal, rigorous manner. We begin by presenting the syntax and sequent calculus of linear logic with full inference rules, highlighting the resource-sensitive nature of proofs. We then introduce proof nets, a graphical representation of linear logic proofs that abstracts away bureaucratic derivation details while preserving correctness. We give formal correctness criteria for proof nets and prove the fundamental Sequentialization Theorem, which ensures that any valid proof net corresponds to a sequent proof. Cut-elimination – the process of removing intermediate assertions (cuts) from proofs – is studied in depth. We provide a detailed proof of cut-elimination for linear logic and analyze its properties, including strong normalization (every reduction sequence terminates) and (strong) confluence (independence of reduction order for the result) (Bierman, On Intuitionistic Linear Logic, 1994). These properties guarantee that logical proofs have a well-defined normal form and that proof normalization can be seen as a deterministic computation.

Building on the proof net concept, we formally define interaction nets as an autonomous graph-rewriting system. We give precise definitions of agents, ports, nets, and interaction rules, and we state the critical properties (linearity, binary interaction, absence of ambiguity, etc.) that interaction rules must satisfy. Key confluence results for interaction nets are presented, showing that even though many rewrites can happen in parallel, the final outcome of computation is unique (up to isomorphism). We also introduce a type system for interaction nets, inspired by linear logic's type discipline, to ensure deadlock-free and deterministic parallel computation. We prove formally that well-typed (so-called *simple*) nets cannot get stuck in deadlocked configurations (cycles of interacting agents) – every well-typed net reduces to a normal form without encountering a "vicious circle" of active pairs. In particular, we show that if a net is well-typed, then reduction preserves

typability (an invariance property), and any irreducible well-typed net must have a proper interface (open ports) rather than an internal deadlock.

Finally, we explore additional formal topics such as categorical semantics of linear logic and its proofs. We define the categorical structures (such as *-autonomous categories and linear categories) that provide sound models of linear logic, and we explain how proof nets (and hence interaction nets via their correspondence to proofs) can be interpreted as morphisms in these categories. This provides a high-level semantic view of why cut-elimination is confluent: different reduction paths correspond to the same composite morphism in any model, ensuring uniqueness of result. We also discuss computational interpretations of linear logic and interaction nets, connecting proof normalization to computation. For example, we outline how interaction nets can implement lambda-calculus reduction with sharing (Lamping's algorithm) and how restrictions of linear logic (like Light Linear Logic) capture complexity classes. Throughout, the emphasis is on formal development: the aim is a self-contained, technically detailed exposition following the style of Braüner's and Bierman's works on proof theory and linear logic.

—

2 Linear Logic – Syntax and Sequent Calculus

Linear logic is built on the idea that logical assumptions are *resources* that cannot be copied or discarded arbitrarily. In this section we define the syntax of propositional linear logic and present its sequent calculus proof system in full detail. We consider Classical Linear Logic (CLL), in which every formula has a dual, as well as the restricted Intuitionistic Linear Logic (ILL) as a subsystem. We give inference rules for all of linear logic's connectives – multiplicatives, additives, and exponentials – along with the identity and cut principles. The presentation will be formal, using sequents and inference rule schemas. We assume a countable set of propositional atoms (variables) ranged over by letters like $P, Q, R, \ldots$.

2.1 Formulas of Linear Logic

Definition: *The set of formulas of propositional linear logic* is defined inductively as follows: - Every propositional atom P is a formula. - If A is a formula, then $A^{\perp}$ (the *linear negation* of A) is a formula. Intuitively, $A^{\perp}$ is the *linear dual* of A. - If A and B are formulas, then so are each of the following, representing the connectives of linear logic: - Multiplicative Conjunction (Tensor): $A \otimes B$ - Multiplicative Disjunction (Par): $A \parr B$ (sometimes written as $A \,\&\!\&\, B$; we use $\parr$) - Additive Conjunction (With): $A \,\&\, B$ - Additive Disjunction (Plus): $A \oplus B$ - Exponential (of-course): $!A$ (read "bang A") - Exponential (why-not): $?A$ (read "quest A", the dual of $!A$) - Additionally, linear logic includes unit formulas for the connectives: - Unit for $\otimes$: 1 (the multiplicative truth, true with no resources) - Unit for $\parr$: $\perp$ (multiplicative falsehood) - Unit for &: $\top$ (additive truth) - Unit for $\oplus$: 0 (additive falsehood, an absurdity)

Linear negation $(\cdot)^{\perp}$ is an involution on formulas satisfying: $(A^{\perp})^{\perp} = A$. It De Morgan dualizes connectives: $(A \otimes B)^{\perp} = A^{\perp} \parr B^{\perp}$, $(A \parr B)^{\perp} = A^{\perp} \otimes B^{\perp}$; and similarly $(A \,\&\, B)^{\perp} = A^{\perp} \oplus B^{\perp}$, $(A \oplus B)^{\perp} = A^{\perp} \,\&\, B^{\perp}$. The units are duals: $1^{\perp} = \perp$, $\top^{\perp} = 0$ (and vice versa). Intuitively, $A^{\perp}$ represents the "resource complement" of A. In classical linear logic, every formula A has a negation $A^{\perp}$ and the logic is *self-dual*. Intuitionistic linear logic can be seen by restricting to formulas in *negation normal form* (e.g. considering sequents with a single formula on the right side). We will focus on the general (classical) system, as intuitionistic linear logic can be embedded within it.

2.2 Sequent Calculus for Linear Logic

A *sequent* is a judgment of the form:

$$\Gamma \vdash \Delta,$$

where Γ and Δ are (multi)sets of formulas. Intuitively, Γ is a multiset of *assumptions/premises* available (each usable at most once), and Δ is a multiset of *conclusions* (goals). In classical linear logic, multiple conclusions are allowed (and in fact Δ is often taken to be either empty or a single formula's negation in the one-sided sequent presentation). However, one can equivalently restrict to a one-sided sequent calculus where all formulas are considered on one side as either positive or negative formulas. For our purposes, we will use a two-sided sequent calculus where linear negation connects left and right contexts. Intuitionistic linear logic is a subsystem where at most one formula appears on the right of the turnstile.

The sequent calculus rules for linear logic are given below. We organize them by the structural/identity rules, then the logical rules for each connective. In each rule schema, $\Gamma, \Delta, \Theta, \ldots$ denote (multi)sets of formulas. Comma in the contexts denotes multiset union (i.e. availability of all those formulas). We write $A^\perp$ for the negation of A.

- Identity / Axiom: An atomic formula and its negation can be introduced on opposite sides with no premises. This is the linear variant of the identity law:

$$\frac{}{\vdash A,\ A^\perp}\ \mathsf{Ax}$$

This rule means that from no premises we can derive the basic sequent $A \vdash A$ (often written with A on left and A on right as $A \vdash A$, which is equivalent to $\vdash A, A^\perp$ by moving one A to the right side as $A^\perp$). It licenses the assumption that A and $A^\perp$ are complementary resources that cancel each other.

- Cut: The cut rule allows us to compose two proofs by "cutting" on a formula A, consuming it as a resource that connects the proofs. It introduces a *hidden* use of formula A which will later be eliminated by cut-elimination. The rule schema:

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma', A^\perp \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathsf{Cut}$$

In words: if $\Gamma \vdash \Delta, A$ is derivable (i.e. A on the right) and A can also be proved from some assumptions Γ' as $\Gamma', A \vdash \Delta'$, then we can "cut" A, removing it from the conclusion of the first and the assumption of the second, to conclude $\Gamma, \Gamma' \vdash \Delta, \Delta'$. The formula A is the *cut formula*. In linear logic, cut is *admissible* (it can be eliminated). We will prove formally in section 4 that any proof using cut can be transformed into a cut-free proof, which is essential for consistency and computational interpretation.

- Exchange: (Structural rule) Linear logic sequents treat contexts Γ and Δ as multisets, so an *Exchange* rule (commuting formulas in context) is admissible implicitly. We assume that the order of formulas in Γ or Δ does not matter (they are resources without inherent sequence). We will not write an explicit rule for Exchange, treating context as unordered.

- Weakening and Contraction: A key aspect of linear logic is that *unrestricted* weakening and contraction are not allowed in general. You cannot freely discard or duplicate assumptions. However, *controlled* versions of these rules are allowed *only* in the presence of the exponential modality ! (or dually ?). We list the general rules for weakening and contraction under exponentials below, after introducing ! and ?. For now, note that without exponentials, linear logic does not

allow one to drop a formula from Γ or duplicate it arbitrarily. Each formula must be used exactly once unless it is marked as reusable (with a !).

Now we present the logical connective rules. Each connective in linear logic has *two* inference rules: one for introducing that connective on the left side of a sequent (in the context of assumptions), and one for introducing it on the right side (in the context of conclusions). In classical terminology, these are sometimes called left-rule and right-rule for the connective (except exponentials, which we handle specially). Because of linear negation duality, the left rule for a connective is often equivalent to the right rule for its De Morgan dual, and vice versa. We give each rule explicitly for completeness:

- Tensor ($\otimes$) rules: $\otimes$ is a binary *multiplicative conjunction*. It represents using two resources together. - *Right rule ($\otimes R$):* To prove a sequent with $A \otimes B$ on the right (as a goal), we must *split* the context of assumptions between proving A and proving B. Formally:

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma' \vdash \Delta, A \otimes B, \Delta'} \otimes R$$

Here the context Γ, Γ' is partitioned into two parts: one used to derive A and one used to derive B. Likewise the conclusions Δ, Δ' may be partitioned (though typically in linear logic one takes conclusions only on one side in a one-sided calculus; in this two-sided presentation, we allow conclusions on both sides for classicality). Intuitively, $\otimes R$ says: to show $A \otimes B$ as available output, we need to independently show A and B from disjoint subsets of the input resources. - *Left rule ($\otimes L$):* To use $A \otimes B$ as an assumption (on the left), we can *unpack* it into its two components, since having $A \otimes B$ as a single resource means we have both A and B together. The rule:

$$\frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, A \otimes B, \Gamma' \vdash \Delta} \otimes L$$

That is, if $A \otimes B$ is available as a resource (in addition to Γ, Γ'), then we may assume A and B separately as resources. Note this rule does not split the context – it *adds* the components A, B into the context in place of $A \otimes B$. In linear logic, this is sound because $A \otimes B$ explicitly represents having both A and B. - *Unit (1) rules:* The multiplicative unit 1 represents "no resource" (the identity for $\otimes$). - Right unit rule ($1R$): From nothing on the right, we can conclude 1 (if there are no other conclusions). Equivalently, 1 is true when no resources are required. Formally:

$$\frac{\Gamma \vdash \ (\text{nothing})}{\Gamma \vdash 1} \ 1R.$$

This rule requires that the conclusion multiset Δ was empty except for the 1. In other words, 1 can be derived if and only if *there is nothing else to prove*, i.e. the sequent's right side had exactly 1. In a one-sided presentation, 1 is usually an axiom on the right with no premises. - Left unit rule ($1L$): There is no need for an assumption rule for 1 because having 1 as a resource means having "no information" – which should not affect the ability to prove a conclusion. Indeed, linear logic usually omits $1L$ or treats 1 as a neutral element in the context (the sequent $\Gamma, 1 \vdash \Delta$ is equivalent to $\Gamma \vdash \Delta$ by a logical axiom). In sequent-calculus style, one may simply allow an implicit structural rule that removes 1 from the context (since 1 on the left offers no resource). We will not explicitly include a $1L$ rule; instead we consider 1 in Γ as inert.

