JFP **28**, e3, 64 pages, 2018.
doi:10.1017/S0956796818000023

A tutorial on computational classical logic and the sequent calculus

PAUL DOWNEN and ZENA M. ARIOLA
University of Oregon, USA
(*e-mail:* pdownen@cs.uoregon.edu, ariola@cs.uoregon.edu)

Abstract

We present a model of computation that heavily emphasizes the concept of *duality* and the interaction between opposites—production interacts with consumption. The symmetry of this framework naturally explains more complicated features of programming languages through relatively familiar concepts. For example, binding a value to a variable is dual to manipulating the flow of control in a program. By looking at the computational interpretation of the sequent calculus, we find a language that lets us speak about duality, control flow, and evaluation order in programs as first-class concepts.

We begin by reviewing Gentzen's LK sequent calculus and show how the Curry–Howard isomorphism still applies to give us a different basis for expressing computation. We then illustrate how the fundamental dilemma of computation in the sequent calculus gives rise to a duality between *evaluation strategies*: strict languages are dual to lazy languages. Finally, we discuss how the concept of *focusing*, developed in the setting of proof search, is related to the idea of type safety for computation expressed in the sequent calculus. In this regard, we compare and contrast two different methods of focusing that have appeared in the literature, *static* and *dynamic* focusing, and illustrate how they are two means to the same end.

1 Introduction

One of the advantages of functional programming languages is their strong foundation in mathematics. All functional languages are, in one way or another, extensions of Church (1932) λ-calculus—one of the original models of computation—as a practical programming tool. And for statically typed functional languages, the mathematical roots grow even deeper. In what's now known as the *Curry–Howard isomorphism* or *proofs-as-programs* paradigm (Curry *et al.*, 1958; de Bruijn, 1968; Howard, 1980), mathematical proofs of a theorem *are* algorithmic programs following a specification. This amazing harmony can be most clearly witnessed in the one-for-one connection between the λ-calculus and (Gentzen, 1935a) natural deduction—a system that formally lays down the rules of *intuitionistic logic*. The rules for justifying proofs in intuitionistic logic correspond exactly to the rules for writing programs in functional languages, and simplifying proofs corresponds to running programs. This connection has led to technical advances that flow both ways: not only we can use mathematics to help write programs in functional languages, but we can also write programs to help develop mathematics with proof assistants.

Natural deduction is not the only logic, however. In fact, natural deduction has a twin sibling called the *sequent calculus*, born at the same time within the seminal paper of Gentzen (1935a; 1935b). Whereas the rules of natural deduction more closely mimic the reasoning that might occur in the minds of mathematicians, the rules of the sequent calculus are themselves easier to reason about, for example, if we want to show that the logic is consistent. Furthermore, unlike natural deduction's presentation of intuitionistic logic, Gentzen's sequent calculus provides a native language for *classical logic* that admits additional reasoning principles like proof by contradiction: if a logical statement cannot be false, then it must be true. As a consequence, the sequent calculus clarifies and reifies the many dualities of classical logic—"true" is dual to "false," "and" is dual to "or"—as pleasant symmetries baked into the very structure of its rules. Yet, even though these two systems look very different from each other and have their own distinct advantages and limitations, they are closely connected and give us different perspectives into the underlying phenomena of logic. And from our point of view, the more vantage points we have, the better.

But since the proofs-as-programs paradigm connects a logic like natural deduction to a language like the λ-calculus, should not there also be some programming language that is connected to the sequent calculus in the same way? As it turns out, there is (Herbelin 1995, 2005)! When interpreted as a programming language, the natural symmetries of the sequent calculus reveal hidden dualities in programming—input and output, production and consumption, construction and deconstruction, structure and pattern—and makes them a prominent part of the computational model. Fundamentally, the sequent calculus expresses computation as an interaction between two opposed entities: a *producer* representing a program that creates information, and a *consumer* representing an environment or context that observes information. Computation then occurs as a communication protocol allowing a producer and consumer to speak to one another. This two-party, protocol-based style of computation gives a different view of computation than the one shown by the λ-calculus. In particular, programs in the sequent calculus can also be seen as configurations of an abstract machine (Ariola *et al.*, 2009), in which the evaluation context is reified as a syntactic object that may be directly manipulated. And due to the connection between classical logic (Griffin, 1990) and control operators like Scheme's (Kelsey *et al.*, 1998) `call/cc` or Felleisen's ((1992)) $\mathscr{C}$, the built-in classicality of the sequent calculus also gives an effectful language for manipulating control flow.

The computational interpretation of the sequent calculus is not just an intellectual curiosity. Thanks to the relationship between natural deduction and the sequent calculus as sibling logics (Gentzen, 1935b), the sequent calculus gives us another angle for investigating real issues that arise in the λ-calculus and functional programming, from source languages down to the machine. For example, in a panel discussion among leading type theorists (Singh *et al.*, 2011), McBride points out how the poor foundation for the computational interpretation of co-induction is a road block for program verification and correctness, which is in contrast to the robust and powerful treatment of induction in functional languages and proof assistants. However, the

symmetries of the sequent calculus show us how both induction and co-induction can be represented as equal and opposite reasoning principles under the unifying umbrella of *structural recursion* (Downen *et al.*, 2015) for both ordinary recursive types and generalized algebraic datatypes. This computational symmetry between induction and co-induction is based on the duality between data types in functional languages and co-data types as objects (Downen and Ariola, 2014), and gives a more robust way for proof assistants to handle recursion in infinite objects.

Moving down into the intermediate representation of programs that exists within optimizing compilers, the logic of the sequent calculus (Downen *et al.*, 2016) shows how compilers can use continuations in a more direct way with a "strategically defunctionalized" (Reynolds, 1998) *continuation-passing style* (CPS). This compromise between continuation-passing and direct style makes it possible to transfer techniques between CPS (Appel, 1992) and *static single assignment* (Cytron *et al.*, 1991) compilers like SML/NJ with direct style compilers like the Glasgow Haskell Compiler (GHC). For example, CPS can faithfully represent *join points* in control flow (Kennedy, 2007), whereas direct style can use arbitrary transformations expressed in terms of the original program (Peyton Jones *et al.*, 2001). Finally, the sequent calculus can also be interpreted as an even lower-level, machine-like language for functional programs (Ohori, 1999), which can be used to reason about fine details like manual memory management (Ohori, 2003). Therefore, the computational interpretation of the sequent calculus acts like a beacon illuminating murky areas in both the design and implementation of functional languages.

1.1 Overview

The objective of this paper is to give an introduction and tutorial to the computational interpretations of the classical sequent calculus as a programming language, with a particular focus on the dualities found in computation and their connection to functional programming. As the broad motivation is for modeling functional programs, we assume that the reader is already familiar with the λ-calculus, natural deduction, and the Curry–Howard correspondence between these two formal systems. We do not, however, assume any previous familiarity with the sequent calculus, and will first provide a review of the sequent calculus as a system of logic before illustrating how it can also be used as a system of computation. The goal of this tutorial is to give a basic and broadly applicable introduction to a family of formal programming languages based on the classical sequent calculus, for the purpose of understanding their applications to functional programming. The reader will then be equipped to adapt existing applications of the sequent calculus (like those mentioned previously) to new scenarios and to use the sequent calculus to discover and develop new solutions to problems in programming languages.

There are different possible computational interpretations that can be given to the sequent calculus, which is partly due to two dilemmas that arise when designing a language based on the sequent calculus. The first and most fundamental dilemma of computation is that the evaluation of individual programs can easily have several diverging paths to choose from that lead to different and incompatible futures. Thus,

a language for the sequent calculus needs an *evaluation strategy*—corresponding to the difference between call-by-value (like ML) and call-by-name (like Haskell) functional languages—for deterministically deciding which path to go down. The second dilemma of computation is that important tasks can be buried within a program, and those tasks must be brought to the surface to complete the evaluation of the program. The job of bringing tasks to the forefront of a program—related to *focusing* (Andreoli, 1992; Laurent, 2002) in logic—can be done at one of two points in the lifetime of the program: either up front at "compile time" before the program is evaluated, or in the moment at "run time" during the evaluation process.

To begin, we introduce and motivate the basic premise of the sequent calculus with its contrast to natural deduction: whereas natural deduction is a logic about just *truth*, the sequent calculus is a logic equally about both *truth* and *falsehood* (Section 2). With this premise in mind, we then review the original classical logic of the sequent calculus: (Gentzen, 1935a) LK (Section 3). In order to draw a programming language from LK, we need a little extra structure than the austere logic provides. Thus, we introduce the core calculus (Herbelin, 2005) that lets us read proofs in the sequent calculus as programs (Section 4). Although the core calculus is rather basic, it is still expressive enough to exhibit the first computational dilemma of evaluation strategy in the sequent calculus. We then populate the core calculus with the logical connectives of LK to give the dual calculi that combine the languages introduced by Curien and Herbelin (2000) and Wadler (2003; 2005) (Section 5). The dual calculi solve the dilemma of evaluation strategy through the language: the dual calculi are actually two different languages—one call-by-value and one call-by-name—with a common syntax that are logically and computationally dual to one another in a way that reaffirms (Filinski, 1989) observation. Additionally, within the dual calculi, we have two approaches to address the dilemma of focusing in the language: either through two different sub-syntaxes in the style of LKQ and LKT (Curien and Herbelin, 2000) that are coordinated with the evaluation strategy to only let us write well-behaved programs, or through adding the missing steps, known as ς-rules (Wadler, 2003), to the evaluation process.

2 Truth versus falsehood

Gentzen (1935a) simultaneously developed both natural deduction and the sequent calculus as formal systems for symbolic logic: tools for studying propositions (which we denote by the variables $A, B, C, \dots$) that might be true or false. One of the ground-breaking insights of the sequent calculus is the use of its namesake sequents to organize the information we have about the various propositions in question. In its most general form, a *sequent* is a conditional conglomeration of propositions:

$$A_1, A_2, \dots, A_n \vdash B_1, B_2, \dots, B_m$$

pronounced "A_1, A_2, …, and A_n entail B_1, B_2, …, or B_m," which states that assuming *each* of $A_1, A_2, \dots, A_n$ is true then at least *one* of $B_1, B_2, \dots, B_m$ must be true. The turnstile ($\vdash$) in the middle of the sequent separates the sequence of *hypotheses* on

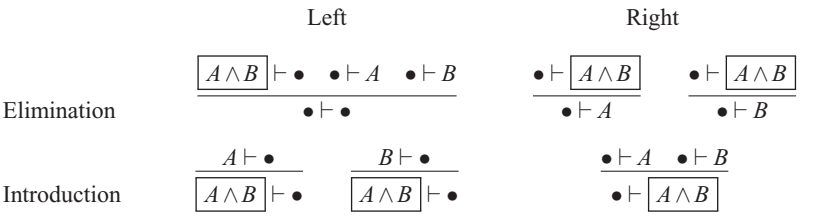


Fig. 1. The orientation of deductions for conjunction ($\wedge$).

the left, which we collectively write as Γ, from the sequence of *consequences* on the right, which we collectively write as Δ.

This separation between the left and right sides of the sequent gives the essential skeletal structure of the sequent calculus as a logic. As special cases, we can form several basic judgments about logical propositions using our above interpretation of the meaning of the sequents by observing that an empty collection of hypotheses denotes "true" and an empty collection of consequences denotes "false" (both written as $\bullet$). A single consequence without hypotheses $\bullet \vdash A$ means "A is true", a single hypothesis without consequences $A \vdash \bullet$ means "A is false", and the empty sequent $\bullet \vdash \bullet$ is a primitive contradiction "true entails false." So already, the basic structure of the sequent gives us a language for speaking about truth, falsehood, and contradiction without assuming anything else about the logic.

The propositions that we deal with in both the logic of natural deduction and the sequent calculus are meant to represent falsifiable or verifiable claims in a particular domain of study, such as "0 is greater than 1." However, in their simplest form, these logics do not account for domain-specific knowledge and leave such basic propositions as *atoms* or uninterpreted variables. Instead, the primary interest of the logic is to characterize the meaning of *logical connectives* that combine or modify existing propositions such as conjunction ($A \wedge B$), disjunction ($A \vee B$), or implication ($A \supset B$). Logic gives us a method for describing the logical connectives by asserting the rules for valid *inferences* we can make of the form:

$$\frac{H_1 \quad H_2 \quad \ldots \quad H_n}{J}$$

where the validity of the *conclusion* J necessarily follows from the validity of the several *premises* $H_1, H_2, \ldots, H_n$, each of which stand for particular sequents.

For example, we can sensibly assert the validity of the deductions involving conjunction shown in Figure 1 based on the meaning of conjunction. Due to the interaction between entailment in the sequent (separating hypotheses from consequences) and the line of inference (separating premises from conclusions), we have two dimensions for orienting inference rules based on the location of their *primary* proposition (marked with a box in Figure 1). On the one hand, rules where the primary proposition appears to the right or left of the turnstile are called *right* and *left* rules, respectively. On the other hand, rules where the primary proposition appears below or above the line of inference are called *introduction* and *elimination*

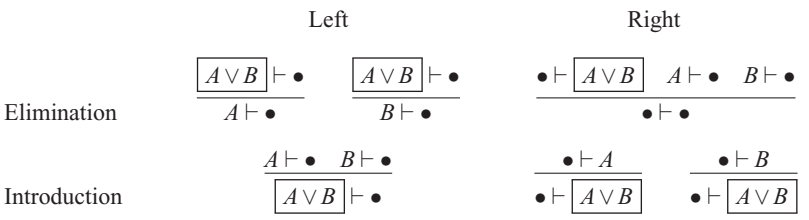


Fig. 2. The orientation of deductions for disjunction ($\vee$).