- Par ($\parr$) rules: $\parr$ (par) is the multiplicative disjunction (linear dual of $\otimes$). It represents a kind of *co * or parallel combination where a consumer can choose one of two resources. The rules for

$\parr$ are dual to those of $\otimes$. - *Right rule ($\parr R$):* On the right, $A \parr B$ means we have a resource that can be treated as either A or B (dually, a goal $A \parr B$ means to achieve either A or B). Because $A \parr B$ is the dual of $A^{\perp} \otimes B^{\perp}$, the $\parr R$ rule corresponds to the $\otimes L$ pattern. We typically present $\parr$ by using linear negation duality: one can omit explicit $\parr R$ if one has $\otimes L$ and the ability to move formulas across the turnstile via negation. For completeness, in two-sided form the $\parr R$ rule would *combine* two conclusions:

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B}\ \parr R$$

This rule is seldom used directly; it says if you can conclude both A and B (in the same context Γ), then you can conclude $A \parr B$. But note that concluding both A and B in linear logic usually means you had to split resources – which $\parr R$ as stated doesn't enforce. A more standard presentation avoids this by treating $\parr$ as a defined connective via negation of $\otimes$. We will treat $\parr$ mostly by duality to $\otimes$. - *Left rule ($\parr L$):* On the left, $A \parr B$ as a resource means a choice between A or B is available (since it is the dual of "having both"). The rule is dual to $\otimes R$, hence context-splitting on the left:

$$\frac{\Gamma, A, \Theta \vdash \Delta \qquad \Gamma', B, \Theta \vdash \Delta}{\Gamma, \Gamma', A \parr B, \Theta \vdash \Delta}\ \parr L$$

This says: to use $A \parr B$ to prove Δ, we must show that Δ can follow if we assume A (in one scenario) and also if we assume B (in another scenario), with the rest of the context Θ shared and Γ vs Γ' partitioning any other assumptions. Intuitively, an $A \parr B$ resource means an adversary or environment could provide either A or B, so we must be ready to handle both cases. - *Unit ($\bot$) rules:* $\bot$ is the unit of $\parr$. - Left unit rule ($\bot L$): $\bot$ on the left represents an absurd resource (falsehood) – having it should allow anything to be proved (ex falso quodlibet in linear form). Thus:

$$\frac{\text{(no premises)}}{\Gamma, \bot \vdash \Delta}\ \bot L.$$

If $\bot$ appears in the context, the sequent is immediately provable (no matter Δ), since $\bot$ means a contradiction or unlimited resource from nothing. - Right unit rule ($\bot R$): Usually omitted, as $\bot$ on the right cannot be achieved from any premises (it is false). In a two-sided system one might say $\vdash \bot$ is not derivable except by trivial contradiction. We will not include an explicit $\bot R$ (it has no introduction rule on the right because $\bot$ is never true).

- With ($\&$) rules: $A \,\&\, B$ is the *additive conjunction* (often read "A and B" in a context where at most one of them will be used, unlike $\otimes$ where both are used). It represents having *both choices available* but only one will actually be chosen – it is like a record or tuple that contains both A and B as possible information. Dually, it means the proponent must provide both A and B. The rules are additive (no splitting of resources; the choice is on the context side): - *Right rules ($\&R$):* To conclude $A \,\&\, B$ (as a goal), we need to conclude *both* A and B separately, using the same resources (context) for each, because $A \,\&\, B$ promises that both A and B are true (the evaluator must produce both). Formally:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&R$$

We require two subproofs with identical context Γ, one deriving A and one deriving B. No context splitting occurs (this is *additive* introduction). - *Left rules ($\&L1$ / $\&L2$):* To use $A \,\&\, B$ on the left (as an assumption), one has a *choice* of which part to use, because $A \,\&\, B$ means we have

an agent that can supply either A or B. There are two left introduction rules, selecting the left conjunct or the right conjunct:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \,\&\, B \vdash \Delta}\ \&L_1 \qquad\qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \,\&\, B \vdash \Delta}\ \&L_2.$$

In each case, having $A \& B$ as a resource permits us to assume either A or B (the other is not used). This reflects that the environment providing the & resource will choose one of the two options to actually make available. - *Unit ($\top$) rules:* $\top$ is the unit of &. It represents "truth" that carries no information (neutral for with). - Right unit rule ($\top R$): To conclude $\top$, we need no premises; $\top$ is always true (like a tautology) but note that *using resources* to prove $\top$ consumes them for nothing, so usually one only concludes $\top$ when there are no other goals and perhaps unused assumptions. In sequent form:

$$\frac{}{\Gamma \vdash \top}\ \top R$$

(No premises; any Γ is actually ignored because we haven't used those assumptions, which linear logic normally wouldn't allow. Typically one requires Γ empty here; or treat unused assumptions as contradictions unless they are marked with !. For simplicity, we consider Γ must be empty for $\top R$ to apply, meaning $\top$ can be concluded only in an initial state with no resources needed.) - Left unit rule ($\top L$): There is no left rule for $\top$ because having $\top$ as a resource does nothing (it provides no actual content). Indeed, $\top$ on the left is inert (much like 1 on left), and including an explicit $\top L$ that simply removes $\top$ is redundant. We omit $\top L$ or treat it implicitly as a structural simplification.

- Plus ($\oplus$) rules: $A \oplus B$ is the *additive disjunction* (linear "or"). It represents a choice between A or B that has already been made by the provider (as opposed to $\parr$ where the *consumer* chooses). The rules are dual to &: - *Right rules ($\oplus R1$, $\oplus R2$):* To conclude $A \oplus B$ as a goal, we have to choose which disjunct to prove, because $A \oplus B$ means *either* A *or* B holds, and we as proof constructors must pick one branch. There are two introduction rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus R_1 \qquad\qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus R_2.$$

Each says: if we can prove A (resp. B) from Γ, then we can conclude $A \oplus B$, committing to the first (resp. second) disjunct. Note that the same context Γ is used; no resource splitting. This is additive choice on the proving side. - *Left rule ($\oplus L$):* To use $A \oplus B$ as an assumption, the environment or context has provided a value that is either A or B. But *we* (the proof) don't get to choose which; we must handle both possibilities. Thus the left rule requires two subcases (like $\&R$):

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\ \oplus L.$$

The same resource context Γ is available in both subproofs; we must be able to derive Δ assuming A and also assuming B. Intuitively, since we don't know which disjunct the opponent chose, our goal Δ must follow in either case. - *Unit (0) rules:* 0 is the unit of $\oplus$ (additive falsehood). - Right unit rule ($0R$): There is no way to conclude 0 on the right; 0 is an impossibility (no disjunct chosen) and thus has no introduction. So no $0R$ rule (or we can say it's never provable except from a contradiction). - Left unit rule ($0L$): Dually, 0 on the left is always usable to derive anything

(since the environment has given an absurd choice). In fact, 0 as a resource is akin to having an immediate contradiction at hand. The rule:

$$\frac{}{\Gamma, 0 \vdash \Delta}\ 0L$$

with no premises – if 0 appears in the context, the sequent is trivially valid (similar to $\bot L$). (Again, typically one of $\bot$ or 0 is redundant; here $\bot$ was multiplicative false, 0 is additive false. In classical linear logic both can exist but one can encode one via the other with exponentials if needed.)

- Exponential (! and ?) rules: The exponential modality ! (pronounced "bang") is used to mark formulas that can be used *structurally*, i.e. copied or discarded. It is what allows linear logic to regain some aspects of classical logic by marking certain propositions as *not consuming resources*. The dual $?A$ (pronounced "why not A") marks formulas that can be freely duplicated/dropped on the *output* side. Intuitively, $!A$ means A is a duplicable resource (available without linear restrictions), and $?A$ means A is a repeatable goal/claim. The rules for exponentials introduce the ability to apply weakening or contraction but only to formulas bearing these modalities. We need some auxiliary terminology: a context Γ is called a *!*-context (or *exponential context*) if every formula in Γ is of the form $!B$ for some B. We will use $!\Gamma$ to denote a multiset of formulas each prefixed by !, and $?\Delta$ for conclusions each prefixed by ?. Now the rules: - *Dereliction:* This rule allows us to move between linear and nonlinear worlds by removing a ! on the left or a ? on the right, essentially "using" the fact that a formula is available linearly.

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ \mathsf{Dereliction} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta}\ \mathsf{Codereliction}$$

Dereliction (left) says: if $!A$ is an unrestricted assumption, we can choose to use it once (drop the !) and treat it as a linear assumption A for the purpose of proving Δ. Codereliction (the dual, often not named separately) says: if we need to prove $?A$ on the right, that is equivalent to proving A (using the resource in a linear fashion) since $?A$ is just A that we promise not to duplicate on the proving side. These rules essentially allow treating $!A$ as A or $?A$ as A when needed. - *Promotion:* This crucial rule allows upgrading a linear proof of A into a proof of $!A$, but it requires that all other assumptions used in proving A were already marked ! (i.e. the proof of A did not consume any linear resources). Formally:

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A}\ \mathsf{Promotion} \quad \text{(with the side condition that every formula in } \Gamma \text{ is of the form } !B)$$

The *Promotion* rule allows us to "upgrade" a linear derivation into a nonlinear one. In this rule, Γ is a multiset of assumptions, and the side condition requires that every formula in Γ must be of the form $!B$ (that is, each assumption is already marked as duplicable). Intuitively, this means that if we can derive A using only assumptions that can be freely copied and discarded, then we are permitted to conclude $!A$, declaring that A itself is duplicable. In other words, Promotion acts as a gate: it ensures that only derivations that do not depend on any *linear* (non-duplicable) information can be "promoted" to the exponential level. This controlled introduction of ! is crucial for preserving consistency in the presence of structural rules like weakening and contraction.

- *Weakening:* Now we can formally add weakening for ! (and ?). Weakening means we can discard an unused assumption or conclusion if it is exponential. - Left Weakening:

$$\frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}\ \mathsf{Weakening}_L$$

This says: if we have an unused resource $!A$ in our context, we may drop it (not use it) and the sequent remains derivable. We cannot do this for a linear A (without !) because that would break linear usage. But $!A$ is assumed to be an unlimited resource that can be discarded. - Right Weakening: Dually,

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta}\ \mathsf{Weakening}_R$$

meaning if our goal included $?A$ (which we choose not to prove or need), we can drop it, since $?A$ is something the environment could have provided without cost. Usually $?A$ weakening is not separately needed if we consider sequent contexts symmetrically, but we include it for completeness. - *Contraction:* Contraction allows us to duplicate an assumption or conclusion if it is marked exponential. - Left Contraction:

$$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ \mathsf{Contraction}_L$$

This rule says: if we have two copies of $!A$ in the context, we may merge them into one (equivalently, we are allowed to use one $!A$ assumption to serve two purposes because $!A$ is duplicable). Note we write the rule in reverse (usually one applies contraction from one to many, but as a rule of inference we show that if the conclusion with a single $!A$ is derivable, then the premise with two $!A$ was derivable, allowing us to remove the duplicate). In practice, this permits using the same $!A$ assumption multiple times in the proof. - Right Contraction:

$$\frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta}\ \mathsf{Contraction}_R$$

Dually, if the sequent $\Gamma \vdash ?A, ?A, \Delta$ is derivable (two identical $?A$ conclusions), we can conclude it with just one $?A$ (merging two identical goals into one). This means the proof can supply one A to satisfy two identical demands if those demands are marked ?.

These rules (together with the earlier logical rules) constitute the full sequent calculus for propositional linear logic. In summary, linear logic enjoys a rich collection of connectives each with precise introduction rules, but controls structural rules by allowing them only on formulas marked with exponentials. As a sanity check, if we mark *all* formulas with ! (making everything duplicable) and ignore linear restrictions, the rules collapse into those of classical logic (with contraction and weakening always allowed). Conversely, if we disallow ! entirely, we get the pure linear logic where every assumption must be used exactly once. Intuitionistic linear logic (ILL) is obtained by restricting to at most one formula on the right of $\vdash$ in each rule (which means Δ is either empty or a single formula in each sequent). We will not separate the rules for ILL here, as any intuitionistic proof can be seen as a special case of the above rules where all steps happen to maintain a single-conclusion form. In classical linear logic (CLL), multiple conclusions can appear, but one often uses the fact that $\Gamma \vdash \Delta$ is equivalent to $\vdash \neg\Gamma, \Delta$ (moving Γ formulas to right with negation) and works in a one-sided system of sequents of the form $\vdash E$ where E is a multiset of literals (each formula or its negation). Our presentation above is essentially two-sided for clarity.

We have now established the formal system of linear logic. In the next sections, we will explore the proof theory of this system – in particular, how proofs can be transformed and represented. We will define the cut-elimination procedure and prove that it always terminates (strong normalization) and preserves provability (cut admissibility). Before that, we introduce proof nets, which provide a more intrinsic view of linear logic proofs by eliminating inessential ordering information in derivations.

—

3 Proof Nets – Graphical Proofs in Linear Logic

Proof nets are a central concept in linear logic, introduced by Girard, which give a *geometric* or graphical representation of proofs. A proof net abstracts away the sequential structure and intermediate steps of a sequent calculus proof, capturing only the essential *logical flow*. Two sequent proofs that differ only by trivial reorderings (bureaucratic rule permutations) will correspond to the *same* proof net. The motivation is to identify proofs that are "essentially the same" and to eliminate spurious nondeterminism in the cut-elimination process. In this section, we define proof nets formally for the multiplicative fragment of linear logic (with negation and possibly units) and discuss their correctness criteria and sequentialization theorem. We also describe cut-elimination as a graph rewriting on proof nets and establish confluence of this process in the multiplicative case. Finally, we touch on extensions of proof nets to additives and exponentials, and the computational interpretation of proof net reduction.

3.1 Proof Structures and Proof Nets: Basic Definitions

We focus first on the multiplicative linear logic (MLL) without units (i.e. using only $\otimes$, $\parr$, and literal negation) for simplicity. Later we will consider units and other connectives.

A *proof structure* is essentially a labeled graph constructed from logical rules without regard to correctness. We will define it inductively mimicking the sequent calculus introduction rules. For MLL, the building blocks are: axioms and cut links for identity and cut, and logical links for $\otimes$ and $\parr$. Each link will connect *formula occurrences* (which we can think of as ports where formulas attach).

Definition: *Proof structure.* An MLL proof structure R consists of: - Axiom links: For every axiom rule in a proof, we have a link connecting two occurrences of dual formulas A and $A^{\perp}$. Graphically, an axiom link can be seen as a thin line or a link connecting two formula nodes A and $A^{\perp}$ (we often omit labeling the link with "Ax", it's just an identity connection). We may represent an axiom link as a pair $(u : A,\ v : A^{\perp})$ indicating formula occurrence u labeled A is connected to occurrence v labeled $A^{\perp}$. Each axiom link connects exactly two formula occurrences, one of which is designated the conclusion of an Ax rule and the other the premise. - Cut links: For each cut, we have a link that connects two occurrences of the *same* formula A and A (one from each proof being connected). In other words, a cut link connects a formula A to another formula A (which in the linear logic sequent would correspond to A and $A^{\perp}$ with one negated, but in a one-sided representation, often we treat cut as connecting $A^{\perp}$ to A implicitly). We can consider a cut link as connecting A to $A^{\perp}$ as well, but in proof nets we often keep track by polarities (some formalisms have explicit cut nodes). For simplicity, think of a cut link as an "edge" labeled Cut connecting two formula occurrences that are intended to cancel out. - Logical links: Each introduction rule ($\otimes R$, $\otimes L$, $\parr L$, $\parr R$ for MLL) corresponds to a logical link in the graph connecting the premises to the conclusion. For example: - A $\otimes$ link takes two premise edges (for A and B) and one conclusion edge (for $A \otimes B$), connecting them in a node that represents the $\otimes$ rule. In a graph, this is often drawn as a node with three ports: two for the premises (A, B) and one for the conclusion ($A \otimes B$). We will call this a tensor node or $\otimes$-link. The formula $A \otimes B$ is associated with the conclusion edge of this link, and A, B with the premise edges. - A $\parr$ link similarly has two premises (for A and B) and one conclusion (for $A \parr B$). But note in a proof net for classical logic, we often prefer a *one-sided* representation using only one type of logical node since $\parr$ is dual to $\otimes$. One common convention: use $\otimes$-nodes and treat a $\parr$ link as a $\otimes$-node on the dual formulas (since $(A \parr B)^{\perp} = A^{\perp} \otimes B^{\perp}$).

However, for clarity we can also have an explicit par node linking A and B premises to $A \parr B$ conclusion. A $\parr$-link node has one conclusion edge labeled $A \parr B$ and two premise edges labeled A and B. - Formula occurrences: The edges in these structures are labeled by formula occurrences (subformulas of the original sequent). Each logical link connects certain formula occurrences to a compound occurrence. Additionally, some formula occurrences will remain *unconnected on one end* if they are the overall assumptions or conclusions of the proof structure. Specifically, a proof structure for a sequent $\Gamma \vdash \Delta$ will have the formulas in Γ and Δ as the *open edges* (also called *conclusions* of the proof structure) not connected by an axiom or cut to anything outside.

One can inductively build a proof structure: start from the initial multiset of formula occurrences (the conclusions of the would-be proof). Apply either an axiom link between two literal occurrences P and $P^{\perp}$ (removing both from the set of open edges, as they are now linked internally), or apply an introduction link ($\otimes$ or $\parr$) by picking the required premises from the open edges and adding the resulting conclusion edge. A cut link can be added by picking two open edges with identical formula and connecting them (and marking them no longer open). In general, a proof net is intended to represent a complete proof, so eventually all formula occurrences except the final sequent's assumptions and conclusions are connected by some link exactly as they would be in a formal proof. However, *not every proof structure corresponds to a valid proof*. We next discuss the correctness criterion that characterizes those proof structures that are *proof nets*, i.e. come from an actual sequent proof.

3.2 Correctness Criterion for MLL Proof Nets

Girard's original correctness criterion for proof nets ensures that a proof structure has the essential property of a valid proof: it can be sequentialized into a derivation. One intuitive way to understand correctness is via graph properties: a proof net must not contain any cycles that correspond to an illegitimate interlocking of rules, and it must be connected in an appropriate sense (all parts of the proof are integrated). We present a common correctness criterion using the notion of switchings, which is due to Danos and Regnier, as it is easier to state formally:

A *switching* of a proof structure is a modification of the structure where for every $\parr$ logical node, we "choose" one of its two premise edges and cut the other. Concretely, each $\parr$ node has two premises (A and B). In a switching, we remove one of the premise connections (imagine "turning off" one of the two). This yields a spanning structure that is a tree or forest when the proof is correct. Formally, to check correctness: - Acyclicity: For each possible switching (each choice of one premise per $\parr$ node), the graph of the proof structure (including axiom links, cut links, and the remaining logical links after removing the unchecked $\parr$ edges) should be acyclic (contain no cycles). - Connectedness: For each switching, the resulting graph should also be connected (all formula occurrences are in one connected component, ignoring the division into left/right of the sequent).

If a proof structure passes this test (acyclic and connected for all switchings), then it satisfies the correctness criterion and is a valid proof net of MLL. Equivalently, one can say: *A proof structure is a proof net if and only if every switching of it yields a spanning tree (connected acyclic graph) connecting all the conclusions.*

Example: Consider a simple proof structure aiming to prove $A \otimes (B \parr C)$ from A and B, C. This structure might have an $\otimes$ node linking A and some edge for $(B \parr C)$ to conclude $A \otimes (B \parr C)$, and a $\parr$ node linking B and C to form $B \parr C$. If the A is linked by an axiom to an $A^{\perp}$ somewhere, etc. Checking the switching criterion involves toggling the $\parr$: choose either the branch where B is connected or the branch where C is connected, and ensure in both cases the graph has no cycle and is connected. If for one choice it disconnected or made a cycle, the structure would not be a

valid net (it would correspond to some nonsensical proof attempt).

Another simpler criterion in the case of MLL (no units) is known as the long-trip criterion: any closed path (cycle) in the proof structure that alternates between *axiom links* and *logical links* is forbidden unless it is trivial. In essence, you shouldn't be able to start at one formula occurrence, follow through a logical link to another formula, then possibly through an axiom, then through another logical link, and come back to the start, because that would indicate a cyclic dependency in the proof. The switching criterion is a combinatorial way to ensure no such cycle exists and that the proof is one piece.

Theorem (Sequentialization): *Any proof structure that satisfies the correctness criterion corresponds to a genuine sequent calculus proof in linear logic.* Conversely, any sequent proof can be translated into a proof structure that satisfies the criterion. This theorem by Girard guarantees that proof nets are a sound and complete representation of sequent proofs (at least for the multiplicative fragment). In practice, given a correct proof net, one can sequentialize it by choosing an order to apply inference rules corresponding to the links, reconstructing a sequential proof. The correctness criteria ensure this process is unambiguous.

Sketch of Sequentialization Proof: One shows by induction on the structure size that at least one of the logical connectives can be chosen as the "last rule" in a sequent proof. For a correct proof net, there will always be a logical connective whose removal (along with possibly a cut) breaks the net into two smaller parts that themselves are correct nets. For example, a $\otimes$ whose conclusion is an overall conclusion can be taken as introduced last. Removing that $\otimes$ link splits the net into subnets corresponding to the premises. The acyclicity/connectedness conditions guarantee that this decomposition is valid and yields smaller proof nets for the premises. Repeating this, one recovers a step-by-step derivation ending in the given net. The details ensure that if multiple choices seem possible, they lead to equivalent proofs (so the net really encapsulates a unique proof modulo rule permutation).

Thus, proof nets free us from thinking about the exact order of applying commutative rules (like the order of tensor introductions, etc.) and focus on the dependency structure.

3.3 Proof Net Representation and Examples

A proof net can be visualized as a graph where *axiom links* are drawn as lines connecting two identical atomic formulas (one considered A, the other $A^{\perp}$), *cut links* as lines connecting two formulas that are cut, *tensor links* as nodes with two incoming edges (premises A, B) and one outgoing edge (conclusion $A \otimes B$), and *par links* as nodes with two incoming edges (A, B) and one outgoing ($A \parr B$).

Example (Multiplicative proof net): Suppose we want a proof of the sequent $P \otimes Q, R \vdash Q \parr R, P$. One possible sequent proof (in textual form) is: from $P \otimes Q$ on the left, we can use $\otimes L$ to get P, Q on left. We also have R on left, and want $Q \parr R, P$ on right. We might exchange P to the right as $P^{\perp}$ (so now right side has $P^{\perp}, Q \parr R$ and left has Q, R) – but let's stick to one-sided view. The proof net for this would have: an axiom link connecting P on left (from $P \otimes Q$ expanded) to that same P on the right side (the conclusion P). Another axiom link connecting Q on left to Q in the conclusion $Q \parr R$ (since Q appears as part of that disjunction's premises in the net). A par link combining Q and R to form $Q \parr R$ (the conclusion), and a tensor link combining P and Q to form $P \otimes Q$ (the premise). Also a trivial edge carrying R through (or one could say an axiom with R and R if we treat R on left and R in $Q \parr R$ dual). The correctness can be checked: if we remove one branch of the par (either keep Q or R) we get no cycles. This indeed corresponds to a valid proof.

(Without diagrams, describing proof nets is cumbersome; typically, one would draw boxes and

lines. In our formal text, we focus on the definitions and properties rather than pictorial examples.)