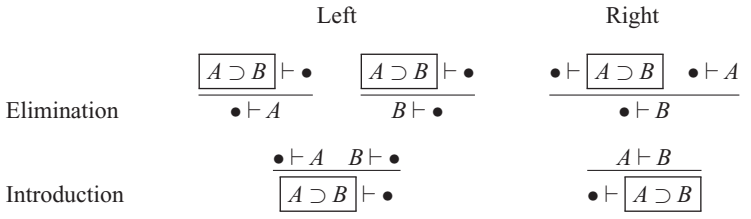


Fig. 3. The orientation of deductions for implication ($\supset$).

rules, respectively. This gives us four quadrants where the rules of inference for conjunction might live.

- Right introduction: if both A and B are true then we can deduce that $A \wedge B$ is true.
- Right elimination: if $A \wedge B$ is true, then we can deduce that A is true and likewise that B is true.
- Left introduction: if A is false then we can deduce that $A \wedge B$ is false, and likewise if B is false.
- Left elimination: if it happens that $A \wedge B$ is false and also both A and B are true, then we must have a contradiction somewhere, as this represents an impossible situation.

Similar inference rules can be given for disjunction and implication under the same right/left and introduction/elimination orientations as shown in Figure 2 and 3. It is interesting to note that the premise to the right introduction rule for implication does not have the same basic form of sequent as in all the other rules. It seems that we need to use the inherent entailment built into sequents to confirm the truth of an implication, so that from $A \vdash B$ (*i.e.,* "A entails B") we can deduce $\bullet \vdash A \supset B$ (*i.e.,* "A implies B is true").

With the dimensions of logical orientation illustrated in Figure 1–3, we can identify one of the primary distinctions between natural deduction and the sequent calculus. Natural deduction is exclusively made up of *right rules*—including both right introduction and right elimination—and the sequent calculus is exclusively made up of *introduction rules*—including both right introduction and left introduction. But neither make use of the left eliminations. In other words, natural deduction is concerned with verifying and using the *truth* of propositions, whereas the sequent calculus is concerned with both the true and false *introductions* of logical connectives.

$$A,B,C \in \mathit{Proposition} ::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid \neg A \mid A \supset B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$\Gamma \in \mathit{Hypothesis} ::= A_1, \ldots, A_n$$
$$\Delta \in \mathit{Consequence} ::= A_1, \ldots, A_n$$
$$\mathit{Sequent} ::= \Gamma \vdash \Delta$$

Core rules:

$$\frac{}{A \vdash A}\ Ax \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut$$

Structural rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A,\Delta}\ WR \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ WL \qquad \frac{\Gamma \vdash A,A,\Delta}{\Gamma \vdash A,\Delta}\ CR \qquad \frac{\Gamma,A,A \vdash \Delta}{\Gamma,A \vdash \Delta}\ CL$$

$$\frac{\Gamma \vdash \Delta,A,B,\Delta'}{\Gamma \vdash \Delta,B,A,\Delta'}\ XR \qquad \frac{\Gamma',B,A,\Gamma \vdash \Delta}{\Gamma',A,B,\Gamma \vdash \Delta}\ XL$$

Logical rules:

$$\frac{}{\Gamma \vdash \top,\Delta}\ \top R \qquad \text{no } \top L \text{ rule} \qquad \text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \vdash \Delta}\ \bot L$$

$$\frac{\Gamma \vdash A,\Delta \quad \Gamma \vdash B,\Delta}{\Gamma \vdash A \wedge B,\Delta}\ \wedge R \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_1 \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_2$$

$$\frac{\Gamma \vdash A,\Delta}{\Gamma \vdash A \vee B,\Delta}\ \vee R_1 \qquad \frac{\Gamma \vdash B,\Delta}{\Gamma \vdash A \vee B,\Delta}\ \vee R_2 \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee L$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A,\Delta}\ \neg R \qquad \frac{\Gamma \vdash A,\Delta}{\Gamma, \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, A \vdash B,\Delta}{\Gamma \vdash A \supset B,\Delta}\ \supset R \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma, A \supset B \vdash \Delta', \Delta}\ \supset L \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma \vdash A - B, \Delta', \Delta}\ -R \qquad \frac{\Gamma, A \vdash B,\Delta}{\Gamma, A - B \vdash \Delta}\ -L$$

$$\frac{\Gamma \vdash A,\Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \forall X.A,\Delta}\ \forall R \qquad \frac{\Gamma, A\{B/X\} \vdash \Delta}{\Gamma, \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash A\{B/X\},\Delta}{\Gamma \vdash \exists X.A,\Delta}\ \exists R \qquad \frac{\Gamma, A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma, \exists X.A \vdash \Delta}\ \exists L$$

Fig. 4. Gentzen's LK sequent calculus.

With this fundamental characterization of the sequent calculus in mind, we will delve into the original sequent-based logic: LK.

3 Gentzen's LK

Gentzen's LK, a simple logic based extensively on the use of the sequents to trace local hypotheses and consequences throughout a proof, is given in Figure 4. The sequents are built out of finite, ordered sequences of propositions denoted by the metavariables Γ and Δ, which may be (1) empty (written $\bullet$), (2) a single proposition (written as just A), or (3) a concatenation of two sequences (written with a comma as Γ, Γ' and Δ, Δ'). Inference rules let us build proof trees by stacking inferences on top of one another. In addition to the binary logical connectives for conjunction,

disjunction, and implication, as well as constants for truth ($\top$) and falsehood ($\bot$), we also include negation (written $\neg A$ and read "not A") as a unary logical connective and subtraction (written $A - B$ and read "A but not B") as the dual to implication. Finally, LK also contains two quantifiers—universal (written $\forall X.A$ and read "for all X, A") and existential (written $\exists X.A$ and read "there is an X such that A")—that abstract over propositional variables denoted by X, Y, or Z. More specifically, the quantifiers act as binders for propositional variables: both $\forall X.A$ and $\exists X.A$ *bind* all occurrences of X in A (otherwise a variable is *free*), and propositions are considered equal up to renaming of bound variables so $\forall X.A = \forall Y.A\{Y/X\}$ and $\exists X.A = \exists Y.A\{Y/X\}$. For simplicity, we limit the presentation to second-order propositional logic, meaning that $\forall$ and $\exists$ only quantify over propositions, not another domain of discourse, like numbers.

Core inference rules

The various inference rules of LK can be thought of in three groups that collectively work toward different objectives. The first group, containing just the axiom (Ax) and cut (Cut) rules, gives the core of LK. The Ax rule lets us draw consequences from hypotheses with the understanding that "A entails A" for any proposition A. The Cut rule lets us eliminate intermediate propositions from a proof. For example, the special case of the Cut rule where the hypothesis Γ, Γ' and consequences Δ, Δ' are all empty is

$$\frac{\bullet \vdash A \quad A \vdash \bullet}{\bullet \vdash \bullet}\ Cut$$

In other words, if there is a proposition A that we know is both true ($\bullet \vdash A$) and false ($A \vdash \bullet$), then we can deduce that a contradiction has taken place ($\bullet \vdash \bullet$). We can then use the intuitive reading of the sequents to extend this reasoning to the general form of Cut, meaning that it is valid to allow additional hypotheses and alternate consequences in both premises when eliminating a proposition in this fashion so long as they are all gathered together in the resulting conclusion. Both Ax and Cut play an important part in the overall structure of LK proof trees. The Ax serves as the primitive leaves of the proof, signifying that there is nothing interesting to justify because we have just what is needed. The Cut lets us use auxiliary proofs or "lemmas" without them appearing in the final conclusion, where on the one hand we show how to derive a proposition A as a consequence and on the other hand we assume A as an hypothesis that may be used in another proof.

Structural inference rules

Next, we have group of inference rules aim to describe the structural properties of the sequents themselves that arise from their meaning. The *weakening* rules say that we can make any proof weaker by adding additional unused hypotheses (WL) or considering alternative unfulfilled consequences (WR) since the presence of irrelevant propositions does not matter. The *contraction* rules say that duplicate hypotheses (CL) and duplicate consequences (CR) can just as well be merged into

one since redundant repetitions do not matter. And finally, the *exchange* rules say that hypotheses (XL) and consequences (XR) can be swapped since the order of propositions does not matter. So even though the hypothesis Γ and consequence Δ of a sequent are both formally represented by ordered sequences, the net effect of the contraction and exchange structural rules is to make them behave like sets—wherein order and amount is ignored—for the purpose of deriving proofs.

It may seem strange that the meaning of a sequent with multiple consequences is that only *one* consequence must be true instead of *all* consequences being true. In other words, the consequences of a sequent are *disjunctive* rather than *conjunctive* so that, for example, $A \vdash B, C$ means "A entails B or C" instead of "A entails B and C." One reason for this interpretation is that disjunctive consequences can be weakened but conjunctive consequences cannot. For example, if we already know that "A entails B or C" then we can deduce "A entails B or C or D" for any D because we already know that either B or C is a consequence of A, so the status of D is irrelevant. However, if we already know that "A entails B and C" then we do not know much about "A entails B and C and D" in general, since D might not actually follow from A at all. A similar argument also explains why the hypotheses of a sequent are conjunctive rather than disjunctive. Therefore, the meaning of sequents, where *all* hypotheses must entail *one* consequence, is essential for enabling weakening on both sides of entailment.

Logical inference rules

Finally, we have the group of inference rules that aims to characterize the logical connectives. These logical rules are generalizations of the introduction rules for the connectives from Figures 1–3: the left rules are named with an L and the right rules are named with an R. Compared to our basic observations, each logical rule is generalized with additional hypotheses and alternative conclusions that are "along for the ride," similar to *Cut*. For example, the two left introduction rules for conjunction in Figure 1 are generalized to

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

which say that if Δ is a consequence of A and Γ, then Δ is just as well a consequence of $A \wedge B$ and Γ (and similarly for B). Likewise, the sequents $\Gamma \vdash \top, \Delta$ and $\Gamma, \bot \vdash \Delta$ are true independent of Γ and Δ because $\top$ is trivially true and $\bot$ is trivially false. Since we also consider both logical negation ($\neg A$) and logical subtraction ($A - B$) as connectives, they too are equipped with left and right introduction rules in Figure 4. The rules for negation have the following special cases when Γ and Δ are empty:

$$\frac{A \vdash \bullet}{\bullet \vdash \neg A}\ \neg R \qquad\qquad \frac{\bullet \vdash A}{\neg A \vdash \bullet}\ \neg L$$

In other words, whenever A is false we can infer that $\neg A$ true, and whenever A is true we know $\neg A$ is false. Similarly, the rules for subtraction have the following

special cases when the hypotheses and consequences are empty:

$$\frac{\bullet \vdash A \quad B \vdash \bullet}{\bullet \vdash A - B}\ -R \qquad\qquad \frac{A \vdash B}{A - B \vdash \bullet}\ -L$$

In other words, whenever both A is true and B is false we can infer that $A - B$ is true, and whenever A entails B we know that $A - B$ must be false. Intuitively, the subtraction $A - B$ can be understood as a single connective with the same logical meaning as the compound proposition $A \wedge (\neg B)$, in the same way that the implication $A \to B$ can be understood as a connective with the same meaning as $(\neg A) \vee B$.

Perhaps the most subtle logical connectives in LK are the quantifiers $\forall$ and $\exists$. The special cases of the introduction rules for $\forall X.A$ and $\exists X.A$ when Γ and Δ are empty are as follows:

$$\frac{\bullet \vdash A}{\bullet \vdash \forall X.A}\ \forall R \qquad \frac{A\{B/X\} \vdash \bullet}{\forall X.A \vdash \bullet}\ \forall L \qquad \frac{\bullet \vdash A\{B/X\}}{\bullet \vdash \exists X.A}\ \exists R \qquad \frac{A \vdash \bullet}{\exists X.A \vdash \bullet}\ \exists L$$

For universal quantification over the variable X in A, if we can prove that A is true without knowing anything about X then we can infer that $\forall X.A$ is true, and if we can exhibit a specific B such that A with B for X is false then we have a counterexample showing that $\forall X.A$ is false. Existential quantification over the variable X in A is reversed, so that exhibiting a specific B such that A with B for X is true is an example showing that $\exists X.A$ is true, whereas showing that A is false without knowing anything about X lets us infer that $\exists X.A$ is false. The extra subtlety of the quantifiers lies in ensuring that we "know nothing else about X." In the sequent calculus, this extra constraint can be captured in the side condition that the variable X does not appear free anywhere else in the sequent, written as the premise $X \notin FV(\Gamma \vdash \Delta)$ in both the $\forall R$ and $\exists L$ rules.

Notice that this extra side condition really is necessary, since without it both quantifiers collapse into one, which is clearly not what we want. For example, we should expect that a $\forall$ entails the corresponding $\exists$, that is $\forall X.A \vdash \exists X.A$, which is proved as follows by choosing *any* arbitrary proposition B to substitute for X:

$$\dfrac{\dfrac{\dfrac{}{A\{B/X\} \vdash A\{B/X\}}\ Ax}{\forall X.A \vdash A\{B/X\}}\ \forall L}{\forall X.A \vdash \exists X.A}\ \exists R$$

So every $\forall$ entails the corresponding $\exists$. Intuitively, the converse should not hold; it should not be that an $\exists$ always entails the corresponding $\forall$. However, consider the following attempted proof of $\exists X.A \vdash \forall X.A$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax \qquad X \notin FV(\bullet \vdash A)}{\exists X.A \vdash A}\ \exists L \qquad X \notin FV(\exists X.A \vdash \bullet)}{\exists X.A \vdash \forall X.A}\ \forall R$$

The this proof is only valid when the side conditions X are met: $X \notin FV(\exists X.A \vdash \bullet)$ is always true for any A but $X \notin FV(\bullet \vdash A)$ only holds when X does not appear free in A. In other words, the $\forall$ and $\exists$ quantifiers are only logically equivalent when their quantified variable is never referenced. When instantiating A as just X for example,

the sequent $\exists X.X \vdash \forall X.X$ is not provable only because of the side conditions since X is indeed free in X. Therefore, the side conditions on $\forall R$ and $\exists L$ are essential for keeping the intended distinct meanings of the quantifiers.