3.4 Cut-Elimination in Proof Nets

One major advantage of proof nets is that cut-elimination (the process of removing cuts from a proof) translates to a simple, local graph rewriting on the net. In a sequent calculus, cut-elimination requires a sequence of rule permutions and transformations. In a proof net, a cut link connects two subnets, and eliminating the cut means directly connecting those subnets together, often by removing two complementary logical links back-to-back. Because proof nets ignore irrelevant sequencing, cut elimination on nets is confluent and local – parallel cuts can reduce independently without conflict. We describe the cut-elimination rules for proof nets (for MLL):

Consider a proof net containing a cut link connecting some formula A (in one part) to its dual $A^\perp$ (in another part). There will be exactly two logical links in the net that introduce those formula occurrences A and $A^\perp$. There are a few cases, depending on what the structure of A is: - If A is an atomic formula, then the cut link is connecting an axiom link pair. A cut between an atomic A and $A^\perp$ can be eliminated by simply removing both the cut link and the axiom link that introduced A and $A^\perp$. Essentially, an Ax followed by a Cut gets removed (this is the logical β-redex analogous to $A \vdash A$ cut). After removing them, any edges that were incident on the axiom link become free (if any – in this case, A and $A^\perp$ were conclusions of the axiom, now they disappear). This corresponds to the trivial case of cut elimination where an identity and cut cancel each other. - If A is a compound formula like $B \otimes C$, then one part of the net ended in a $\otimes$-link producing $A = B \otimes C$, and the other part of the net ended in some link producing $A^\perp = (B \otimes C)^\perp = B^\perp \parr C^\perp$. The cut is essentially between a $\otimes$ conclusion and a $\parr$ conclusion. The cut-elimination *reduces* this by merging the $\otimes$ link and the $\parr$ link: effectively, we connect the premises B, C of the tensor directly to the premises $B^\perp, C^\perp$ of the par. The result will connect B with $B^\perp$ (forming a smaller cut or axiom if B is atomic) and C with $C^\perp$. In other words, we replace the pattern ($\otimes$ linking B, C to A), ($\parr$ linking $B^\perp, C^\perp$ to $A^\perp$), and a cut between A and $A^\perp$ with two cuts: one between B and $B^\perp$, and one between C and $C^\perp$. This corresponds to the familiar sequent calculus reduction where a cut on $B \otimes C$ is replaced by two cuts on B and C. If B and C are simpler, those cuts might further reduce later. If either side was an axiom, one of those new cuts may become an axiom-cut pair that can cancel. - If A is a compound like $B \parr C$, a dual situation occurs: one side ends in a $\parr$ link for A, the other in a $\otimes$ for $A^\perp = B^\perp \otimes C^\perp$. The reduction similarly splits the cut into two cuts between B and $B^\perp$ and between C and $C^\perp$. - If A is additive ($B \,\&\, C$ or $B \oplus C$), cut elimination similarly propagates through, but additive cases have a subtlety: a cut on a & can introduce a choice – effectively, cut on $A \,\&\, B$ yields a situation that one side had a & node (two premises) and the other side might have to duplicate something because both A and B need to be dealt with. In proof nets for additives, one often needs to introduce some bookkeeping like making copies of subnets (the so-called *box* construction in full proof nets for additives). For the scope of this formal development, we focus on the multiplicative case where cut-elimination is straightforward. - For exponentials, cut-elimination involves gradually pushing cuts inside boxes and using rules like dereliction and promotion inversely. We will not detail those here; it's known that cut-elimination is strongly normalizing for linear logic but not as confluent when exponentials are involved (because choices of *digging* into boxes can be permuted). We stick mostly to multiplicative and additive cut-elimination.

The important property is that cut reduction in proof nets is local and strongly confluent: if a net contains multiple cuts, they can be reduced in any order or in parallel, and the final outcome (cut-free net) will be the same up to isomorphism. This is in contrast to sequent proofs in classical logic where the order of cut reduction can drastically change the end sequent or the shape of proof

(and can even diverge). Linear logic's linearity and the proof net representation ensure *confluence*. In fact, one can prove:

Proposition (Confluence of Cut-Elimination in MLL): In the multiplicative fragment, if a proof net N reduces in one step to two different nets P and Q (by reducing two different cut redexes), then there is a common net R to which both P and Q further reduce in one or more steps. Equivalently, the reduction of proof nets forms a confluent rewriting system. This means that the *normal form* of a proof net (the cut-free net) is unique, regardless of reduction order (Fernandez and Mackie, A Calculus of Interaction Nets, 1999). This property arises because linear logic disallows interfering interactions between cuts – reductions are independent thanks to linear use of formulas.

Proof Sketch: Suppose two cuts in a net are reduced in either order. One must check that the second cut-reduction is not affected by the first reduction. In MLL, if you have two cut redexes, either they are on totally separate parts of the net (trivially independent) or they share some part. If they share a part, it typically means one cut is on A, another on a subformula of A or on something independent. Careful case analysis shows reducing one doesn't prevent the other from reducing to the same final configuration. This is simpler in nets than in sequent calculus because we have no bureaucracy of rule permutations – it's direct graph rewriting. Therefore different reduction paths can be rearranged to reach the same result.

Another key property is strong normalization: there are no infinite sequences of cut reductions in propositional linear logic (no endless loop of cut-elimination). This can be shown by defining a measure (such as the multiset of formula complexities of all cuts) that decreases with each reduction step. We omit the full proof of strong normalization here, but note that Girard's original work established that linear logic is *strongly normalizing* (and this has been reproduced in various forms, e.g., via reducibility candidates). Combined with confluence, this implies cut-elimination is a *decidable* and confluent procedure that always terminates in a unique cut-free proof net.

3.5 Proof Nets with Units, Additives, and Exponentials

The presence of units $(1, \bot, \top, 0)$ and additive/exponential connectives complicates proof net correctness and cut-elimination. Typically, units are handled with *empty* links or special cases in correctness criteria (e.g., $\bot$ requires a connection to nothing, etc.). Additives $(\&, \oplus)$ require a more global correctness condition or a use of *boxes* (as in expansion nets) because they break the simple switching criterion (the & connective introduces a form of branching that is not purely local). Exponentials in proof nets are handled by the use of *boxes* as well: subnets that are encapsulated and can be duplicated or discarded as a whole. The formalism of proof structures with boxes (Girard's LC nets or later formalisms by Danos et al.) is beyond our current scope. However, the essential logical properties remain: a correct proof net including these connectives still represents a sequent proof, and cut-elimination still holds (strong normalization remains, confluence is trickier globally but holds in the sense of obtaining the same normal form modulo some commuting conversions). In later sections on categorical semantics, we will see how exponentials are interpreted in models, providing a semantic view of why cut-elimination remains coherent.

For now, having established proof nets for the core linear logic, we proceed to connect these ideas to interaction nets, which can be seen as a further abstraction focusing purely on the *cut-elimination as computation* aspect.

—

4 Cut-Elimination and Normalization in Linear Logic

Cut-elimination is a fundamental meta-theorem in proof theory: any proof with cuts can be transformed into a cut-free proof. In linear logic, cut-elimination not only holds, but it enjoys good properties (like strong normalization and confluence, as discussed). In this section, we provide a rigorous proof of the Cut-Elimination Theorem for the sequent calculus of linear logic. We also discuss the consequences: normalization (no infinite reduction) and confluence (unique normal forms). These results not only ensure logical consistency but also underlie the computational interpretation of linear logic proofs (cut-elimination corresponds to executing a program without nontermination or ambiguity, at least for terminating fragments).

4.1 Cut-Elimination Theorem

Theorem (Cut-Elimination for Linear Logic): *If a sequent $\Gamma \vdash \Delta$ is derivable in linear logic (classical or intuitionistic) using the cut rule, then $\Gamma \vdash \Delta$ is also derivable without using cut.* In other words, cut is a *redundant* rule (admissible rule) in the proof system. Every proof containing cuts can be transformed (by a sequence of local proof transformations) into a cut-free proof of the same sequent.

Proof: The proof is by structural induction on the size/complexity of the proof with cuts, or equivalently by a measure on cuts (such as the multiset of the cuts' formula sizes). The strategy is to reduce the *topmost* cut in the proof (i.e. a cut that appears highest in the derivation tree). We perform a case analysis on the last inference steps above the cut. The critical cases are when the cut formula C is introduced by the last rule in one or both subproofs. We systematically rewrite the proof to push the cut below those introductions or eliminate it entirely:

- *Case 1:* The cut formula C was introduced by a logical rule on both sides of the cut. For example, the left subproof ends in a rule introducing C on the right of the sequent, and the right subproof ends in a rule introducing C on the left. A typical scenario: left subproof ends with $\otimes R$ producing $C = A \otimes B$; right subproof ends with $\parr L$ introducing the same C on left. The cut is

$$\dfrac{\dfrac{\Pi'_L \vdash A \quad \Pi''_L \vdash B}{\Pi_L}\otimes R \qquad \dfrac{\Sigma_1, A \vdash \Theta \quad \Sigma_2, B \vdash \Theta}{\Pi_R}\parr L}{\Pi_L, \Pi_R}\,\mathsf{Cut}$$

where Π'_L and Π''_L partition Π_L and $\Sigma_{1,2}$ partition Π_R appropriately. The reduction will replace this portion with derivations of $\Sigma_1, \Pi''_L \vdash \Theta, B$ and $\Sigma_2, \Pi'_L \vdash \Theta, A$ (two smaller cuts), effectively matching A with one side and B with the other. This corresponds to the proof net reduction we described: one cut on $A \otimes B$ becomes two cuts on A and B. The measure of cut complexity decreases (the cut formula sizes strictly decrease), so this transformation will eventually remove all cuts. There are similar subcases for each combination of rules: e.g. if C introduced by $\oplus R$ on one side and $\& L$ on the other, or $!R$ on one side and $!L$ on the other (promotion/dereliction), etc. Each such pair of rules has a known localized cut-reduction step. We systematically apply the appropriate cut-reduction. - *Case 2:* The cut formula C is introduced only on one side, and on the other side it is not the principal formula of the last rule. For example, the left subproof ends by introducing C (principal formula in last rule), and the right subproof's last rule involves a different formula than C. In this case, we permute the cut upwards past the last rule of the right subproof. Since the rightmost rule did not involve C, we can move the cut above it. This typically increases the height of the cut in the proof structure but not the measure of the cut formula. By successively moving the cut up, eventually it will either encounter the introduction of C on the other side (Case 1), or reach the top of the proof. If it reaches the top (meaning C was never introduced on that

side because maybe that side was an initial axiom containing C), then the cut is between C and $C^{\perp}$ both stemming from axioms – in that case, the cut can be removed directly (Case 3 below). - *Case 3:* The cut formula C is atomic or stems from an identity/axiom. If one subproof ends with an axiom $\vdash C, C^{\perp}$ and we are cutting C with some other proof that has $C^{\perp}$, then we can simply eliminate that axiom and the cut in one step (the result is just to inline the axiom's trivial proof into the other side). This removes the cut altogether. If both sides are axioms (C and $C^{\perp}$), cut elimination is trivial as well.

By performing these transformations (Cases 1-3) exhaustively, we eventually eliminate the cut in question, at the cost of introducing zero, one, or multiple smaller cuts (Case 1 often introduces two smaller cuts, Case 2 moves the cut, Case 3 eliminates it). We then apply the inductive hypothesis or repeat the process for each remaining cut. The measure of proof complexity (e.g. the multiset of cut formula ranks or the sum of sizes of cut formulas) decreases with each reduction, guaranteeing termination of this process. Ultimately, all cuts are removed, yielding a cut-free proof. This completes the proof that any proof with cuts can be transformed into a cut-free proof.

This constructive procedure not only proves the theorem but also defines the *cut-elimination algorithm*.

Corollary (Consistency): Since linear logic has no rule that can introduce a sequent with no assumptions and no conclusions except the trivial Ax, a cut-free proof of an *empty sequent* (no premises and no conclusions, which would represent a contradiction) cannot exist. Because any proof can be made cut-free, it follows that no proof of the empty sequent exists even with cuts (unless the logic is inconsistent). Thus linear logic is consistent (cannot derive false from nothing).

4.2 Properties: Strong Normalization and Confluence

The cut-elimination process described is effectively a rewrite system on proofs (or proof nets). We now state two crucial properties of this process:

- Strong Normalization: *Every sequence of cut-reduction steps terminates after finitely many steps.* In other words, there are no infinite reduction sequences; the cut-elimination algorithm always reaches a normal form (a cut-free proof). This is proved by assigning a well-founded measure to proofs (for example, the degree of a proof defined as the sum of sizes of all cut formulas, or using a multiset ordering on the multiset of cut formula complexities) that decreases with each reduction step. Each of the reduction cases clearly reduces the measure: cutting on smaller subformulas or moving cut above a rule does not increase the sum of cut formula sizes, and when properly defined, one can show a strict decrease under a well-founded ordering. Thus infinite descent is impossible. More semantically, Girard's *candidates of reducibility* method (adapted from normalization proofs in lambda calculus) can be applied to show that every proof with cuts reduces to a cut-free proof in a finite number of steps. The strong normalization for full linear logic (including exponentials) was part of Girard's original result, though it is non-trivial especially because exponentials and contraction can simulate higherrecursion. Nevertheless, linear logic was designed to control structural rules sufficiently to maintain strong normalization (unlike full classical logic which can encode non-terminating self-referential proofs).

- Confluence: *The cut-elimination reduction is confluent (Church-Rosser property).* This means that if a proof (or proof net) P can reduce to two different proofs Q_1 and Q_2 via different sequences of reductions, then there is a further proof R to which both Q_1 and Q_2 can reduce. Equivalently, the normal form (cut-free form) of a given proof is unique (up to trivial reordering of independent rules). Confluence in the sequent calculus is a bit subtle because the order of applying reductions might yield different looking proofs. However, in the proof net representation, confluence is more transparent: as noted, proof nets eliminate the artificial nondeterminism. For

linear logic without the additives/exponentials, cut-elimination is strongly confluent in the sense of local confluence implying global confluence (Newman's Lemma applies since strong normalization gives termination). For the full logic with exponentials, confluence holds in the sense that any two maximal reduction sequences lead to logically equivalent cut-free proofs, though the exact structure (especially with permutations of promotion rules inside boxes) might differ. In practice, one often doesn't need the full strength of confluence, only the assurance that the final *multiset of axioms* or *overall result* is unique. In categorical semantics (discussed later), this uniqueness corresponds to the fact that a proof with cuts denotes a single morphism, and cut-elimination corresponds to the equality of morphisms – there is no ambiguity in the composite's semantics.

To illustrate confluence more concretely, consider a proof with two cuts that are independent (they cut on different formulas in separate parts of the proof). Reducing cut 1 first or cut 2 first yields the same end state where both are reduced. If the cuts are not independent (perhaps sharing some parts of the proof), the critical pairs of reductions can be analyzed and shown to commute. For example, one cut might be on $A \otimes B$, and another cut might involve one of A or B internally. Detailed analysis (as done in proof net confluence proofs) shows the end result is unique. Thus we can be confident that cut-elimination as a computational process does not depend on arbitrary choices, which is important for thinking of proofs as programs (the program's result is deterministic).

Remark: It's worth noting that in pure *classical* logic (without linear restrictions), cut-elimination is *not confluent* – different reduction orders can lead to very different-looking normal proofs or even different normal forms modulo permutations. The linear restriction is what restores confluence by making duplication explicit and manageable. Intuitionistic logic also has a form of confluence because with a single conclusion, there's less symmetry to cause divergence in permuting reductions. Linear logic can be seen as an even tighter control that yields a nicely behaved normalization process. This aligns with the philosophy that *proofs correspond to programs and cut-elimination corresponds to computation:* confluence ensures that program execution (cut-elimination) is deterministic in outcome, and strong normalization for terminating programs ensures no infinite loops. (Of course, with exponentials, one can encode non-terminating processes if one allows an infinite context or recursive use of !, but that goes beyond propositional logic's scope.)

Having established cut-elimination, we have essentially validated the core "execution engine" of linear logic's proof theory. Next, we transition to interaction nets, which take these ideas of local reduction and resource usage into a stand-alone computational framework.

—

5 Interaction Nets – A Graphical Computation Model

Interaction nets were introduced by Yves Lafont as a framework for parallel computation based on graph rewriting. The design of interaction nets was directly influenced by proof nets of linear logic, extracting their fundamental computational content while abstracting away the logical syntax (Lafont, Interaction Nets, 1989). An interaction net system can be seen as a set of *agents* interacting via *rules* that resemble cut-elimination steps. In this section, we give a formal definition of interaction nets, discuss the constraints that ensure good properties (like confluence and locality), and prove some fundamental theorems about interaction net reduction. We will also draw connections to the linear logic concepts introduced earlier.

5.1 Definition of Interaction Nets

An interaction net system $\mathcal{N}$ is specified by: - A set Σ of *symbols* (agent labels). Each symbol $\alpha \in \Sigma$ has an *arity* (a nonnegative integer) specifying the number of auxiliary ports of an agent of that kind. By convention, an agent has one principal port and zero or more auxiliary ports. If the arity of α is n, then an agent labeled α has $n+1$ connection points: one principal port and n auxiliary ports. - A set R of interaction rules. Each rule specifies how a particular pair of agents can interact (rewrite) when they are connected principal-port to principal-port. A rule is written as a pair of net patterns $L \to R$, where L (left side) and R (right side) are small nets (graph fragments). The left side L has exactly two agents connected to each other at their principal ports (this pair is called an active pair), possibly with some additional structure hanging off their auxiliary ports. The right side R is another net (graph) with the same free connection points as L had (to ensure the rewrite is a closed local operation). We will give a formal condition for rules below. Intuitively, a rule says: "when an agent of type α meets an agent of type β on their principal ports, they can interact and be replaced by the net R." We denote the principal connection by writing $\alpha \bowtie \beta \ \to$ (some net).

Now we define what a net is. A *net* (interaction net) over Σ is an undirected graph (more precisely a multigraph) satisfying: - Nodes of the graph are agents labeled by symbols in Σ. - Each agent node has one distinguished incident edge called the principal port edge, and other incident edges corresponding to auxiliary ports (each port holds at most one edge). - Edges connect ports of agents. An edge may connect an auxiliary port of one agent to an auxiliary port of another, or principal to auxiliary, etc., with the only restriction being at most one edge per port. Edges are undirected (there is no notion of input vs output flow on the edge itself, although we may designate port types for type system reasons later). - There may be some number of *free ports* (open connection points) where an edge end is not attached to an agent – these are considered the interface of the net (like inputs/outputs or free wires). A net with no free ports is *closed*. A net with free ports is often seen as a computation with some parameters or awaiting connection to another net.

We often depict an agent as a node with an arrow indicating the principal port. For example: an agent α of arity 2 has three ports: one principal (marked with an arrow) and two auxiliary. If we label the auxiliary ports for reference, we might draw α with three stubs, one marked as principal.

Two agents connected by their principal ports form an active pair. According to the rules R, an active pair (α, β) that matches the left side of some rule can be rewritten. If no rule applies to a given pair of agents, that pair is *inactive* (even if their principal ports are connected, if there's no rule for that combination, it's a deadlock or just no computation happens).

Linear wiring and ports: Interaction nets enforce a *linearity condition*: no port is connected to more than one edge, so the graph has no hyperedges or fanbuiltin. Also, an agent's principal port is connected to at most one other agent's port (so no agent can simultaneously actively pair with two others). This implies that at most one rule can apply at any given connection, ensuring no ambiguity in reduction. In formal terms: - *(Linearity)* Inside any interaction rule $L \to R$, each variable (name for a wire connection) occurs exactly twice: once on the left pattern and once on the right pattern (Lafont, Interaction Nets, 1989). This ensures that connections are one-to; you cannot have one wire name appearing three times (which would mean one wire connecting three ports). Thus, each edge in the net connects exactly two ports (or one port to a free end). There is no implicit copying or merging of wires in a single rewrite; duplications must be represented by explicit agents (like a "duplication" agent).

- *(Principal port uniqueness)* Each agent has exactly one principal port. Thus an active pair always consists of exactly two agents whose principal ports are connected. There is no scenario of

three agents all interacting at once at a single junction – interactions are strictly pairwise. This is sometimes called binary interaction: agents interact only two at a time, via their principal ports.

- *(No ambiguity)* For any given pair of agents connected on principal ports, there is at most one rule that applies to that pair. In practice, this means the rules R are such that if two agents α and β can interact, the system defines a unique result of that interaction. This avoids overlaps or conflicts in rewriting (the system is like an orthogonal term rewriting system). If no rule is defined for that pair, then they simply do not interact (the net is stuck unless some other interaction becomes available via rewiring).

We now formalize a reduction step in an interaction net: If an interaction net contains a subgraph isomorphic to the left side L of some rule (typically this means two agents α, β connected principal-to-principal, with certain arrangement of their auxiliary connections), then this subgraph can be replaced by the right side R of the rule, resulting in a new net. This replacement is done in such a way that any wires that were connected to the subgraph's interface (free ports of that pair) are reattached to the corresponding ports in R. Because variables (wire names) appear linearly on both sides of the rule, the connectivity is preserved. Essentially, the "context" around the active pair remains connected to whatever emerges in R. This single rewrite is called an interaction step.

One can think of an interaction net rewrite as analogous to a single step of cut-elimination: two proof structures (agents) interact and produce a new structure, with all connections (wires) correctly re-linked.

Example of a Rule As a simple illustrative example, suppose we have a symbol `Cons` (for a list constructor, arity 2: principal port plus two auxiliary for head and tail) and a symbol `Append` (for appending lists, arity 2: principal plus two auxiliary for the lists to append). We might have a rule: when a `Cons` agent's principal port meets an `Append` agent's principal port, they interact to "prepend" the head element to the result of append. Written as a rule:

$$\texttt{Cons} \bowtie \texttt{Append} \;\to\; \texttt{Append}$$

Graphically, the left side has a Cons node connected to an Append node at principal ports; Cons's auxiliary ports: one holds an element x and one a list y; Append's auxiliary ports: one connected to a list y (the same y, by sharing variable name) and one to another list z. The right side would be an Append node whose left auxiliary is y and right auxiliary is a Cons node that has x and z on its auxiliaries. In effect, this models the equation: $\texttt{Cons}(x, y)$ appended with z reduces to $\texttt{Cons}(x, (y\ \texttt{Append}\ z))$. This is a typical interaction rule making list append recursive. Such a rule respects linearity (each wire name y, z, x appears twice across left and right, maintaining connectivity).

We will not further detail specific rules here, as our focus is on general properties. However, keep in mind interaction nets can encode many data structures and algorithms via appropriate agents and rules (another classic example is the interaction net for λ-calculus reduction with sharing, where agents represent application, abstraction, and a special duplicator for shared variables).

5.2 Properties of Interaction Net Reduction

Interaction nets, by construction, have desirable properties for computation: locality, deterministic parallelism, and confluence. We state and prove the important properties, mirroring what Lafont identified.

- Locality: Interaction rules are local graph rewrites involving only two agents and their immediate connections. This means any reduction step can be performed independently of the rest of the net, as long as the interacting pair is identified. This locality is analogous to the local nature of proof net cut reduction (which only looks at the two linked inference steps involved in the cut).

- Deterministic Parallelism: Because no two rules overlap or interfere (no ambiguity and binary interactions only), if two distinct pairs of agents in a net are active (can reduce) and they share no common agent or port, they can reduce simultaneously in parallel without conflict. This is an important feature: the net can be seen as inherently parallel, and the result of parallel reductions is the same as any sequential order (formalized by confluence). Determinism is preserved even under parallel execution.