Collapsing $\forall$ and $\exists$ is not just troublesome for the quantifiers themselves, but catastrophically collapses truth and falsehood in the logic as a whole. More specifically, removing the side conditions from $\forall R$ and $\exists L$ makes LK inconsistent by making the contradictory sequent $\bullet \vdash \bullet$ derivable. One such derivation of contradiction is built in three parts. First, we can prove that $\exists X.X$ is true because there is *some* provably true proposition in LK, for example $\top$. Second, we can prove that $\forall X.X$ is false because there is *some* provably false proposition in LK, for example $\bot$. Third, without the side conditions on free propositional variables, we would be able to derive a proof of $\exists X.X \vdash \forall X.X$ as seen above, which is the glue that connects the first two parts together via cuts. In total, we would be able to derive the following contradiction in LK:

$$\dfrac{\dfrac{\dfrac{}{\bullet \vdash \top}\,\top L}{\bullet \vdash \exists X.X}\,\exists R \qquad \dfrac{\dfrac{\dfrac{}{X \vdash X}\,Ax}{\exists X.X \vdash X}\,\exists L}{\exists X.X \vdash \forall X.X}\,\forall R}{\bullet \vdash \forall X.X}\,Cut \qquad \dfrac{\dfrac{}{\bot \vdash \bullet}\,\bot L}{\forall X.X \vdash \bullet}\,\forall L}{\bullet \vdash \bullet}\,Cut$$

which is only ruled out by the side conditions on $\forall R$ and $\exists L$ that prevent a proof of the sequent $\exists X.X \vdash \forall X.X$.

3.1 Goal-directed proof search

LK enables a "bottom up" style of building proofs by starting with a final sequent as a goal that we would like to prove and building the rest of the proof up from there. When read in reverse, each logical rule identifies a connective in the goal below the line of inference and breaks it down into simpler sub-goals above the line. For example, let us consider how to build an LK proof that the proposition $((A \wedge B) \wedge C) \supset (B \wedge A)$ is true. First, we begin with the sequent $\bullet \vdash ((A \wedge B) \wedge C) \supset (B \wedge A)$ as the goal and notice that the primary connective exposed in the only proposition available is implication, so we can apply the right implication rule:

$$\dfrac{\begin{array}{c}\vdots\\ (A \wedge B) \wedge C \vdash B \wedge A\end{array}}{\bullet \vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\,\supset R$$

Next, we may break down the conjunction in the consequence $B \wedge A$ with the right conjunction rule, splitting the proof into two parts:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ (A \wedge B) \wedge C \vdash B\end{array} \quad \begin{array}{c}\vdots\\ (A \wedge B) \wedge C \vdash A\end{array}}{(A \wedge B) \wedge C \vdash B \wedge A}\,\wedge R}{\bullet \vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\,\supset R$$

At this point, the consequences of both our goals are generic, lacking any specific connectives to work with. Therefore, we must shift our attention to the left and

begin breaking down the hypotheses. Since the hypothesis $(A \wedge B) \wedge C$ contains a superfluous C, we use the first left conjunction rule in both branches of the proof to discard it:

$$\dfrac{\dfrac{\dfrac{\vdots}{A \wedge B \vdash B}}{(A \wedge B) \wedge C \vdash B}\wedge L_1 \quad \dfrac{\dfrac{\vdots}{A \wedge B \vdash A}}{(A \wedge B) \wedge C \vdash A}\wedge L_1}{\dfrac{(A \wedge B) \wedge C \vdash B \wedge A}{\bullet \vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset R}\wedge R$$

Now, we may apply another left conjunction rule to select the appropriate hypothesis needed for both sub-proofs:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{B \vdash B}}{A \wedge B \vdash B}\wedge L_1}{(A \wedge B) \wedge C \vdash B}\wedge L_1 \quad \dfrac{\dfrac{\dfrac{\vdots}{A \vdash A}}{A \wedge B \vdash A}\wedge L_1}{(A \wedge B) \wedge C \vdash A}\wedge L_1}{\dfrac{(A \wedge B) \wedge C \vdash B \wedge A}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset R}\wedge R$$

And finally, we can now close off both sub-proofs with the Ax rule, finishing the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{B \vdash B}Ax}{A \wedge B \vdash B}\wedge L_1}{(A \wedge B) \wedge C \vdash B}\wedge L_1 \quad \dfrac{\dfrac{\dfrac{}{A \vdash A}Ax}{A \wedge B \vdash A}\wedge L_1}{(A \wedge B) \wedge C \vdash A}\wedge L_1}{\dfrac{(A \wedge B) \wedge C \vdash B \wedge A}{\bullet \vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\supset R}\wedge R$$

3.2 Consistency and cut elimination

One of Gentzen's motivations for developing the LK sequent calculus was to study the consistency of natural deduction. A consistent logic does not prove a contradiction, so that no proposition is proven both true and false. More specifically, we can say that a sequent calculus is *consistent* whenever there is no proof of the empty sequent $\bullet \vdash \bullet$. For a logic like LK, these two conditions are the same: from a contradiction weakening gives us $\bullet \vdash A$ and $A \vdash \bullet$ for any A, and from any A that's proven both true and false, *Cut* gives us $\bullet \vdash \bullet$. Consistency is important because without it provability is meaningless: it is not particularly interesting to exhibit a proof that some proposition A is true when we already know of a single proof that shows every proposition is true (and false)!

So in the interest of showing LK's consistency, how might we possibly begin to build a proof of the empty sequent from the bottom up? Let us consider which of LK's inference rules (from Figure 4) could possibly deduce $\bullet \vdash \bullet$. It cannot be any of the structural rules because they all force at least one hypothesis or consequence in the conclusion below the line. Likewise, it cannot be any of the logical rules: since they are introduction rules, they all include at least one proposition built from a connective on either side of the deduced sequent. It also cannot be the axiom

rule, which only deduces simple non-empty sequents of the form $A \vdash A$. Indeed, the *only* inference rule that might ever deduce an empty sequent—and therefore lead to inconsistency—is *Cut* as shown previously.

This observation that only cuts can lead to contradictions is Gentzen (1935b) great insight to logical consistency. If we want to know that a sequent calculus like LK is consistent, it is enough to ask if the *Cut* rule is important for provability. If *Cut* is not essential in any proof, so any provable sequent can be deduced without the help of *Cut*, then $\bullet \vdash \bullet$ is unprovable since it cannot be deduced without *Cut*. This application highlights the importance of (Gentzen, 1935a) *cut elimination* (originally called *Hauptsatz*), which says that every LK proof can be reduced to a cut-free one.

Theorem 1 (Cut elimination)
For all LK proofs of $\Gamma \vdash \Delta$, *there exists an alternate LK proof of* $\Gamma \vdash \Delta$ *that does not contain any use of the Cut rule.*

Corollary 1 (Consistency)
There is no LK proof of $\bullet \vdash \bullet$.

The simplest cases of cut eliminations case are when an *Ax* axiom is cut with a proof $\mathscr{D}$ of $\Gamma \vdash A, \Delta$ or $\mathscr{E}$ of $\Gamma, A \vdash \Delta$. This particular maneuver does not add anything interesting to the nature of the underlying proof, and so correspondingly eliminating the cut should just give the same proof back unchanged, as we can see in both cases:

$$\dfrac{\begin{array}{c}\mathscr{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{array} \quad \dfrac{}{A \vdash A}\,Ax}{\Gamma \vdash A, \Delta}\,Cut \Longrightarrow \begin{array}{c}\mathscr{D}\\ \vdots\\ \Gamma \vdash A, \Delta\end{array} \qquad \dfrac{\dfrac{}{A \vdash A}\,Ax \quad \begin{array}{c}\mathscr{E}\\ \vdots\\ \Gamma, A \vdash \Delta\end{array}}{\Gamma, A \vdash \Delta}\,Cut \Longrightarrow \begin{array}{c}\mathscr{E}\\ \vdots\\ \Gamma, A \vdash \Delta\end{array}$$

Notice here that cutting an axiom with both $\mathscr{D}$ and $\mathscr{E}$ does not change the sequent in either conclusion, which comes from the precise way that *Cut* merges the side propositions in the two premises. For $\mathscr{D}$, the extra consequence A coming from the axiom $A \vdash A$ replaces the cut A in exactly the right position, and likewise for $\mathscr{E}$. If *Cut* put the propositions of its conclusion in any other order, then we would need to exchange the result of one or both of the above steps with XL and XR to put them back into the right order.

The rest of the proof of cut elimination can be divided into two main parts: the *logical* steps and the *structural* steps. The logical steps of cut elimination consider the cases when we have a cut between two proof trees ending in the left and right rules for the same connective occurring in the same proposition, and show how to rewrite the proof into a new one that does not mention that particular connective. The structural steps of cut elimination handle all the other cases where we do not have a left and right introduction for the same proposition facing one another in a cut. These steps involve rewriting the structure of the proof and propagating the rules until the relevant logical steps can take over. The final ingredient is to ensure that this procedure for eliminating cuts always gives a definite result, and does not spin off into an infinite regress.

Logical cut elimination steps

Notice how different inference rules of LK treat the division of extraneous hypotheses and consequences among multiple premises differently. On the one hand, rules like $\wedge R$ and $\vee L$ duplicate the side propositions Γ and Δ from the conclusion to both premises. On the other hand, rules like *Cut* and $\supset L$ merge different side propositions from the two premises into the common conclusion, creating an ordering between them during the merge. Why are these particular rules given in such different styles, and why is the particular merge order chosen? One way to understand the impact of these details is to look at the interaction between the logical and structural rules during cut elimination, so let us examine a few exemplary steps of the cut elimination procedure when logical rules meet each other.

First, consider what happens when compatible $\wedge R$ and $\wedge L_1$ introductions, with premises $\mathscr{D}_1$, $\mathscr{D}_2$, and $\mathscr{E}$, respectively, meet in a *Cut*:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash A, \Delta\end{array}\mathscr{D}_1 \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash B, \Delta\end{array}\mathscr{D}_2}{\Gamma \vdash A \wedge B, \Delta}\wedge R \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma', A \vdash \Delta'\end{array}\mathscr{E}}{\Gamma', A \wedge B \vdash \Delta'}\wedge L_1}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut}}{} \quad\Longrightarrow\quad \dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash A, \Delta\end{array}\mathscr{D}_1 \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma', A \vdash \Delta'\end{array}\mathscr{E}}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut}$$

Reducing this cut involves selecting the appropriate premise $\mathscr{D}_1$ of the $\wedge R$ introduction so that it can meet with the single premise of $\wedge L_1$. The number of cuts are not reduced by this step, but instead the active proposition $A \wedge B$ of the cut has been reduced to A, which (non-trivially) justifies why this step is making progress in the cut elimination procedure.

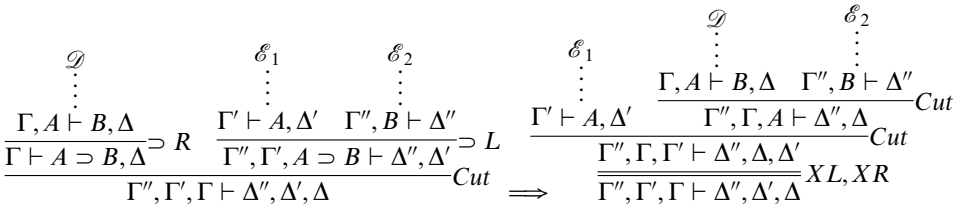
Not every cut-elimination step winds up so neatly organized, unfortunately, and sometimes the result is necessarily out of order and must be corrected. For example, consider the following reduction step of a *Cut* between compatible $\neg R$ and $\neg L$ inferences with premises $\mathscr{D}$ and $\mathscr{E}$, respectively:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, A \vdash \Delta\end{array}\mathscr{D}}{\Gamma \vdash \neg A, \Delta}\neg R \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma' \vdash A, \Delta'\end{array}\mathscr{E}}{\Gamma', \neg A \vdash \Delta'}\neg L}{\Gamma', \Gamma \vdash \Delta', \Delta}\mathit{Cut}}{} \quad\Longrightarrow\quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma' \vdash A, \Delta'\end{array}\mathscr{E} \quad \dfrac{\begin{array}{c}\vdots\\ \Gamma, A \vdash \Delta\end{array}\mathscr{D}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\mathit{Cut}}{\Gamma', \Gamma \vdash \Delta', \Delta}XL, XR$$

Here, the *Cut* we get from reducing the proposition $\neg A$ to A results in a sequent that is out of order compared to the conclusion we started with. Thus, we need to re-order the sequent with some number of XL and XR exchanges to restore the original conclusion. The fact that reducing a negation introduction cut inverts the order of propositions comes from the inherent inversion of negation: there's no obvious way to prevent this scenario by modifying *Cut*.

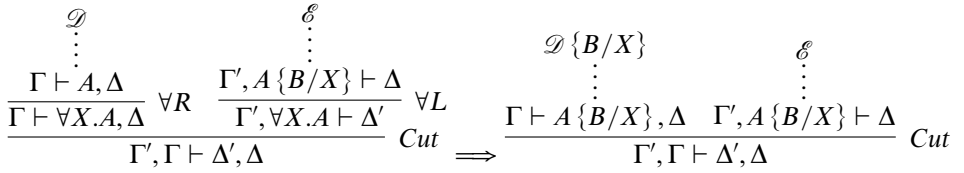
A similar re-ordering occurs with implication, where a *Cut* between compatible $\supset R$ and $\supset L$ inferences, with premises $\mathscr{D}$, $\mathscr{E}_1$, and $\mathscr{E}_2$, can be reduced as

follows:



Here, we start with the side-propositions of $\mathscr{E}_1$ and $\mathscr{E}_2$ merged together with $\supset L$, but after reducing the *Cut*, $\mathscr{D}$ lies in between the two of them, so the conclusion must be re-ordered to match the original. The need to place $\mathscr{D}$ in the middle comes from the fact that its concluding sequent has A on the left and B on the right, so our only available cuts must correspondingly place $\mathscr{E}_1$ to the left and $\mathscr{E}_2$ to the right, no matter how they are nested.

Finally, we can see how the free variable side conditions on the $\forall R$ and $\exists L$ rules play a key role in cut elimination. For example, consider the following reduction step of a cut between compatible $\forall R$ and $\forall L$ inferences with $\mathscr{D}$ and $\mathscr{E}$, respectively:



Notice that in order to make a direct cut between $\mathscr{D}$ and $\mathscr{E}$, we need to substitute B for X in $\mathscr{D}$ to make the two sides match up properly. The fact that X does not occur free in $\Gamma \vdash \Delta$ means that after substitution, both Γ and Δ remain unchanged in the conclusion of the proof. If instead X appeared free somewhere in Γ or Δ, then the logical cut elimination step for $\forall$ would change the conclusion that ruins the result of the procedure. As we saw previously, without the side conditions the $\forall$ and $\exists$ quantifiers are equivalent, which lets us derive a proof of the contradictory sequent $\bullet \vdash \bullet$ that is ruled out by cut elimination. So the side conditions on the $\forall R$ and $\exists L$ rules are not just a useful aid to cut elimination, but are crucial to the entire endeavor.

Structural cut elimination steps

The logical steps may be the primary focus of cut elimination, but there are still more cases they do not cover. In particular, what happens when one of the weakening, contraction, or exchange rules immediately precedes a cut? The full cut elimination procedure must also account for the structural steps in which a cut is forced to interact with a structural rule.

The most straightforward structural step of cut elimination handles the case of weakening adding an unused proposition right before its cut. Such a cut is eliminated by deleting the partner premise of the cut. For example, for the *WL* rule which adds

the unused hypothesis A in the cut, we can discard the proof of A as follows:

$$\dfrac{\mathscr{D} \quad \dfrac{\mathscr{E}}{\Gamma' \vdash \Delta'}}{\dfrac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut}\ WL \quad \Longrightarrow \quad \dfrac{\mathscr{E}}{\dfrac{\Gamma' \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}}\ WL, WR, XR$$

Dually, a cut of an unused consequence A introduced by WR can be eliminated by discarding the other premise that uses A as a hypothesis. Note that in the case where *both* premises to the cut end in a weakening, both of these dual steps can sensibly apply, leading to a potential non-deterministic choice in the cut elimination procedure.

The structural step for contraction is similar, but one premise is duplicated rather than discarded. For example, for the CL rule which merges two duplicate hypotheses in the cut, we can duplicate the proof of A as follows:

$$\dfrac{\mathscr{D} \quad \dfrac{\mathscr{E}}{\Gamma', A, A \vdash \Delta'}}{\dfrac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut}\ CL \quad \Longrightarrow \quad \dfrac{\mathscr{D} \quad \dfrac{\mathscr{D} \quad \mathscr{E}}{\dfrac{\Gamma \vdash A, \Delta \quad \Gamma', A, A \vdash \Delta'}{\dfrac{\Gamma', A, \Gamma \vdash \Delta', \Delta}{\Gamma', \Gamma, A \vdash \Delta', \Delta}\ XL}\ Cut}}{\dfrac{\Gamma', \Gamma, \Gamma \vdash \Delta', \Delta, \Delta}{\Gamma', \Gamma \vdash \Delta', \Delta}}\ CL, CR, XR$$

And the dual structural step involving CR is symmetric to the above. As before with weakening, in the case where the cut proposition is contracted on both the left and right, there is a non-deterministic choice of which structural step to apply.

The trickiest structural rules to accommodate during cut elimination are the exchange rules. By reordering the sequent, these can have the effect of moving the active proposition of interest in rules like Cut or the logical rules, so it is held on the inside of the sequent (next to $\vdash$). To get around this issue, we can handle exchange by generalizing the Cut rule to allow for the cut proposition to appear anywhere in the sequent as follows:

$$\dfrac{\Gamma \vdash \Delta_1, A, \Delta_2 \quad \Gamma_2', A, \Gamma_1' \vdash \Delta'}{\Gamma_2', \Gamma_1', \Gamma \vdash \Delta', \Delta_1, \Delta_2}\ CutX$$

Note that this generalization from Cut to $CutX$ does not change that sequents can be proved: Cut is an instance of $CutX$ and $CutX$ is derivable as a combination of a Cut and potentially many XLs and XRs. However, the more general form of $CutX$ lets us express a cut elimination step where the exchange rules are folded into the cut. In the case, where the cut proposition is exchanged with XL we have the step

$$\dfrac{\mathscr{D} \quad \dfrac{\mathscr{E}}{\Gamma_2', A, B, \Gamma_1' \vdash \Delta'}}{\dfrac{\Gamma \vdash \Delta_1, A, \Delta_2 \quad \Gamma_2', B, A, \Gamma_1' \vdash \Delta'}{\Gamma_2', \Gamma_1', \Gamma \vdash \Delta', \Delta_1, \Delta_2}\ CutX}\ XL \quad \Longrightarrow \quad \dfrac{\mathscr{D} \quad \mathscr{E}}{\dfrac{\Gamma \vdash \Delta_1, A, \Delta_2 \quad \Gamma_2', A, B, \Gamma_1' \vdash \Delta'}{\Gamma_2', \Gamma_1', \Gamma \vdash \Delta', \Delta_1, \Delta_2}\ CutX}$$

and the step for XR is symmetric to the above.

So the *CutX* rule makes the structural steps for exchange trivial. However, this generalization of cut means that there are many more cases to consider. Because the cut proposition may not be the active proposition on the inside of the sequent, it may happen that the rules immediately proceeding the *CutX* have nothing to do with the cut. In these cases, we need yet more structural rules that commute a cut with other rules when they do not interact with one another. As an example of structural commutation, we could have the following weakening on the left of A followed by a cut of C, which is reduced as follows:

$$\dfrac{\dfrac{\mathscr{D}}{\Gamma \vdash \Delta_1, C, \Delta_2} \quad \dfrac{\dfrac{\mathscr{E}}{\Gamma_2', C, \Gamma_1' \vdash \Delta'}}{\Gamma_2', C, \Gamma_1', A \vdash \Delta'}\, WL}{\Gamma_2', \Gamma_1', A, \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, CutX \quad \Longrightarrow \quad \dfrac{\dfrac{\mathscr{D}}{\Gamma \vdash \Delta_1, C, \Delta_2} \quad \dfrac{\mathscr{E}}{\Gamma_2', C, \Gamma_1' \vdash \Delta'}}{\Gamma_2', \Gamma_1', \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, CutX}{\Gamma_2', \Gamma_1', A, \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, WL, XL$$

As an example of logical commutation, we could have a conjunction introduction of $A \wedge B$ on the left followed by a cut of C, which is reduced like so

$$\dfrac{\dfrac{\mathscr{D}}{\Gamma \vdash \Delta_1, C, \Delta_2} \quad \dfrac{\dfrac{\mathscr{E}}{\Gamma_2', C, \Gamma_1', A \vdash \Delta'}}{\Gamma_2', C, \Gamma_1', A \wedge B \vdash \Delta'}\, \wedge L_1}{\Gamma_2', \Gamma_1', A \wedge B, \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, CutX \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\mathscr{D}}{\Gamma \vdash \Delta_1, C, \Delta_2} \quad \dfrac{\mathscr{E}}{\Gamma_2', C, \Gamma_1', A \vdash \Delta'}}{\Gamma_2', \Gamma_1', A, \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, CutX}{\Gamma_2', \Gamma_1', \Gamma, A \vdash \Delta', \Delta_1, \Delta_2}\, XL}{\Gamma_2', \Gamma_1', \Gamma, A \wedge B \vdash \Delta', \Delta_1, \Delta_2}\, \wedge L_1}{\Gamma_2', \Gamma_1', A \wedge B, \Gamma \vdash \Delta', \Delta_1, \Delta_2}\, XL$$

There are many more such commuting steps for all the cases where the cut proposition is not the active one next to the turnstile, each of which push the cut up into the premis(es) of the proceeding rule similar to the above examples.

3.3 Logical duality

Another application of sequent calculi is to study the *dualities* of logic through the deep symmetries of the system (Gentzen, 1935b). The turnstile of entailment ($\vdash$) provides the pivot of duality separating left from right and true from false. Logical duality in the LK sequent calculus expresses a relationship between the connectives that follows De Morgan's laws about the way negation distributes over conjunction and disjunction:

$$\neg(A \wedge B) \dashv\vdash (\neg A) \vee (\neg B)$$
$$\neg(A \vee B) \dashv\vdash (\neg A) \wedge (\neg B)$$

Here, we interpret the equivalence relation $A \dashv\vdash B$ as the mutual provability of A and B: that both $A \vdash B$ and $B \vdash A$ are provable. Focusing on the opposite roles of the left and right sides of a sequent, we can immediately observe that the introduction rules of conjunction and disjunction from Figure 4 are mirror images

Duality of sequents:

$$(\Gamma \vdash \Delta)^\perp \triangleq \Delta^\perp \vdash \Gamma^\perp \qquad\qquad (A_1, \ldots, A_n)^\perp \triangleq A_n^\perp, \ldots, A_1^\perp$$

Duality of propositions:

$$\begin{array}{ll} (X)^\perp \triangleq X & (\neg A)^\perp \triangleq \neg(A^\perp) \\ (A \wedge B)^\perp \triangleq (A^\perp) \vee (B^\perp) & (A \vee B)^\perp \triangleq (A^\perp) \wedge (B^\perp) \\ (A \supset B)^\perp \triangleq (B^\perp) - (A^\perp) & (B - A)^\perp \triangleq (A^\perp) \supset (B^\perp) \\ (\forall X.A)^\perp \triangleq \exists X.(A^\perp) & (\exists X.A)^\perp \triangleq \forall X.(A^\perp) \end{array}$$

Fig. 5. Duality in the LK sequent calculus.

of one another by flipping the sequents across their turnstile. Similarly, $\supset$ and $-$ are dual to one another as well as both the $\forall$ and $\exists$ quantifiers, and negation is its own dual, with both $\neg R$ and $\neg L$ reflecting the same inference flipped about entailment.

Since each connective has a dual counterpart, we can express the duality of sequent calculus proofs—for every LK proof $\mathscr{D}$ of a sequent:

$$\begin{array}{c} \mathscr{D} \\ \vdots \\ A_n, \ldots, A_2, A_1 \vdash B_1, B_2, \ldots, B_m \end{array}$$

there is a dual proof $\mathscr{D}^\perp$ of the dual sequent:

$$\begin{array}{c} \mathscr{D}^\perp \\ \vdots \\ B_m^\perp, \ldots, B_2^\perp, B_1^\perp \vdash A_1^\perp, A_2^\perp, \ldots, A_n^\perp \end{array}$$

The duality relation on judgments and propositions, is given in Figure 5. Note that the duality operation $A^\perp$ may be understood as taking the negation of the proposition, $\neg A$, and pushing the negation inward all the way using the De Morgan laws, until a proposition variable X is reached (Gentzen, 1935b).

Theorem 2 (Logical duality)

For any LK proof $\mathscr{D}$ of the sequent $\Gamma \vdash \Delta$, there exists a dual proof $\mathscr{D}^\perp$ of the dual sequent $\Delta^\perp \vdash \Gamma^\perp$.

Note that Gentzen did not consider the dual counterpart to implication as a connective, as we do, but rather eliminated implication from the system by encoding it in terms of disjunction and negation given above for the purposes of establishing duality.

Due to the natural syntactic symmetry of the LK sequent calculus, logical duality comes from an exchange between left and right: left rules mirror right rules and hypotheses to the left of entailment mirror consequences to the right. Thus, establishing logical duality in the sequent calculus follows from a straightforward induction on the structure of proofs, working from the bottom conclusion up to the axioms.