- Confluence (Strong Confluence): Formally, strong confluence means if a net N reduces in one step to P and in one step to a different net Q (two different redexes reduced), then there is a net R such that P reduces to R and Q also reduces to R in one step. In other words, any two distinct single-step reductions from the same net can be joined by another reduction step. This is a stronger property than global confluence but it implies confluence for the whole system. Proposition 1 (Strong Confluence): *If a net N contains two different active pairs that can reduce to P (reducing one pair) or to Q (reducing the other pair), then in the next step P can reduce to a net R and Q can reduce to the same net R.*

Proof Sketch: Consider two active pairs in N: say agents (A_1, A_2) form one active pair and (B_1, B_2) form another, and suppose these pairs are distinct (they do not share agents or ports – if they share a port, then it's actually the same redex or the net structure wouldn't allow that because an agent can only actively pair with one partner). Reducing (A_1, A_2) yields a net P where that pair is replaced by the right side of the rule. Reducing (B_1, B_2) yields Q. Now in P, the pair (B_1, B_2) is still present and unchanged (because the first reduction was local to (A_1, A_2)). So P can further reduce (B_1, B_2) to some net R. Similarly, in Q, the pair (A_1, A_2) is still present and can reduce to a net R' after the fact. However, one must argue that $R' = R$ – but indeed, since the rules are deterministic and local, the order of applying them does not matter. Both orders lead to a net where both redexes are resolved. Because each rule application splices in the same right-hand pattern regardless of order, the final net is isomorphic. Thus $R = R'$ is the common successor. This argument relies on the two redexes being independent (not sharing agents). If they did share an agent, then actually one reduction might disable the other – but that scenario would mean either they shared a principal port (impossible, an agent has one principal) or one was part of the other's redex pattern (which can't happen because two agents cannot simultaneously be in two distinct active pairs). Therefore, the strong confluence condition holds in all cases allowed by the net structure. This property essentially follows from the abstract rewriting system being orthogonal (left-linear and nonrules). Since our rules are linear and non-ambiguous, orthogonality gives confluence. More informally, "different reduction paths yield the same result."

- Termination (or Deadlock-freedom under conditions): Interaction net reduction may or may not terminate, depending on the rules and the net. In general, interaction nets can encode non-terminating computations (like an infinite loop in lambda calculus encoded via a cyclic net or unlimited reduction). However, if we impose conditions such as *no cyclic interaction ("vicious circles")* or a well-founded size measure on the net, we can guarantee termination. A notion introduced by Lafont is that of *simple* or *semi-simple* nets, which disallow the creation of certain cyclic structures of active pairs (Fernandez and Mackie, A Calculus of Interaction Nets, 1999). We will revisit a type discipline that ensures such simplicity.

For now, we note that if an interaction net does terminate, by confluence the normal form is unique. If it does not terminate (reductions can go on forever), we consider that analogous to a non-terminating program (which is logically consistent – it just never produces a result).

- Invariant Preservation: Interaction rules can be seen as preserving certain invariants. One trivial invariant is the multiset of *symbol occurrences* in the net (though this can change, typically

rules reduce the "complexity" or number of agents, but not always – some rules might rearrange agents). More interesting is type invariant which we will formalize in the next section: if a net is well-typed (each wire has a type and agents respect input/output types), then after any reduction step it remains well-typed. This is analogous to subject reduction in lambda calculus. We will prove this property as Proposition 6 in the next section, after defining the type system.

In summary, interaction nets form a confluent graph rewriting system that can be executed in parallel without conflicts. These formal properties fulfill the promise that the execution order of interactions does not affect the final outcome, making interaction nets naturally suited for parallel computation.

5.3 Examples and Intuition

One example worth mentioning conceptually is the λ-calculus implementation. Interaction nets have been used to implement optimal reduction for the lambda-calculus (Lamping's algorithm), by representing the sharing of sub-expressions explicitly with special agents (duplicators) instead of duplicating entire subgraphs. This application shows the power of interaction nets in handling complex rewrite systems with sharing, achieving a form of optimal evaluation (reducing each redex at most once in a shared environment). The details are beyond our scope, but conceptually: an abstraction and an application form an active pair (like a cut linking $\lambda x.M$ and an argument N) which reduces by substituting N into M – in nets this is achieved by connecting N through a duplicator to all occurrences of x in M. The explicit duplicator agent then ensures that if N has any reductions, they are shared among all its copies. This is how linear logic's resource management (exponential ! and corresponding "digging" operations) inspire the interaction net solution.

Now, having defined interaction nets and established confluence and locality, we move on to adding a type discipline to interaction nets. This will provide a way to restrict nets to those that correspond to valid computations (avoiding deadlocks like unmatched agents or cyclic interactions) and to connect interaction nets back to linear logic formally.

—

6 Typing in Interaction Nets and Deadlock-Freedom

Interaction nets, being an untyped graph rewriting system, can express many computations – including some that get stuck in undesirable ways (for example, two agents connected with no rule to reduce them, or cyclic structures that reduce forever without producing a result). To mitigate this, Lafont proposed a type discipline for interaction nets (Lafont, Interaction Nets, 1989), a complete symmetry between constructors and destructors, intended to ensure *deterministic and deadlock-free parallelism*. The idea is to assign types to ports such that only certain agent connections are allowed (those that have matching types) and every active pair has a rule. By designing the type system carefully, one can guarantee that well-typed nets do not encounter undefined interactions or vicious cycles. This section formalizes a type system for interaction nets and proves key properties: subject reduction (types are preserved by reduction) and deadlock-freedom (well-typed nets cannot get stuck in a configuration where no rule applies but the computation isn't truly finished).

6.1 Types and Typing Rules for Nets

We assume a set of *base types* (atomic types) ranged over by $\tau, \sigma, \ldots$. From these, one can construct compound types corresponding to data structures (though in many cases base types suffice to illustrate the discipline). In linear logic terms, types will play a role analogous to formulas, and agent connections will resemble linear implications between types.

Each agent symbol $\alpha \in \Sigma$ is given a type specification of the form:

$$\alpha : T^+; T_1^-, T_2^-, \ldots, T_n^-,$$

where T^+ is the type of the principal port (we decorate it with + to indicate, say, an "output" type) and T_i^- are the types of the n auxiliary ports (each decorated with $-$ to indicate "input" type). The arity of α is n in this case. We call α a constructor if its principal port is output type (positive) and a destructor if its principal port is input type; but since we mark each port with polarity, this nomenclature is just intuitive (constructors produce a data of type T^+ from some inputs, destructors consume something of type T^- to produce results). For example, we might declare $\texttt{Cons} : \texttt{List}^+; \texttt{Elem}^-, \texttt{List}^-$. This means a Cons agent has principal port type List (and it outputs a list), and two aux ports: one expects an element and one expects a list (inputs). Dually, an Append agent could be $\texttt{Append} : \texttt{List}^-; \texttt{List}^-, \texttt{List}^+$ (principal port expects a list as input, aux ports: one input list, one output list).

A net is well-typed if every edge (wire) in the net is assigned a type such that: - For each agent, if its principal port has type T^+, and an auxiliary port i has type U^-, then the wire connected to port i must carry type U (with opposite polarity on the other end if it connects to another agent's port), and the wire on the principal must carry type T (with appropriate polarity on the other side). -

(Principal Port Complementarity) If an edge connects the principal ports of two agents, then one port must have type T^+ (an output) and the other must have the complementary type T^- (an input), for some type T. In a well-typed net, connecting two principal ports with the same polarity is disallowed. In other words, principal ports must have complementary types when connected. - If an edge is free (connected to only one agent port, leaving the net), then we consider it part of the net's *interface*, and it can be assigned a type either as input or output of the net as a whole.

Typing Judgment: We can formalize typing of a net as a judgment $\Gamma \vdash N : \Delta$, where Γ is a multiset of typed free input ports of N and Δ is a multiset of typed free output ports of N. This resembles a sequent of linear logic! Indeed, one can see a correspondence: each agent's principal/aux types correspond to how a logical inference rule would use a formula (for example, a constructor agent might correspond to introducing a connective in a proof net). However, for interaction nets, we will define typing directly with rules rather than fully reducing to logic.

To construct a well-typed net, we follow these rules: - Wire rule: A single wire connecting an output port of type T to an input port of type T is a well-typed net (essentially an identity path). In typing judgment form, if one end is free output T and the other free input T, we could write $T \vdash \cdot \ : \ T$ (which is analogous to the axiom $T \vdash T$ in logic). If the wire is closed between two agent ports, it doesn't show up in the interface but must connect opposite polarities of the same type. - Agent rule: If we have an agent $\alpha : A^+; B_1^-, \ldots, B_n^-$ and we want to build a net with this agent, we should attach wires to its ports consistent with the types. For the principal port of type A^+, the wire connected to it must carry type A and have A^- on the other side if it connects to another agent or remain as free output A for the net. For each auxiliary port of type B_i^-, the wire must carry type B_i and either come from the principal port of some other agent of type B_i^+ or a free input of type B_i of the net. In typing terms, if $N_0, N_1, \ldots, N_n$ are nets that produce outputs matching $B_1, \ldots, B_n$ respectively, and we connect them into α's aux ports, then we form a new net. A simpler way: an agent introduces a relationship between the type on its principal and the types on auxiliaries, akin to a rule $B_1, \ldots, B_n \vdash A$. - Composition / Contraction rule: Placing two nets side by side with disjoint interfaces corresponds to the logical rule of exchange (concatenating

contexts). Connecting outputs of one to inputs of another corresponds to cut/composition in the type world.

Instead of building a formal proof system for typing (which would essentially recapitulate linear logic's inference rules in a direct way), we can define:

Definition: A net N is *well-typed* if there exists an assignment of types to every free port and every wire such that: 1. Each agent α with declared $\alpha : T^+; U_1^-, \ldots, U_n^-$ has its principal port's wire of type T and each auxiliary i's wire of type U_i. Moreover, if an auxiliary port is connected to another agent's port, that other port must have the opposite polarity on the same type. If it's connected to a free port, that free port is given type U_i (as an input). 2. If a principal port of an agent of type X^+ is connected to a principal port of another agent of type X'^+, then we require X' to actually be X and one of them to be considered as X^+ and the other as X^- in the typing (i.e., one of the agents must actually be declared with that type as a $-$ principal, otherwise two $+$ cannot connect). In practice, this situation shouldn't happen in a well-typed net: principal-to-principal connections should always connect complementary types by design (one agent a constructor of type, one a destructor of that type). 3. (Global well-formedness) There are no *dangling* type requirements: every wire ensures that the type on one end matches the type on the other.

This essentially means the net can be seen as a well-formed proof structure in a linear logic-like system, where agents are like logical rules. In fact, Danos and Regnier's work showed that by using *partitioning* of auxiliary ports, one can interpret multiplicative connectives in interaction nets type systems (Fernandez and Mackie, A Calculus of Interaction Nets, 1999). However, we will not dive into partitions in detail. We assume our type system is sufficiently expressive to prevent the specific deadlock called "vicious circle" (a cycle of principal ports connected together).

Vicious circle: A *vicious circle* is a cycle in the net where principal ports of agents form a closed loop (each agent's principal connected to next agent's principal, etc.), so there's no way to reduce because every connection is principal but no matching rule may apply (unless the rules allowed an n-cycle, which they typically don't). Such a cycle is an irreducible deadlock. In a well-typed net, we want to rule these out. Intuitively, this can be done by ensuring a sort of acyclicity condition on the type level – e.g., a ranking or that there's no sequence of type flow that returns to the same point. Danos Regnier's partition idea is one way to impose this: each agent's auxiliary ports are partitioned in a way that ensures principal port connections wellthe net. For simplicity, we assert:

Constraint: If a net is well-typed, it cannot contain a cycle of principal-to-principal connections. (Often this is a consequence of being able to assign a *level* to types or an ordering such that the principal connection always goes from a higher level to a lower level, preventing cycles. This is analogous to how in a proof net, correctness forbids cycles.)

Now we present formal properties:

Proposition 5 (Subject Reduction / Invariance): *If a net N is well-typed and $N \to N'$ by an interaction rule, then N' is also well-typed with the same type assignments on the interface.* In other words, well-typedness is preserved by reduction.