Non-contradiction and excluded middle

To illustrate how the left and right sides of proofs get swapped, consider the case when the bottom conclusion is inferred from a use of the $\wedge R$ rule:

$$\dfrac{\begin{array}{cc}\mathscr{D} & \mathscr{E} \\ \vdots & \vdots \\ \Gamma \vdash A, \Delta & \Gamma \vdash B, \Delta\end{array}}{\Gamma \vdash A \wedge B, \Delta}\ \wedge R$$

Then by the inductive hypothesis, we get a proof $\mathscr{D}^{\perp}$ of $(\Gamma \vdash A, \Delta)^{\perp} \triangleq \Delta^{\perp}, A^{\perp} \vdash \Gamma^{\perp}$ and a proof $\mathscr{E}^{\perp}$ of $(\Gamma \vdash B, \Delta)^{\perp} \triangleq \Delta^{\perp}, B^{\perp} \vdash \Gamma^{\perp}$, from which we can deduce $(\Gamma \vdash A \wedge B, \Delta)^{\perp} \triangleq \Delta^{\perp}, (A^{\perp}) \vee (B^{\perp}) \vdash \Gamma^{\perp}$ by $\vee L$:

$$\dfrac{\begin{array}{cc}\mathscr{D}^{\perp} & \mathscr{E}^{\perp} \\ \vdots & \vdots \\ \Delta^{\perp}, A^{\perp} \vdash \Gamma^{\perp} & \Delta^{\perp}, B^{\perp} \vdash \Gamma^{\perp}\end{array}}{\Delta^{\perp}, A^{\perp} \vee B^{\perp} \vdash \Gamma^{\perp}}\ \vee L$$

The duality of proofs in the LK sequent calculus means that if a proposition A is true, so that we have a proof of $\bullet \vdash A$, then its dual must be false, so that we have a proof of $A^{\perp} \vdash \bullet$. Analogously, if a proposition A is false, then its dual must be true. For example, consider the following general proof of the law of non-contradiction, stating that $A \wedge (\neg A)$ is false:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \wedge (\neg A) \vdash A}\ \wedge L_1}{A \wedge (\neg A), \neg A \vdash \bullet}\ \neg L}{A \wedge (\neg A), A \wedge (\neg A) \vdash \bullet}\ \wedge L_1}{A \wedge (\neg A) \vdash \bullet}\ CL$$

Duality gives a general proof of the law of excluded middle, stating that $A \vee (\neg A)$ is true:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \vdash A \vee (\neg A)}\ \vee R_1}{\bullet \vdash \neg A, A \vee (\neg A)}\ \neg R}{\bullet \vdash A \vee (\neg A), A \vee (\neg A)}\ \vee R_1}{\bullet \vdash A \vee (\neg A)}\ CR$$

The existence of a general proof for the law of excluded middle ($\bullet \vdash A \vee (\neg A)$) is forced by Theorem 2 because we have a general proof for the law of non-contradiction ($A \wedge (\neg A) \vdash \bullet$).

4 A core calculus

The logics of natural deduction and the sequent calculus are rather different from one another. As previously discussed in Section 2, one major point of distinction between the two styles of logic is that natural deduction is right-handed, favoring truth to the exclusion of falsehood, whereas the sequent calculus is ambidextrous,

directly handing truth and falsehood on both the left and right sides of entailment. That means that the sequent calculus does not correspond to the λ-calculus the same way that natural deduction does. So what might a programming language based on a sequent calculus like LK look like?

Before delving into the entirety of LK, let us first consider a *core* language shown in Figure 6, (Herbelin, 2005) $\mu\tilde{\mu}$-calculus, that corresponds to the core part of LK and lies at the heart of several sequent-based languages (Curien and Herbelin, 2000; Wadler, 2003; Munch-Maccagnoni, 2009; Curien and Munch-Maccagnoni, 2010), including the one we will explore. Notice that the language of types in this core lacks any logical connectives, so that the only types are uninterpreted variables X, Y, Z, *etc.* The $\mu\tilde{\mu}$-calculus is a bare language for describing only input, output, and interactions: the types on the right side of a sequent describe the outputs of a program and the types on the left side of a sequent describe the inputs of a program. When the two opposite sides come together—when the opposed forces of input and output meet—we have an interaction that sparks computation. Note that the type system brings out an aspect of deduction that was implicit in the sequent calculus: the role of a distinguished *active* proposition that is currently under consideration. For example, in the $\wedge R$ rule from Figure 4, we are currently trying to prove the proposition $A \wedge B$, so it is considered the active proposition of the conclusion $\Gamma \vdash A \wedge B, \Delta$.

By putting attention on at most one active proposition, we get three classifications of sequents: active on the right, active on the left, or passive (without an active proposition on either side). These three forms of sequents likewise classify three different forms of $\mu\tilde{\mu}$ expressions that might be part of a program:

- An active sequent on the right ($\Gamma \vdash v : A | \Delta$) describes a *term* v that sends information of type A as its output (that is, v is a *producer* of type A).
- An active sequent on the left ($\Gamma | e : A \vdash \Delta$) describes a *co-term* e that receives information of type A as its input (that is, e is a *consumer* of type A).
- A passive sequent ($c : (\Gamma \vdash \Delta)$) describes a *command* c that is an executable program capable of running on its own without any distinguished input or output.

In each case, the environments Γ and Δ describe any additional passive (non-active) inputs and outputs to an expression by specifying the types of *free variables* $(x, \dots)$ and *free co-variables* $(\alpha, \dots)$ that expression might reference, respectively. Like in LK, these environments are finite, ordered sequences which may be (1) empty (written $\bullet$), (2) a variable or co-variable paired up with its type (written $x : A$ and $\alpha : A$, respectively), or (3) a concatenation of two sequences (written with a comma as Γ, Γ' and Δ, Δ'). As a further constraint, we stipulate that each variable and co-variable can appear *at most* once in an environment, so that the concatenation of repeated type assignments like $x : A, x : B$ or $\alpha : A, \alpha : B$ is undefined.

The expressions of the $\mu\tilde{\mu}$-calculus come from the axiom and cut rules of LK plus an additional pair of *activation* rules AR and AL. The Ax rule of LK is divided into two separate rules in $\mu\tilde{\mu}$: the VR rule creates a term by just referring to a variable available from its environment, and similarly the VL rule creates a co-term

$$
\begin{aligned}
A,B,C \in \mathit{Type} &::= X \\
v \in \mathit{Term} &::= x \mid \mu\alpha.c \\
e \in \mathit{CoTerm} &::= \alpha \mid \tilde{\mu}x.c \\
c \in \mathit{Command} &::= \langle v \| e \rangle \\
\Gamma \in \mathit{InputEnv} &::= x_1 : A_1, \ldots, x_n : A_n \\
\Delta \in \mathit{OutputEnv} &::= \alpha_1 : A_1, \ldots, \alpha_n : A_n \\
\mathit{Sequent} &::= (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta) \mid c : (\Gamma \vdash \Delta)
\end{aligned}
$$

Core rules:

$$\frac{}{x : A \vdash x : A \mid} \; VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A} \; VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \; AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \; AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : A \vdash \Delta'}{\langle v \| e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)} \; Cut$$

Structural rules:

$$\frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma \vdash \alpha : A, \Delta)} \; WR \qquad \frac{c : (\Gamma \vdash \Delta)}{c : (\Gamma, x : A \vdash \Delta)} \; WL$$

$$\frac{c : (\Gamma \vdash \beta : A, \alpha : A, \Delta)}{c\{\alpha/\beta\} : (\Gamma \vdash \alpha : A, \Delta)} \; CR \qquad \frac{c : (\Gamma, x : A, y : A \vdash \Delta)}{c\{x/y\} : (\Gamma, x : A \vdash \Delta)} \; CL$$

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A, \beta : B, \Delta')}{c : (\Gamma \vdash \Delta, \beta : B, \alpha : A, \Delta')} \; XR \qquad \frac{c : (\Gamma', y : B, x : A, \Gamma \vdash \Delta)}{c : (\Gamma', x : A, y : B, \Gamma \vdash \Delta)} \; XL$$

$$\frac{\Gamma \vdash v : C \mid \Delta}{\Gamma \vdash v : C \mid \alpha : A, \Delta} \; WR \qquad \frac{\Gamma \vdash v : C \mid \Delta}{\Gamma, x : A \vdash v : C \mid \Delta} \; WL$$

$$\frac{\Gamma \vdash v : C \mid \beta : A, \alpha : A, \Delta}{\Gamma \vdash v\{\alpha/\beta\} : C \mid \alpha : A, \Delta} \; CR \qquad \frac{\Gamma, x : A, y : A \vdash v : C \mid \Delta}{\Gamma, x : A \vdash v\{x/y\} : C \mid \Delta} \; CL$$

$$\frac{\Gamma \vdash v : C \mid \Delta, \alpha : A, \beta : B, \Delta'}{\Gamma \vdash v : C \mid \Delta, \beta : B, \alpha : A, \Delta'} \; XR \qquad \frac{\Gamma', y : B, x : A, \Gamma \vdash v : C \mid \Delta}{\Gamma', x : A, y : B, \Gamma \vdash v : C \mid \Delta} \; XL$$

$$\frac{\Gamma \mid e : C \vdash \Delta}{\Gamma \mid e : C \vdash \alpha : A, \Delta} \; WR \qquad \frac{\Gamma \mid e : C \vdash \Delta}{\Gamma, x : A \mid e : C \vdash \Delta} \; WL$$

$$\frac{\Gamma \mid e : C \vdash \beta : A, \alpha : A, \Delta}{\Gamma \mid e\{\alpha/\beta\} : C \vdash \alpha : A, \Delta} \; CR \qquad \frac{\Gamma, x : A, y : A \mid e : C \vdash \Delta}{\Gamma, x : A \mid e\{x/y\} : C \vdash \Delta} \; CL$$

$$\frac{\Gamma \mid e : C \vdash \Delta, \alpha : A, \beta : B, \Delta'}{\Gamma \mid e : C \vdash \Delta, \beta : B, \alpha : A, \Delta'} \; XR \qquad \frac{\Gamma', y : B, x : A, \Gamma \mid e : C \vdash \Delta}{\Gamma', x : A, y : B, \Gamma \mid e : C \vdash \Delta} \; XL$$

Fig. 6. $\mu\tilde{\mu}$: The core language of the sequent calculus.

by referring to a co-variable. The *Cut* rule connects a term and co-term that are waiting to send and receive information of the same type, so that the output of the term is forwarded to the co-term as input (and dually, the input of the co-term is drawn from the output of the term). Finally, the *activation* rules *AR* and *AL* pick a

particular (co-)variable from the environment of a command to activate by creating an output or input abstraction, respectively. Intuitively, if the variable x stands for an unknown input in a command c, then the input abstraction $\tilde{\mu}x.c$ is a co-term that, when given a place to draw information, will bind that location to the input channel x while running c. Dually, if the co-variable α stands for an unknown output in a command c, then the output abstraction $\mu\alpha.c$ is a term that, when given a place to send information, will bind that location to the output channel α while running c.

Structural rules and static scope

To give a full account of the static semantics of the $\mu\tilde{\mu}$-calculus, we need to address the issue of how the structural properties of the sequent calculus are represented. For instance, the co-term $\tilde{\mu}z.\langle x \| \alpha \rangle$ should have the type $x : X \mid \tilde{\mu}z.\langle x \| \alpha \rangle : Y \vdash \alpha : X$, but the core typing rules alone are not enough. Rather, the structural properties of sequents (weakening, contraction, and exchange) define the meaning of static variables and co-variables.

Similar to LK, the structural properties of sequents in $\mu\tilde{\mu}$ can be expressed by explicit structural rules that allow for a single (co-)variable to appear any number of times in an expression. The full collection of these structural scoping rules are shown in Figure 6, which corresponds one-for-one with the structural rules of Gentzen's LK sequent calculus over each form of $\mu\tilde{\mu}$ expression. The weakening rules say that even if a free (co-)variable is in scope in an expression, it does not have to be referenced, as in the co-term $\tilde{\mu}z.\langle x \| \alpha \rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X \mid}\,VR \quad \dfrac{}{\mid \alpha : X \vdash \alpha : X}\,VL}{\langle x \| \alpha \rangle : (x : X \vdash \alpha : X)}\,Cut}{\langle x \| \alpha \rangle : (x : X, z : Y \vdash \alpha : X)}\,WL}{x : X \mid \tilde{\mu}z.\langle x \| \alpha \rangle : Y \vdash \alpha : X}\,AL$$

The contraction rules say that a free (co-)variable can be referenced an additional time by renaming two distinct (co-)variables into one, as in the command $\langle \mu\delta.\langle y \| \alpha \rangle \| \tilde{\mu}z.\langle y \| \alpha \rangle \rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : X \vdash y : X \mid}\,VR \quad \dfrac{}{\mid \beta : X \vdash \beta : X}\,VL}{\langle y \| \beta \rangle : (y : X \vdash \beta : X)}\,Cut}{\langle y \| \beta \rangle : (y : X \vdash \delta : Y, \beta : X)}\,WR}{y : X \vdash \mu\delta.\langle y \| \beta \rangle : Y \mid \beta : X}\,AR \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X \mid}\,VR \quad \dfrac{}{\mid \alpha : X \vdash \alpha : X}\,VL}{\langle x \| \alpha \rangle : (x : X \vdash \alpha : X)}\,Cut}{\langle x \| \alpha \rangle : (x : X, z : Y \vdash \alpha : X)}\,WL}{x : X \mid \tilde{\mu}z.\langle x \| \alpha \rangle : Y \vdash \alpha : X}\,AL}{\langle \mu\delta.\langle y \| \beta \rangle \| \tilde{\mu}z.\langle x \| \alpha \rangle \rangle : (x : X, y : X \vdash \alpha : X, \beta : X)}\,Cut}{\langle \mu\delta.\langle x \| \beta \rangle \| \tilde{\mu}z.\langle x \| \alpha \rangle \rangle : (x : X \vdash \alpha : X, \beta : X)}\,CL$$

Finally, the exchange rules say that the order of the (co-)variables in scope does not matter. Notice that none of these rules are syntactically visible in their expression. Unlike the axiom, activation, and cut rules that only apply to expressions starting with a very specific form like a (co-)variable, abstraction, or interaction, the structural rules could potentially apply to expressions of any form so they are not directed by syntax.

The form of the structural rules in Figure 6 shows the role of the active sequents for controlling the impact of structural rules on the principle type of interest. In particular, the type of the term in the sequent $\Gamma \vdash v : A \mid \Delta$ and the type of the co-term in $\Gamma \mid e : A \vdash \Delta$ cannot be subject to weakening, contraction, or exchange. Instead, structural rules only apply the (co-)variables in the environment, meaning that if we want to contract or weaken the type of a (co-)term with one of its free (co-)variables, we must first associate its input or output with another (co-)variable by forming a command like so

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash v : A \mid \alpha : A \quad \overline{\mid \beta : A \vdash \beta : A}\ VL}{\langle v \| \beta \rangle : (\Gamma \vdash \beta : A, \alpha : A, \Delta)}\ Cut}{\langle v \| \alpha \rangle : (\Gamma \vdash \alpha : A, \Delta)}\ CR}{\Gamma \vdash \mu\alpha.\langle v \| \alpha \rangle : A \mid \Delta}\ AR \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash v : A \mid \Delta \quad \overline{\mid \alpha : A \vdash \alpha : A}\ VL}{\langle v \| \alpha \rangle : (\Gamma \vdash \alpha : A, \Delta)}\ Cut}{\langle v \| \alpha \rangle : (\Gamma \vdash \beta : B, \alpha : A, \Delta)}\ WR}{\Gamma \vdash \mu\beta.\langle v \| \alpha \rangle : B \mid \alpha : A, \Delta}\ AR$$

and symmetrically for co-terms. Likewise, if we want to exchange the current active type of a (co-)term with another one in the environment, we need to take a similar detour through a command that explicitly switches the primary input or output channel as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash v : A \mid \beta : B, \Delta \quad \overline{\mid \alpha : A \vdash \alpha : A}\ VL}{\langle v \| \alpha \rangle : (\Gamma \vdash \alpha : A, \beta : B, \Delta)}\ Cut}{\langle v \| \alpha \rangle : (\Gamma \vdash \beta : B, \alpha : A, \Delta)}\ XR}{\Gamma \vdash \mu\beta.\langle v \| \alpha \rangle : B \mid \alpha : A, \Delta}\ AR \qquad \dfrac{\dfrac{\overline{x : A \vdash x : A \mid}\ VR \qquad \Gamma, y : B \mid e : A \vdash \Delta}{\langle x \| e \rangle : (\Gamma, y : B, x : A \vdash \Delta)}\ Cut}{\langle x \| e \rangle : (\Gamma, x : A, y : B \vdash \Delta)}\ XL}{\Gamma, x : A \mid \tilde{\mu} y.\langle x \| e \rangle : B \vdash \Delta}\ AL$$

4.1 Two dual substitutions

Having examined the *static* properties of the $\mu\tilde{\mu}$-calculus—its syntax and types—we still need to consider the *dynamic* properties of $\mu\tilde{\mu}$, to explain what it means to run a program. To answer the question "what is computation in the sequent calculus?" we turn to cut elimination (previously mentioned in Section 3.2) that outlines a method of reducing commands as the main unit of computation. In other words, computation in $\mu\tilde{\mu}$ is the behavior that results from cutting together a compatible producer and consumer in a command, so that they may meaningfully interact with one another. In the bare $\mu\tilde{\mu}$-calculus with no logical connectives, we can only have three forms of commands: a cut between (co-)variables $\langle x \| \alpha \rangle$, a cut with an output abstraction $\langle \mu\alpha.c \| e \rangle$, and a cut with an input abstraction $\langle v \| \tilde{\mu} x.c \rangle$. In the first case, a command $\langle x \| \alpha \rangle$ represents a basic *final* state that can reduce no further, and even though its typing derivation contains a *Cut*, it is a trivial sort of cut that corresponds more closely to a passive version of LK's *Ax*:

$$\dfrac{\overline{x : A \vdash x : A \mid}\ VR \quad \overline{\mid \alpha : A \vdash \alpha : A}\ VL}{\langle x \| \alpha \rangle : (x : A \vdash \alpha : A)}\ Cut$$

In the second two cases, we can capture the meaning of input and output abstractions via *substitution* —written as $\{v/x\}$ and $\{e/\alpha\}$—in the style of β reduction

in the λ-calculus as illustrated by the following μ and $\tilde{\mu}$ operational rules:

$$(\mu) \qquad \langle \mu\alpha.c \| e \rangle \mapsto c\,\{e/\alpha\} \qquad (\tilde{\mu}) \qquad \langle v \| \tilde{\mu}x.c \rangle \mapsto c\,\{v/x\}$$

The $\tilde{\mu}$ reduction step substitutes the term v for the variable x introduced by an input abstraction, distributing it into the command c to the points where it is referenced. The μ reduction step is the mirror image, which substitutes a co-term e for a co-variable α introduced by an output abstraction. Both of these substitution operations must take care to avoid capturing the *free variables* of the substituted (co-)terms as in the λ-calculus. Definitions of capture-avoiding substitution and the free variables found in (co-)terms and commands (denoted by $FV(c)$, $FV(V)$, $FV(e)$) can be found in Section 6.

The μ and $\tilde{\mu}$ substitution steps eliminate a cut, but how do they correspond to cut elimination in LK? The procedure described in Section 3.2 does not appear to use a substitution operation, only a collection of small, local manipulations of cuts. As it turns out, the substitutions used by the μ and $\tilde{\mu}$ rules correspond to the structural steps of LK cut elimination, except performed all at once instead of incrementally. Cuts of a passive proposition can be viewed as a substitution for a (co-)variable and the substitution operation itself exhaustively applies the steps that commute logical and structural rules with passive cuts. For example, the commutation of an active cut with a passive cut corresponds to the equation

$$\langle v \| e \rangle\,\{v'/x\} = \langle v\,\{v'/x\} \| e\,\{v'/x\} \rangle$$

that defines one case of substitution. In effect, this transports a passive cut to all of its active positions within a proof as one step, and the activation rules for μ- and $\tilde{\mu}$-abstractions explicitly signal that a cut is passive. And as additional logical rules are added later in Section 5, similar commutations are uniformly characterized by the standard rules of substitution. The cut elimination steps for weakening and contraction on the active proposition of a cut then correspond to properties that the substitution operation satisfies

$$c\,\{e/\alpha\} = c \quad (\alpha \notin FV(c)) \qquad (c\,\{\alpha/\beta\})\,\{e/\alpha\} = c\,\{e/\beta\}\,\{e/\alpha\} \quad (\alpha \notin FV(e))$$
$$c\,\{v/x\} = c \quad (x \notin FV(c)) \qquad (c\,\{x/y\})\,\{v/x\} = c\,\{v/y\}\,\{v/x\} \quad (x \notin FV(v))$$

or in other words, substituting for a (co-)variable that is never referenced does nothing, and substituting for a merged pair of (co-)variables is the same as substituting for both individually.

We can now give two different formalizations of the dynamic semantics of the $\mu\tilde{\mu}$-calculus, each of which have their own distinct purpose. The first is the *operational semantics* of $\mu\tilde{\mu}$ that explains exactly step-by-step how to execute a command by performing repetitions of the μ and $\tilde{\mu}$ operational rules (*i.e.,* the reflexive, transitive closure of the $\mapsto$ relation written $\twoheadmapsto$). Note that, unlike in the λ-calculus, the next step of the operational semantics is immediately obvious in the $\mu\tilde{\mu}$-calculus and needs no search to identify: the next step of a command is always found at the top-level if there is one.

The second is the *rewriting theory* of $\mu\tilde{\mu}$ that provides more opportunities for reductions, including performing a step before it would normally occur during

the execution of a command (as in a pre-processing pass or optimization) or additional reductions that do not occur during execution (*i.e.,* are not one of the operational rules defining $\mapsto$) but preserve its behavior nonetheless. Single-step rewriting is denoted by $\rightarrow$ and allows the reductions to apply in any context (*i.e.,* $\rightarrow$ is compatibly closed), and the multi-step rewriting is denoted by $\twoheadrightarrow$ (*i.e.,* $\twoheadrightarrow$ is the reflexive, transitive closure of $\rightarrow$). For the $\mu\tilde{\mu}$-calculus, single-step rewriting includes the μ and $\tilde{\mu}$ operational rules given above, as well as some additional rules. In particular, the following η_μ and $\eta_{\tilde{\mu}}$ reductions that eliminate trivial output and input abstractions are allowed, because they do not change the extensional behavior of the (co-)terms:

$$(\eta_\mu) \qquad \mu\alpha.\langle v \| \alpha \rangle \rightarrow v \quad (\alpha \notin FV(v)) \qquad (\eta_{\tilde{\mu}}) \qquad \tilde{\mu}x.\langle x \| e \rangle \rightarrow e \quad (x \notin FV(e))$$

In other words, the term that sends the output of v to α only to forward that information along as its own output is the same as v itself. Dually, the co-term that binds its input to x only to forward that information along to another co-term e can be written more simply as just e. In all, the rewriting theory of $\mu\tilde{\mu}$ is formed by repetitions of $\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}$ reductions in any context, and is defined in more detail in Section C3.

4.2 The fundamental dilemma of computation

Unfortunately, the aforementioned operational semantics for $\mu\tilde{\mu}$ is *non-deterministic*, to the point where program execution may take completely divergent and unrelated paths. The non-determinism of the $\mu\tilde{\mu}$-calculus corresponds to the fact that the cut elimination for LK included critically non-deterministic choices between structural rules. The phenomenon is embodied by the fundamental conflict between input and output abstractions, as shown by the two dual μ and $\tilde{\mu}$ reductions for performing substitution:

$$c_1\,\{(\tilde{\mu}x.c_2)/\alpha\} \leftarrow_\mu \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \mapsto_{\tilde{\mu}} c_2\,\{(\mu\alpha.c_1)/x\}$$

Both the term $\mu\alpha.c_1$ and co-term $\tilde{\mu}x.c_2$ are fighting for control in the above command, and either one may win. The non-deterministic outcome of this conflict is exemplified in the case where neither α nor x are referenced in their respective commands by weakening

$$c_1 \leftarrow_\mu \langle \mu\_.c_1 \| \tilde{\mu}\_.c_2 \rangle \mapsto_{\tilde{\mu}} c_2$$

showing that programs may produce different results each time they are run, since the same starting point may step to two different and completely arbitrary commands. This form of divergent reduction paths is called a *critical pair* and is evidence that the rewriting theory is not *confluent*. A confluent system guarantees that reductions can be applied in any order and still reach the same result. From the perspective of programming language semantics, this type of non-determinism can be undesirable since it makes it impossible to predict a single definitive result of a program since there may be multiple incompatible results depending on the choices made during execution. If we want to regain properties like confluence or determinism, which are enjoyed by the λ-calculus, then some of these freedoms must be curtailed.

$$V \in \mathit{Value}_v ::= x \qquad\qquad E \in \mathit{CoValue}_v ::= e$$

Operational rules:

$$(\mu_v) \qquad \langle \mu\alpha.c \| E \rangle \mapsto c\{E/\alpha\} \qquad\qquad (\tilde{\mu}_v) \qquad \langle V \| \tilde{\mu}x.c \rangle \mapsto c\{V/x\}$$

Rewriting rules:

$$(\eta_\mu) \qquad \mu\alpha.\langle v \| \alpha \rangle \to v \quad (\alpha \notin FV(v)) \qquad\qquad (\eta_{\tilde{\mu}}) \qquad \tilde{\mu}x.\langle x \| e \rangle \to e \quad (x \notin FV(e))$$

$$\frac{c \mapsto_{\mu_v\tilde{\mu}_v} c'}{c \to_{\mu_v\tilde{\mu}_v} c'}\ \mu_v\tilde{\mu}_v$$

Fig. 7. The call-by-value (v) semantics for the core $\mu\tilde{\mu}_v$-calculus.

In order to recover determinism for the sequent calculus, Curien and Herbelin (2000) observed that we only need to choose an *evaluation strategy* that deterministically picks the next step to take by giving priority to one reduction over the other:

> *Call-by-value consists in giving priority to the μ redexes, while call-by-name gives priority to the $\tilde{\mu}$ redexes.*

Prioritization between the two opposed sides means that there must be some potential μ or $\tilde{\mu}$ redexes that we could reduce but choose not to, thereby yielding priority to the other side of the command. From another viewpoint, choosing a priority between the two sides of a command is the same thing as choosing a restriction on the terms and co-terms that can be substituted by the μ and $\tilde{\mu}$ rules. And reversing directions, choosing which terms and co-terms are substitutable by μ and $\tilde{\mu}$ reductions also chooses the evaluation strategy.