Proof: Consider a single interaction rule $L \to R$ being applied. The left side L consists of two agents (say α and β) with their ports and some connections. Since N was well-typed, the sub-net matching L had a valid typing. This means the two agents' principal ports had complementary types (say α had principal type X^+ and β had X^- for some X). The auxiliary connections of L each had matching types on both sides. Now the right side R is another configuration of agents and wires. The rule itself was presumably designed in a way that respects type discipline: it will have the same free connections typed as L. By the linearity of variables in the rule, each connection

in R corresponds to a connection in L type-wise. So we can carry over the type labeling from L to R. Each agent in R is one that presumably has a declared type scheme. We need to check that the way R is connected still satisfies the typing rules. Since the rule is part of the system specification, we expect that it was crafted to preserve types – essentially, $L \to R$ can be viewed as a sound logical inference transformation (like reducing a cut on a formula X yields something type-consistent). Therefore, N' inherits a consistent type assignment. More concretely, one can verify case by case: if $\alpha \bowtie \beta$ reduces to some sub-net, then the types that met at α and β will be routed through the new connections. If any new agent appears in R, its declared type relation matches what was in L. For example, in a $\otimes$-$\parr$ cut situation, the two new cuts on subformulas are well-typed if the original was. We conclude N' is well-typed. This argument is essentially the same as proving subject reduction in a typed calculus or proving that cut-elimination preserves correctness of proof nets. Indeed, one can map this reasoning to a proof-net correctness scenario.

Proposition 6 (Deadlock-freedom / Dynamic Correctness): *If a net N is well-typed (and, say, finite and closed), then N will never reach a deadlocked state except possibly the empty net.* More specifically, any irreducible (normal form) well-typed net consists only of free ports (interface) and no interacting agents. It cannot contain a vicious circle of connected agents (Fernandez and Mackie, A Calculus of Interaction Nets, 1999).

Proof: Assume for sake of contradiction that N is well-typed and in normal form (no rule can apply), but N contains some agents or connections internally. If no rule can apply and yet an agent is present, it means each agent's principal port is connected to something but not to a complementary agent that matches a rule. There are a few possibilities: (a) An agent's principal port is connected to another agent's principal port, but their pair has no rule. But if they are well-typed, one's type is X^+ and the other's X^- for some X, and if no rule exists for that pair, then the net specification is incomplete with respect to type X (this would violate determinism assumption – ideally, if such a well-typed connection exists, the system should have provided a rule; otherwise, the type system allowed an unsupported interaction which is unsound). We assume the type discipline was chosen exactly to avoid this – i.e., for any two complementary types there is a rule. Therefore case (a) likely cannot happen for a sound system. (b) An agent's principal port is free (connected to nothing) – that means the net had a free output or input that hasn't been consumed. But in a closed net (no free interface), that can't happen. In an open net, that just means the net is waiting for input or producing output, not a deadlock. (c) The only remaining case: there is a cycle of agents where each agent's principal port connects into the next agent's principal port in a ring. Such a cycle cannot reduce because it would require breaking it at some point, but if no rule applies perhaps because maybe the types go around in a loop mismatched, or simply no initial "cut" to reduce. However, a well-typed net should not allow a closed cycle: how would you assign a consistent type environment to a cycle? Typically you'd need an infinite derivation or a cyclic proof to justify it. Linear logic forbids cyclic proofs – you cannot have $A \vdash A$ proven without an axiom; a cycle of principal connections would correspond to something like $A_1 \vdash A_2, A_2 \vdash A_3, ..., A_n \vdash A_1$ collectively, which is not sequentializable into an acyclic proof net unless one of the A_i is the same and you had a cut against an axiom, but axiom would break the cycle. Thus by the correctness criterion of proof nets (or by induction on the typing derivation length), one can show that a cycle of principal connections cannot be well-typed (unless trivial). For instance, Fernandez Mackie show that well-typed configurations are semi-simple (no vicious circle) (Fernandez and Mackie, A Calculus of Interaction Nets, 1999). Therefore, the assumption that a well-typed irreducible net had a cycle leads to a contradiction – it wouldn't have been typable.

Hence, any well-typed net cannot reduce further only if it has no agent interactions left. It

might have some structure of wires connecting free ports (like an identity wiring between an input and output), which is a normal form essentially representing the result. But it cannot be stuck with two or more agents still connected. This establishes that well-typed nets are deadlock-free: they either reduce to nothing (all output produced, all input consumed) or to a form where whatever remains is essentially waiting on outside input or is just a direct connection.

Combining Propositions 5 and 6, we have what Lafont called dynamic correctness : starting from a well-typed net, every reduction sequence will successfully complete without getting stuck, and along the way the net remains well-typed.

In summary, the type discipline brings the interaction net model closer to the logical origins. It guarantees that interaction rules cover all necessary cases (no missing rules for a given type match) and that the net's structure corresponds to a logical proof-like object that cannot contain cycles (by linear logic's acyclicity). Thus, we achieve a robust framework where all well-formed programs (nets) terminate in a confluent manner.

This essentially closes the circle: interaction nets implement the cut-elimination of a logical proof in a self-contained graphical way. In the next section, we will explore more explicitly the correspondence between proof nets of linear logic and interaction nets, and discuss the categorical semantics that unify these perspectives.

—

7 Proof Nets and Interaction Nets – Correspondence and Computation

In this section, we connect the dots between the proof-theoretic view (proof nets in linear logic) and the computational view (interaction nets). We have hinted throughout that interaction nets generalize proof nets (Lafont, Interaction Nets, 1989). Here we make that connection precise in two ways: 1. Correspondence: We describe how a cut-free proof net in linear logic can be seen as an interaction net, and how cut-elimination steps correspond to interaction rule rewrites. Essentially, an interaction net can be thought of as a "program" which is the image of a proof under the Curry-Howard isomorphism, and running the program corresponds to normalizing the proof. 2. Categorical Semantics: We discuss briefly how both proof nets and interaction nets can be interpreted in a *monoidal category* or related categorical structures. This yields a high-level understanding of why the computations are confluence and what they compute.

7.1 Translating Proof Nets to Interaction Nets

Consider a proof net for a linear logic proof. A cut-free proof net is a graph of formula occurrences connected by links for rules and axiom links. We can obtain an interaction net by "forgetting" the logical labels and keeping the structure. Each logical inference link (like a $\otimes$ node or a $\parr$ node) can be seen as an interaction-net agent. In fact, one can define a mapping: - The $\otimes$ link (taking two premises A, B to one conclusion $A \otimes B$) becomes an agent with principal port type corresponding to $A \otimes B$ and two auxiliary ports of types A and B. - The $\parr$ link (one conclusion $A \parr B$, two premises) becomes an agent similarly. - An axiom link (connecting P to $P^{\perp}$) is essentially a direct wire connecting two ports (could be seen as a trivial agent or just left as a wire). - The cut link (connecting C to $C^{\perp}$) becomes an active pair of two agents (or rather, if we allow a rule directly, cut is the redex to reduce).

If we include cut links, a proof net with cuts can be directly interpreted as an interaction net: each cut is an interaction of two complementary edges. There will be exactly an interaction rule

corresponding to eliminating that cut (by the logical cut-elimination step). For example, a cut between a $\otimes$ conclusion $A \otimes B$ and a $\parr$ conclusion $(A \parr B)^\perp$ (which is $A^\perp \otimes B^\perp$) corresponds to an active pair of a $\otimes$-agent and a $\parr$-agent in the net, and the interaction rule is exactly the one that replaces them with appropriate wiring (connecting A to $A^\perp$ and B to $B^\perp$), which mirrors the proof net cut-elimination step. Thus, every step of cut-elimination in the proof net corresponds to an interaction net rewrite. This establishes a simulation: proof net reduction is simulated by the interaction net.

Conversely, given an interaction net, one can often associate it with a proof net (though not uniquely – the sequentialization theorem ensures at least existence). One interprets each agent as a logical rule introduction or a cluster of them, and each wire as a logical formula occurrence. The type discipline we introduced ensures this mapping is sound: a well-typed interaction net corresponds to a correct proof net. Essentially, the type of a wire is the formula labeling that edge in the proof net, and agents enforce the same relationships as rule introductions. The absence of vicious circles ensures the structure is sequentializable. Therefore, one could say: *well-typed interaction nets are essentially proof nets (possibly with cuts), and interaction net reduction is cut-elimination.*

This correspondence is not just analogy; it can be formalized: there exist translations between linear logic (or some fragment thereof, like multiplicative exponential) and an interaction net calculus such that the computations correspond (Lafont, Interaction Nets, 1989). For instance, the λ-calculus example is mediated by the linear logic translation of λ into proof nets (via call-by-name or call-by-value simulations).

What does this mean computationally? It means that writing a program as an interaction net and executing it is the same as writing a proof in linear logic and normalizing it. The *integrated logic approach* Lafont mentions (Lafont, Interaction Nets, 1989) is exactly this: instead of having a logic separate from computation, the computation rules *are* the logical rules. The type system we add ensures the "program" respects the logic's constraints, so that running it (reducing nets) is logically sound.

One benefit of this view is correctness by construction: if you derive an interaction net from a proof net, you know it cannot go wrong (by the cut-elimination theorem). This is akin to using proofs-as-programs: well-typed programs do not go wrong.

7.2 Categorical Semantics

Linear logic proofs have semantics in symmetric monoidal closed categories (and more structure for exponentials). Interaction nets, being a form of graph rewriting, also admit a semantic interpretation. We outline the categorical view, which will also explain confluence semantically.

A *-autonomous category* (or a linear category in Bierman's sense (Bierman, On Intuitionistic Linear Logic, 1994) is a symmetric monoidal closed category with a dualizing object, providing semantics for classical linear logic. Without going deep into category theory, the takeaway is: in such a category, each formula (type) is interpreted as an object, and each proof (net) as a morphism. Cut-elimination corresponds to the equality of two morphisms after composition. Specifically: - Each logical rule (tensor, par, etc.) corresponds to a categorical construction (tensor product, etc.). - A cut corresponds to composing two morphisms. - Cut-elimination says that a certain composed morphism can be rewritten (via natural isomorphisms of the category) to a simpler form. In a *correct* model, all ways of composing yield the same morphism (this is coherence).

For example, consider a proof that $\vdash C$ with cuts on intermediate formulas. Its interpretation is a morphism from the monoidal unit (no input) to the object C. If there are cuts, that morphism is built by composing morphisms for subproofs. One can show the composite is equal to some direct morphism that would be the cut-free proof's interpretation. This equality in the category is

the semantic manifestation of cut-elimination. Therefore, semantics ensures that cut-elimination is confluent and strongly normalizing: there is a unique morphism (the normal form) that all reduction sequences must end up at, because in the category there's just one composite morphism.

Interaction nets can also be interpreted in such categories: each agent becomes a certain morphism in the category. The interaction rules correspond to commuting diagrams or equalities of morphisms. For instance, the rule for `Cons` and `Append` in a category of lists corresponds to the equality: $\texttt{append} \circ (\texttt{cons} \times id) = \texttt{cons} \circ (id \times \texttt{append})$ (this is a familiar list concatenation identity). The confluence of the net reduction follows because in the category both paths are equal, so whichever way the net reduces, the denoted morphism is the same. Complexity of reduction doesn't affect the result.

Thus, categorical semantics provides a high-level correctness and confluence argument: every well-typed interaction net denotes a morphism, and reduction is just different ways of rewriting the composition of morphisms. Since composition is associative and categorical laws (like naturality, etc.) hold, any two reduction paths yield the same overall morphism. This is essentially another proof of confluence and normalization: you can't loop forever because that would mean an infinite descending chain of morphism equalities (impossible in a well-founded category unless there is non-termination in the logic which there isn't for propositional fragment).