Reflecting the above observation back to the calculus, we can restore determinacy to the operational semantics and confluence to the rewriting theory by making the substitution rules strategy-aware: $\tilde{\mu}$ only substitutes *values* for variables and μ only substitutes *co-values* for co-variables. In other words, the decision of which values and co-values are substitutable is enough information to determine an evaluation strategy in the $\mu\tilde{\mu}$-calculus. To get call-by-value reduction, we can restrict the notion of value to exclude output abstractions and leave co-values unrestricted, thereby giving priority to the μ redexes as shown in Figure 7. Dually for call-by-name reduction, we can restrict the notion of co-value to exclude input abstractions and leave values unrestricted, thereby giving priority to the $\tilde{\mu}$ redexes as shown in Figure 8. Notice that in any case, the η_μ and $\eta_{\tilde{\mu}}$ reductions are not affected by the restrictions on (co-)values, because they do no substitution and are sound under any choice of evaluation strategy. These restrictions on substitution give us exactly (Curien and Herbelin, 2000) notions of the call-by-value and call-by-name, which restores determinacy and confluence to the semantics of $\mu\tilde{\mu}$. Excluding a (co-)term from the collection of (co-)values effectively prioritizes it by blocking opposing reductions, whereas including a (co-)term as a (co-)value diminishes its priority since it can be deleted or duplicated by substitution.

$$V \in Value_n ::= v \qquad\qquad E \in CoValue_n ::= \alpha$$

Operational rules:

$$(\mu_n) \qquad \langle \mu\alpha.c \| E \rangle \mapsto c\{E/\alpha\} \qquad\qquad (\tilde{\mu}_n) \qquad \langle V \| \tilde{\mu}x.c \rangle \mapsto c\{V/x\}$$

Rewriting rules:

$$(\eta_\mu) \qquad \mu\alpha.\langle v \| \alpha \rangle \to v \qquad (\alpha \notin FV(v)) \qquad\qquad (\eta_{\tilde{\mu}}) \qquad \tilde{\mu}x.\langle x \| e \rangle \to e \qquad (x \notin FV(e))$$

$$\frac{c \mapsto_{\mu_n\tilde{\mu}_n} c'}{c \to_{\mu_n\tilde{\mu}_n} c'}\ \mu_n\tilde{\mu}_n$$

Fig. 8. The call-by-name (n) semantics for the core $\mu\tilde{\mu}_n$-calculus.

5 The dual calculi

With the core $\mu\tilde{\mu}$ language firmly in place, we can now enrich it with additional programming constructs that correspond to the logical elements—the connectives and logical rules—of Gentzen's LK sequent calculus. The syntax and typing rules for these extra logical constructs are shown in Figure 9, which extends the core $\mu\tilde{\mu}$-calculus from Figure 6. To help syntactically distinguish terms from co-terms, we use the notational convention throughout that round parentheses are the grouping brackets for terms, and square brackets are the grouping brackets for co-terms. The correspondence with LK is that by erasing program-level constructs of a typing derivation and replacing type constructors with the corresponding logical connectives (replacing $\to$ with $\supset$, $\times$ with $\wedge$, *etc.*,), we get an LK proof derivation: there is an LK proof derivation of $\Gamma \vdash \Delta$ if and only if there is a typing derivation of $c : (\Gamma' \vdash \Delta')$ for the Γ', Δ' corresponding to Γ, Δ and some command c, and similarly for typed terms $(\Gamma' \vdash v : A' \mid \Delta')$ and co-terms $(\Gamma' \mid e : A' \vdash \Delta')$.

This language combines both (Curien and Herbelin, 2000) $\overline{\lambda}\mu\tilde{\mu}$-calculus (the portion associated with implication) and (Wadler, 2003) dual calculus (the portion associated with conjunction, disjunction, and negation which was directly inspired by $\overline{\lambda}\mu\tilde{\mu}$) into a single calculus corresponding to all of the simply-typed LK sequent calculus. Furthermore, the quantifiers of LK are interpreted as a sequent calculus version of system F (Reynolds, 1983; Girard *et al.*, 1989): universal quantification ($\forall$) acts as an abstraction over types analogous to implication, and existential quantification ($\exists$) is the mirror image of $\forall$. We refer to this combined language here as the "dual calculi" because, as we will soon see, the language is the basis for two different but highly related calculi that exhibit dual computational behavior to one another.

Since the right introduction rules for logical connectives are shared by both natural deduction and the sequent calculus, the dual calculi terms for creating results of product, sum, and function types have the same form as in the λ-calculus. Units are introduced by a constant, (), products are introduced by pairing, (v, v'), sums are introduced by injection, $\mathsf{in}_1(v)$ and $\mathsf{in}_2(v)$, and functions are introduced by λ-abstractions, $\lambda x.v$. Additionally, the terms for creating results of universally quantified types are Λ-abstractions, $\Lambda X.v$, as in system F, and the results of

$$A,B,C \in \mathit{Type} ::= X \mid 1 \mid 0 \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$v \in \mathit{Term} ::= x \mid \mu\alpha.c \mid () \mid (v,v) \mid \mathsf{in}_1(v) \mid \mathsf{in}_2(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid e \cdot v \mid \Lambda X.v \mid B@v$$
$$e \in \mathit{CoTerm} ::= \alpha \mid \tilde{\mu}x.c \mid [] \mid \mathsf{out}_1[e] \mid \mathsf{out}_2[e] \mid [e,e] \mid \mathsf{not}[v] \mid v \cdot e \mid \tilde{\lambda}\alpha.e \mid B@e \mid \tilde{\Lambda}X.e$$
$$c \in \mathit{Command} ::= \langle v \| e \rangle$$

Core rules:

$$\frac{}{x : A \vdash x : A \mid} \; VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A} \; VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \; AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \; AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : A \vdash \Delta'}{\langle v \| e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)} \; Cut$$

Structural rules:

The same weakening (WL, WR), contraction (CL, CR), and exchange (XL, XR) as in Fig. 6.

Logical rules:

$$\frac{}{\Gamma \vdash () : 1 \mid \Delta} \; 1R \qquad \text{no } 1L \text{ rule} \qquad \text{no } 0R \text{ rule} \qquad \frac{}{\Gamma \mid [] : 0 \vdash \Delta} \; 0L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma \vdash (v,v') : A \times B \mid \Delta} \times R \qquad \frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \mathsf{out}_1[e] : A \times B \vdash \Delta} \times L_1 \qquad \frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \mathsf{out}_2[e] : A \times B \vdash \Delta} \times L_2$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \mathsf{in}_1(v) : A + B \mid \Delta} + R_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \mathsf{in}_2(v) : A + B \mid \Delta} + R_2 \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e,e'] : A + B \vdash \Delta} + L$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \mid \Delta} \neg R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \mathsf{not}[v] : \neg A \vdash \Delta} \neg L$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta} \to R \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma', \Gamma \mid v \cdot e : A \to B \vdash \Delta', \Delta} \to L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma', \Gamma \vdash e \cdot v : A - B \mid \Delta', \Delta} - R \qquad \frac{\Gamma \mid e : A \vdash \alpha : B, \Delta}{\Gamma \mid \tilde{\lambda}\alpha.e : A - B \vdash \Delta} - L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A \mid \Delta} \forall R \qquad \frac{\Gamma \mid e : A\{B/X\} \vdash \Delta}{\Gamma \mid B@e : \forall X.A \vdash \Delta} \forall L$$

$$\frac{\Gamma \vdash v : A\{B/X\} \mid \Delta}{\Gamma \vdash B@v : \exists X.A \mid \Delta} \exists R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\Lambda}X.e : \exists X.A \vdash \Delta} \exists L$$

Fig. 9. The syntax and types for the dual calculi.

existentially quantified types are "masked" terms, $B@v$, that hide some occurrences of the type B in the underlying term v from being visible from the outside. In contrast, the left introduction rules of the sequent calculus are distinct from the right elimination rules of natural deduction, so the difference between the λ-calculus and the dual calculi really appears when results are used.

Instead of function application, the left implication introduction $\to L$ builds a co-term that represents a *call-stack*. If v is a term that produces a result of type

A, and e is a co-term that consumes a result of type B, then the call-stack $v \cdot e$ is a co-term that works with a function value of type $A \to B$ by feeding it v as an argument and sending the returned result to e. For example, given that $x_1 : A_1$, $x_2 : A_2$, $x_3 : A_3$, and $\beta : B$, then the call-stack $x_1 \cdot [x_2 \cdot [x_3 \cdot \beta]]$ is expecting to consume a function of type $A_1 \to (A_2 \to (A_3 \to B))$:

$$\dfrac{\dfrac{}{x_1 : A_1 \vdash x_1 : A_1 \mid}\,VR \quad \dfrac{\dfrac{}{x_2 : A_2 \vdash x_2 : A_2 \mid}\,VR \quad \dfrac{\dfrac{}{x_3 : A_3 \vdash x_3 : A_3 \mid}\,VR \quad \dfrac{}{\mid \beta : B \vdash \beta : B}\,VL}{x_3 : A_3 \mid x_3 \cdot \beta : A_3 \to B \vdash \beta : B}\,{\to}L}{x_3 : A_3, x_2 : A_2 \mid x_2 \cdot x_3 \cdot \beta : A_2 \to A_3 \to B \vdash \beta : B}\,{\to}L}{x_3 : A_3, x_2 : A_2, x_1 : A_1 \mid x_1 \cdot x_2 \cdot x_3 \cdot \beta : A_1 \to A_2 \to A_3 \to B \vdash \beta : B}\,{\to}L$$

Like the common notational convention in the simply-typed λ-calculus that the function type constructor associates to the right, so that $A_1 \to A_2 \to A_3 \to B = A_1 \to (A_2 \to (A_3 \to B))$, we adopt a similar notational convention that the call stack constructor associates to the right, so that $x_1 \cdot x_2 \cdot x_3 \cdot \beta = x_1 \cdot [x_2 \cdot [x_3 \cdot \beta]]$.

The left introductions for the other type constructors follow a similar pattern, with each one building a co-term that expects to consume a value of that type. There are two left conjunction introductions corresponding to the two projections out of a product. If e_1 is a co-term that consumes a value of type A, then $\times L_1$ builds the co-term $\mathsf{out}_1[e_1]$ that works with a value of type $A \times B$ by projecting out the first element of the product and sending it to e_1 when needed (and similarly for the second projection $\mathsf{out}_2[e_2]$ built by $\times L_2$). If e_1 and e_2 are co-terms that consume values of type A and B, respectively, then $+L$ builds the co-term $[e_1, e_2]$ that works with a value of type $A + B$ by checking its constructor: an injection of the form $\mathsf{in}_1(v_1)$ has the value of v_1 sent to e_1 as needed, and likewise an injection of the form $\mathsf{in}_2(v_2)$ has the value of v_2 sent to e_2 as needed. The co-term for the empty type 0 is a constant, [], which observes an impossible term that cannot produce any output. The co-term for $\forall L$ is similar to the call stacks of $\to L$, so that if e is a co-term that consumes a value at the particular type $A\{B/X\}$, then $B@e$ works with a value of the general type $\forall X.A$ by first specializing the polymorphic value and then passing it along to e. Perhaps the most unusual co-term comes from $\exists L$, but this is just the mirror image of the $\forall R$ term. If e is a co-term that consumes a value of type A, containing a generic type variable X, then $\exists L$ gives the abstracted co-term $\tilde{\Lambda} X.e$ that works with a value of type $\exists X.A$ by instantiating X with the value's hidden type before passing the underlying value to e.

The two type constructors that are not typically found in the λ-calculus, but sometimes in a sequent calculus like LK or the dual calculi, are negation and subtraction. The negation type $\neg A$ represents an inversion between producers and consumers—terms and co-terms—during computation. Intuitively, negation expresses a form of *continuations*: a term of type $\neg A$ is actually a consumer of A. The right negation introduction allows terms to contain consumers, so that if e is a co-term expecting an input of type A then $\neg R$ builds the term $\mathsf{not}(e)$. Dually, the left negation introduction allows co-terms to contain producers, so that if v is a term expecting to output a result of type A then $\neg L$ builds the co-term $\mathsf{not}[v]$. When a

$$V \in \mathit{Value}_v ::= x \mid (V,V) \mid \mathsf{in}_1(V) \mid \mathsf{in}_2(V) \mid \mathsf{not}(e) \mid \lambda\, x.v \mid e \cdot V \mid \Lambda X.v \mid B@V$$
$$E \in \mathit{CoValue}_v ::= e$$

Operational rules:

$$(\beta_v^{\times}) \quad \langle (V_1,V_2) \| \mathsf{out}_i[E] \rangle \mapsto \langle V_i \| E \rangle \qquad (\beta_v^{+}) \quad \langle \mathsf{in}_i(V) \| [E_1,E_2] \rangle \mapsto \langle V \| E_i \rangle$$
$$(\beta_v^{\neg}) \quad \langle \mathsf{not}(e) \| \mathsf{not}[v] \rangle \mapsto \langle v \| e \rangle$$
$$(\beta_v^{\rightarrow}) \quad \langle \lambda\, x.v \| V \cdot E \rangle \mapsto \langle v\{V/x\} \| E \rangle \qquad (\beta_v^{-}) \quad \langle E \cdot V \| \tilde{\lambda}\, \alpha\, .e \rangle \mapsto \langle V \| e\{E/\alpha\} \rangle$$
$$(\beta_v^{\forall}) \quad \langle \Lambda X.v \| B@E \rangle \mapsto \langle v\{B/X\} \| E \rangle \qquad (\beta_v^{\exists}) \quad \langle B@V \| \tilde{\Lambda} X.e \rangle \mapsto \langle V \| e\{B/X\} \rangle$$

Rewriting rules:

$$\frac{c \mapsto_{\beta_v} c'}{c \rightarrow_{\beta_v} c'}\ \beta_v$$

Fig. 10. The β semantics for the call-by-value (v) half of the dual calculi.

negated term and co-term meet each other in a command, the inversion is undone so that their underlying components change places and continue the interaction. The subtraction type $A - B$ is dual to a function type: whereas a function represents an answer that depends on another answer, a subtraction represents a question that depends on another question. The left subtraction introduction allows for consumer transformations which are mirror images of λ-abstractions, so that the $-L$ rule builds a co-term of the form $\tilde{\lambda}\alpha.e$ of type $A - B$ when e is a consumer of A that references a co-variable α of type B. On the other side, the right subtraction introduction pairs up a producer and a consumer, so that if v produces an A result and e consumes a B result then the $-R$ rule builds the term $e \cdot v$ of type $A - B$. Subtraction gives another way for continuations to appear in terms, so that a result of type $A - B$ yields both an answer (A) and a question (B) at the same time.