7.3 Computational Interpretations

Finally, it's worth discussing what interaction nets compute. Each reduction sequence can be seen as performing a computation. For example: - In logic, a cut-free proof of $\vdash A$ is a construction (like a program output) of type A. If the sequent had hypotheses (inputs), then a proof of $\Gamma \vdash \Delta$ with Γ, Δ not empty can be seen as a function from assumptions to conclusions. - In an interaction net, if we supply certain input data on free ports, the net will reduce and eventually produce output data on other free ports. This is a computation. Confluence ensures the output doesn't depend on the order of processing parts of the net.

We can consider the complexity of this computation. Linear logic, via its exponentials, can restrict complexity. For instance, Light Linear Logic (a restriction of linear logic) characterizes polynomial time computations. Similarly, one could design interaction nets that only use certain patterns (like no unrestricted duplication) to ensure polynomial time execution. The complexity of interaction net reduction in general can vary: some nets could explode (exponential number of steps) if they mimic non-linear logic (like arbitrary recursion). But if they come from proofs in a logic known to be strongly normalizing of a certain complexity, the nets inherit that.

One practical aspect: Interaction nets can yield optimal sharing as in Lamping's algorithm. This means they can reduce certain lambda terms without duplicating work, achieving a more efficient reduction than naive beta-reduction. So in terms of complexity, interaction nets can outperform traditional reduction strategies by preserving sharing of subterms.

Another aspect is parallelism: because interaction nets allow parallel reduction, one could in principle realize a speed-up by distributing active pairs across processors. The confluence assures that this yields the same result. This is an advantage for implementing functional programming languages or graph reduction semantics.

In conclusion, interaction nets serve as a low-level model of computation grounded in the principles of linear logic. They inherit logical properties (like confluence and strong normalization under constraints) which ensure that they are a robust model for deterministic computation. The correspondence with proof nets means they are not an ad-hoc model but one justified by logic. The categorical semantics ties everything together in a unified framework, reinforcing that proofs, programs, and computations are facets of the same underlying formal processes.

—

8 Categorical Semantics of Linear Logic and Interaction Nets

(In this final section, we delve deeper into the categorical semantics that we only briefly touched upon. Readers more interested in direct syntax and reduction can skip this section, as it is more abstract. However, it solidifies understanding by placing linear logic in a mathematical context of category theory, from which many properties become intuitive.)

Linear logic can be modelled in certain categories. We will outline the key definitions: Symmetric Monoidal Closed Categories, *-autonomous categories, and Linear Categories (Seely categories), and indicate how proofs translate to morphisms. We will also comment on how interaction nets fit this picture.

8.1 Symmetric Monoidal Closed Categories (SMCC)

A symmetric monoidal closed category is a category $\mathcal{C}$ equipped with: - A monoidal product $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, which is associative up to isomorphism and has a unit object I (interpreting the multiplicative truth 1). - Symmetry isomorphisms $A \otimes B \cong B \otimes A$ satisfying coherence. - For every object A, the functor $A \otimes -$ has a right adjoint, denoted $A \multimap -$. This adjoint gives an *internal Hom*: an object $A \multimap B$ which behaves like the type of linear maps from A to B. Adjunction means $\mathcal{C}(A \otimes B, C) \cong \mathcal{C}(B, A \multimap C)$ naturally. This internal Hom interprets linear implication $A \multimap C$ or the linear function space from A to C.

In such a category, one can interpret: - Tensor $\otimes$ as the multiplicative conjunction ($\otimes$) of linear logic. - The unit I as the unit 1. - The internal Hom $A \multimap B$ as the linear implication $A \multimap B$ (which in classical linear logic corresponds to $A^{\perp} \parr B$). - If we have finite products (a terminal object for $\top$ and binary products for &) and finite coproducts (initial object for 0 and binary coproducts for $\oplus$), the category can interpret additives as well. - If the category has a comonad ! (a monoidal comonad) satisfying certain conditions (as per Seely's framework (Bierman, On Intuitionistic Linear Logic, 1994), it can interpret the exponentials of linear logic.

A proof in linear logic corresponds to a morphism in the category. The exact correspondence can be formalized via *functorial semantics*, but intuitively: - An axiom $A \vdash A$ is interpreted as the identity morphism $id_A : A \to A$. - A cut corresponds to composition of morphisms. If $\pi : \Gamma \vdash A, \Delta$ and $\rho : A, \Theta \vdash \Lambda$ are proofs (morphisms $f : \otimes\Gamma \to A \otimes \otimes\Delta$ and $g : A \otimes \otimes\Theta \to \otimes\Lambda$ in a one-sided encoding), then cutting them corresponds to $g \circ (id_A \otimes f)$ or some such composition. The details depend on the exact encoding, but essentially composition in category = cut elimination steps combined. - The logical rules like $\otimes R$ produce morphisms using the monoidal unit and associativity constraints, etc.

A *-autonomous category* is an SMCC with a dualizing object $\perp$ such that $X^{\perp} = (X \multimap \perp)$ gives a dual for every X. This interprets linear negation. *-autonomous categories interpret classical linear logic (without exponentials) fully.

8.2 Linear Categories and New–Seely Categories

In categorical models of Intuitionistic Linear Logic (ILL) the exponential modality is crucial. Seely's original definition (see [**?**]) proposes a *Seely category* as follows:

A *Seely category* is a symmetric monoidal closed category $(\mathcal{C}, \otimes, I)$ with finite products, together with a comonad $(!, \delta, \epsilon)$ on $\mathcal{C}$ and natural isomorphisms

$$n_{A,B} : !A \otimes !B \cong !(A \times B) \quad \text{and} \quad p : I \cong !1,$$

such that the functor ! sends the cartesian comonoid structure of products to the monoidal comonoid structure induced by $\otimes$.

However, as shown in later work (see, e.g., [**?**]), Seely's definition is unsound in the sense that it may not preserve equality of proofs (i.e. distinct proofs can have different interpretations). To overcome this problem, an alternative—often called the *new–Seely category* or *Linear Category*—has been introduced. Its definition is as follows:

A *Linear Category* is a symmetric monoidal closed category $\mathcal{C}$ together with a symmetric monoidal comonad

$$(!, \delta, \epsilon; m_{A,B}, m_I)$$

such that:

1. For every free !–coalgebra $(!A, d_A)$ the maps

   $$d_A : !A \to !!A \quad \text{and} \quad e_A : !A \to I$$

   endow $!A$ with a commutative comonoid structure in $\mathcal{C}$, and these maps are coalgebra morphisms.

2. Every coalgebra morphism

   $$f : (!A, d_A) \to (!B, d_B)$$

   is also a comonoid morphism.

In a Linear Category the comonad ! is monoidal via natural isomorphisms

$$m_{A,B} : !A \otimes !B \to !(A \times B) \quad \text{and} \quad m_I : I \to !1,$$

which are coherent with the comonad structure. One can then show that the co–Kleisli category $\mathcal{C}_!$ is cartesian closed. (See, e.g., [**?**] for a detailed account.) In other words, the extra structure imposed in a Linear Category guarantees that proof equality is respected in the categorical model.

Every new–Seely category (i.e. every Linear Category in the sense above) yields a cartesian closed co–Kleisli category.

Proof. The proof consists of showing that the adjunction between $\mathcal{C}$ and its co–Kleisli category $\mathcal{C}_!$ is a monoidal adjunction. In particular, one verifies that the natural isomorphisms

$$n_{A,B} : !A \otimes !B \cong !(A \times B) \quad \text{and} \quad m_I : I \cong !1$$

induce a comonad that preserves the comonoid structure. Consequently, the co–Kleisli category $\mathcal{C}_!$ inherits finite products and is cartesian closed. □

Corollary Every new–Seely (Linear) category is a sound categorical model for the multiplicative–exponential fragment of ILL.

We shall henceforth assume that our categorical model of ILL is given by a Linear Category in this sense, which provides the necessary structure to interpret proofs and to ensure that equal proofs have equal morphisms.

This matches the rules for exponentials. The Weakening and Contraction rules become properties of the coalgebra morphisms for !.

Given such a model, one can prove soundness: if a sequent is provable, then a corresponding morphism exists in all such models. And completeness: if a morphism exists, the sequent is provable

(subject to some caveats in completeness). For example, coherence spaces (with linear maps) form a *-autonomous category with a ! comonad, providing a model of linear logic.

In categorical semantics, the cut-elimination theorem corresponds to the statement: if two proofs π, π' denote the same morphism (by composition equalities), then π and π' can be transformed into each other by cut-elimination reductions. And confluence corresponds to the fact that the morphism denotation is unique, so no matter how you reduce a proof, you end up at the same morphism.

For interaction nets, giving a categorical semantics means assigning a category to the net system such that each agent is a generating morphism and each rule is a relation that becomes an equality of morphisms. Often, interaction nets semantics uses *graph rewriting* categories or adhesive categories, but one simpler approach: interpret each agent as a morphism in a monoidal category of a particular kind (for example, if we model a specific computation like arithmetic, take a category where objects are data types and morphisms are functions). Then an interaction step corresponds to replacing a composed morphism with an equivalent composed morphism (like the list append example earlier). Provided these equalities indeed hold in the category, the net rewriting is sound and confluent.

8.3 Illustration: A Simple Model

To ground this, consider a toy model: Let $\mathcal{C}$ be a category where objects are sets (types) and morphisms are binary relations (or better, sets of possible outputs given an input). This is not quite linear (it's more nondeterministic), but consider a subcategory where morphisms are actually partial functions (to keep determinism). We can interpret $\otimes$ as the Cartesian product of sets in this toy model (note that in general the monoidal product $\otimes$ in a symmetric monoidal closed category need not be the Cartesian product; here we use this interpretation for simplicity).

- I as a singleton set $\{\star\}$. - A symbol $\alpha : T^+; U_1^-, \ldots, U_n^-$ as a function $f_\alpha : U_1 \times \cdots \times U_n \to T$. This function encodes what output (of type T) the agent produces from inputs (of types U_i). For example, `Cons` : `List`; `Elem`, `List` is a function taking (elem, list) and producing a list (the result of prepending). - Two agents connected principal-to-principal means we compose their corresponding functions (the output of one feeds into the input of another). The interaction rule $\alpha \bowtie \beta \to R$ implies an equality of the composed function with the function represented by net R. If all such equalities hold, the computations will be consistent.

This sets up a kind of algebraic semantics where each rewrite rule is analogous to an algebraic law. Ensuring these laws are consistent (no contradictions) is similar to checking critical pairs in term rewriting – but if it comes from a *-autonomous category, it is consistent by construction.

While this is just a sketch, it shows that: - Linear logic provides a *logic of resources* to which interaction nets are faithful. - Categorical semantics provides *models* that guarantee uniqueness of result, giving another angle to see confluence. - Through these semantics, one can also derive complexity bounds (for instance, by interpreting ! in a category that limits its usage, one can see how often an exponential can be applied, relating to complexity).

8.4 Conclusion

- Linear Logic offers a refined resource-sensitive logic with cut-elimination as a fundamental property. - Proof Nets eliminate syntactic bureaucracy in proofs and highlight the geometric nature of cut-elimination, ensuring canonicity (correctness criteria and sequentialization theorem). - Cut-Elimination is strongly normalizing and confluent for linear logic, which we proved by induction and supported via categorical semantics. - Interaction Nets use these ideas for computation: agents and wires abstract proofs, and rewriting corresponds to cut-elimination. We listed properties (linearity, locality, confluence) and gave formal proofs (where possible) of confluence (critical pair analysis) and type preservation. - Type Discipline for nets was introduced to ensure that every

interaction is accounted for (no stuck states) and to prevent cycles. We proved that types are preserved and guarantee deadlock-freedom, aligning with Lafont's result that well-typed nets never get stuck in a vicious circle. - Proof Net vs Interaction Net correspondence was drawn, showing that well-typed nets essentially encode proofs, and net reduction corresponds to proof normalization. - Categorical Semantics were outlined to unify these perspectives and provide a soundness and confluence argument at a high level.