The above intuition on the dynamic meaning of types in the dual calculi can be codified into an operational semantics. Recall from Section 4.2 that the semantics of the core $\mu\tilde{\mu}$-calculus was split in two to restore determinacy and confluence: one semantics corresponding to call-by-value and the other to call-by-name. Likewise, there are two deterministic operational semantics and two confluent rewriting theories for the dual calculi, so that the same language bears two different calculi (hence the name). Since both semantics of the core $\mu\tilde{\mu}$-calculus are already given in Figures 7 and 8, we only need to suitably expand the notions of value and co-value to accommodate the new (co-)term introductions and explain the logical steps of cut elimination (referred to by the common name β) that occur when two opposed introduction forms of the same type meet in a command. The call-by-value β operational rules are given in Figure 10 and the call-by-name β operational rules are given in Figure 11, both of which extend the core semantics from Figures 7 and 8, respectively. Thus, we end up with the call-by-value $\mu_v\tilde{\mu}_v\beta_v$ operational semantics and $\mu_v\tilde{\mu}_v\eta_\mu\eta_{\tilde{\mu}}\beta_v$ rewriting theory as well as the call-by-name $\mu_n\mu_n\beta_n$ operational semantics and $\mu_n\mu_n\eta_\mu\eta_{\tilde{\mu}}\beta_n$ rewriting theory for the dual calculi. The $\beta^{\times}$, β^{+}, and $\beta^{\neg}$ rules come from (Wadler, 2003) dual calculus, whereas the $\beta^{\rightarrow}$ rule is inspired by (Curien and Munch-Maccagnoni, 2010) revision of the $\overline{\lambda}\mu\tilde{\mu}$-calculus. The reason this

$$V \in \mathit{Value}_n ::= v$$

$$E \in \mathit{CoValue}_n ::= \alpha \mid \mathsf{out}_1[E] \mid \mathsf{out}_2[E] \mid [E,E] \mid \mathsf{not}[v] \mid v \cdot E \mid \tilde{\lambda}\alpha\,.e \mid B@E \mid \tilde{\Lambda}X.e$$

Operational rules:

$$(\beta_n^{\times}) \quad \langle (V_1,V_2) \| \mathsf{out}_i[E] \rangle \mapsto \langle V_i \| E \rangle \qquad (\beta_n^{+}) \quad \langle \mathsf{in}_i(V) \| [E_1,E_2] \rangle \mapsto \langle V \| E_i \rangle$$

$$(\beta_n^{\neg}) \quad \langle \mathsf{not}(e) \| \mathsf{not}[v] \rangle \mapsto \langle v \| e \rangle$$

$$(\beta_n^{\rightarrow}) \quad \langle \lambda\, x.v \| V \cdot E \rangle \mapsto \langle v\{V/x\} \| E \rangle \qquad (\beta_v^{-}) \quad \langle E \cdot V \| \tilde{\lambda}\alpha\,.e \rangle \mapsto \langle V \| e\{E/\alpha\} \rangle$$

$$(\beta_n^{\forall}) \quad \langle \Lambda X.v \| B@E \rangle \mapsto \langle v\{B/X\} \| E \rangle \qquad (\beta_n^{\exists}) \quad \langle B@V \| \tilde{\Lambda}X.e \rangle \mapsto \langle V \| e\{B/X\} \rangle$$

Rewriting rules:

$$\frac{c \mapsto_{\beta_n} c'}{c \rightarrow_{\beta_n} c'}\ \beta_n$$

Fig. 11. The β semantics for the call-by-name (n) half of the dual calculi.

differs from the original $\beta^{\rightarrow}$ rule (Curien and Herbelin, 2000) for the $\bar{\lambda}\mu\tilde{\mu}$-calculus,

$$\langle \lambda x.v \| v' \cdot e \rangle \mapsto \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle \qquad x \notin FV(e)$$

will show up later on in Section 5.2.

Notice that, like in the core $\mu\tilde{\mu}$-calculus, the form of the operational β rules are the same in both semantics, so that the only difference is the definition of *value* and *co-value* referred to in those rules. The rule of thumb is that a β rule only applies when an introductory value and co-value interact in a command. For example, the call-by-value $\beta_v^{\times}$ rule will only project from a pair value to extract a component that is also a value. These restrictions are captured in the call-by-value definition of value that admits only "simple" terms and hereditarily excludes complex terms like $\mu\alpha.c$ (representing an arbitrarily complex computation before yielding a result on α) from the values of product and sum types, which matches the behavior of products and sums in strict functional languages like ML. However, there is no such restriction on co-terms in the call-by-value operational semantics, and as such any co-term counts as a co-value. Dually, the call-by-name $\beta_n^{\times}$ rule will only project out of a pair when it is needed by a projection co-value to send that component to the underlying co-value. These restrictions are captured in the call-by-name definition of co-value that admits only "strict" co-terms and hereditarily excludes complex co-terms like $\tilde{\mu}x.c$ (representing an arbitrarily complex computation before demanding a result for x) from the co-values of product and sum types. However, there is no restriction on terms in the call-by-name operational semantics, and as such any term counts as a value.

5.1 Untyped fixed points and infinite loops

It's worthwhile to mention that although the dual calculi are primarily seen as typed languages, their semantics do not use any type information to run commands. We can therefore execute untyped commands as well as typed ones, which of course creates the possibility of getting stuck at fatal type errors. Untyped commands

also open up the possibility of running general recursive programs, which can be encoded in a similar manner as in the λ-calculus without any additional features of the language. For example, Curry's untyped fixed-point Y combinator in the λ-calculus:

$$Y \triangleq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

can be analogously defined in the dual calculi using functions as

$$Y \triangleq \lambda f.\mu\alpha.\langle \lambda x.\mu\beta.\langle f \| \mu\gamma.\langle x \| x \cdot \gamma\rangle \cdot \beta\rangle \| (\lambda x.\mu\beta.\langle f \| \mu\gamma.\langle x \| x \cdot \gamma\rangle \cdot \beta\rangle) \cdot \alpha\rangle$$

The two share analogous behavior: in the λ-calculus $Y\ f = f\ (Y\ f)$ and in the dual calculi $\langle Y \| f \cdot \alpha\rangle = \langle f \| \mu\beta.\langle Y \| f \cdot \beta\rangle \cdot \alpha\rangle$. Also analogous to the non-terminating untyped term $\Omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x)$ in the λ-calculus, the dual calculi both have non-terminating untyped commands, which can be written using functions or more simply with negation:

$$\Omega \triangleq \langle \mathsf{not}(\tilde{\mu}x.\langle x \| \mathsf{not}[x]\rangle) \| \mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]\rangle$$

For example, in the call-by-name operational semantics, we have the following infinite execution of Ω:

$$\begin{aligned}
\Omega &\triangleq \langle \mathsf{not}(\tilde{\mu}x.\langle x \| \mathsf{not}[x]\rangle) \| \mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]\rangle \\
&\mapsto_{\beta_n^-} \langle \mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle \| \tilde{\mu}x.\langle x \| \mathsf{not}[x]\rangle\rangle \\
&\mapsto_{\tilde{\mu}_n} \langle \mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle \| \mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]\rangle \\
&\mapsto_{\mu_n} \langle \mathsf{not}(\mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]) \| \mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]\rangle \\
&\mapsto_{\beta_n^-} \langle \mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle \| \mathsf{not}[\mu\alpha.\langle \mathsf{not}(\alpha) \| \alpha\rangle]\rangle \\
&\mapsto_{\mu_n} \cdots
\end{aligned}$$

Note that encoding general recursion in the untyped sequent calculus requires *some* logical connective, like negation or implication. The core $\mu\tilde{\mu}$-calculus gives a more restrained language of binders and substitution that does not express general recursion even in the untyped calculus, where general (and non-confluent) μ- and $\tilde{\mu}$-reduction is still *strongly normalizing* (Polonovski, 2004)—that is, there are no infinite sequences of $\mu\tilde{\mu}$-reductions. This fact is in contrast with the untyped λ-calculus that can express general recursion, because β-reduction is not strongly normalizing in the untyped calculus.

5.2 Focusing on computation

There is a problem lurking in the β-based operational semantics for the dual calculi. Consider how we would evaluate the projection $\mathrm{out}_1((f\ 1), 2)$ in a call-by-value functional language like ML. First, we would compute the application $f\ 1$ to construct the pair value, then we would compute the out_1 projection of that pair and extract the value returned by $f\ 1$ as the result of the expression. However, if we represent this program as the following command in the call-by-value dual calculus, where α stands for the empty or top-level context that is implicit in the functional

expression:

$$\langle((\mu\beta.\langle f \| 1 \cdot \beta\rangle), 2) \| \mathrm{out}_1[\alpha]\rangle$$

we find that no operational rule matches this command, so we are stuck! This is not just a problem with the call-by-value operational semantics. The command

$$\langle(1, 2) \| \mathrm{out}_1[\tilde{\mu}x.\langle 0 \| \alpha\rangle]\rangle$$

that corresponds to the expression **let** $x = \mathrm{out}_1(1, 2)$**in** 0 in a functional language, is also stuck in the call-by-name operational semantics.

This is clearly an undesirable situation that breaks the connection between the λ-calculus and dual calculi—we should not get stuck on such commands with unfinished computation in introduction forms—so something needs to be done to refocus the attention in a command to the next step of computation. As it stands now in the dual calculi, we either have too many programs with unexplained behavior, or too few behaviors for executing programs. Correspondingly, there are two general techniques to remedy prematurely stuck commands and restore the connection between λ-calculus and the dual calculi:

(1) The *static* approach (Curien and Herbelin, 2000) removes the superfluous parts of the syntax that cause β reduction to get stuck, but are not necessary to express all the same computations as the original language.
(2) The *dynamic* approach (Wadler, 2003) adds the necessary extra steps to the operational semantics that *lift* buried computations to the top of the command, so that they are exposed and may take over control of the computation.

Both of these techniques can be viewed as an application of an idea called *focusing* (Andreoli, 1992; Laurent, 2002) from proof search at different points in a programs life—either at "run time" or at "compile time"—to make sure that the call-by-value and call-by-name semantics are complete without missing out on any essential capabilities of the language.

Static focusing

For the static method of focusing, consider which syntactic patterns could lead to β-stuck commands. In the call-by-value command above, $\langle((\mu\beta.\langle f \| 1 \cdot \beta\rangle), 2) \| \mathrm{out}_1[\alpha]\rangle$, the problem is that a pair with a non-value component (namely the first one) is interacting with a projection co-value. Because the pair does not have values for both components, the $\beta_v^\times$ operational step does not apply. Dually, the call-by-name command above, $\langle(1, 2) \| \mathrm{out}_1[\tilde{\mu}x.\langle 0 \| \alpha\rangle]\rangle$, puts a pair value in interaction with a projection that has a non-co-value component. Because the projection does not contain a co-value, the $\beta_n^\times$ operational step does not apply. After examining all the β_v rules, we see that the call-by-value β_v operational semantics is only equipped to deal with certain introduction forms containing values (namely the pairing $\times R$, injection $+R$, and masking $\exists R$ terms as well as calling $\to L$ co-terms). Similarly, the call-by-name β_n operational semantics is only equipped to deal with certain

introduction co-terms containing co-values (namely the projection $\times L$, matching $+L$, and calling $\to L$, and specializing $\forall L$ co-terms).

We can rule out the problematic commands via static focusing by limiting ourselves to a sub-syntax of the dual calculi. However, since each operational semantics (both call-by-value and call-by-name) have difficulty with different parts of the syntax, static focusing effectively splits the language in two: one sub-syntax for each evaluation strategy. For call-by-value, we must bake in the notion of values into the syntax and restrict the $\times R$, $+R$, $-R$, $\exists R$, and $\to L$ inference rules appropriately. Doing so gives us the LKQ sub-calculus (Curien and Herbelin, 2000) shown in Figure 12. Notice how the sub-syntax of LKQ no longer lets us write terms like $\mathsf{in}_1(\mu\beta.c)$ and $(\mu\beta_1.c_1, \mu\beta_2.c_2)$ because a μ-abstraction is not a value; instead such terms can only be written with intermediate bindings as $\mu\alpha.\langle\mu\beta.c \| \tilde{\mu}x.\langle \mathsf{in}_1(x) \| \alpha\rangle\rangle$ and $\mu\alpha.\langle\mu\beta_1.c_1 \| \tilde{\mu}x_1.\langle\mu\beta_2.c_2 \| \tilde{\mu}x_2.\langle(x_1, x_2) \| \alpha\rangle\rangle\rangle$ reminiscent of *CPS* (Reynolds, 1993). The statically focused calculus makes the call-by-value evaluation order more explicit in the program itself. Similar such restrictions are imposed on the term constructors of subtraction and existential types, and on the argument of function call stacks. Dually for call-by-name, we must bake in the notion of co-values into the syntax and restrict the $\times L$, $+L$, $\to L$, $\forall L$, and $-R$ inference rules appropriately, giving the LKT sub-calculus shown in Figure 13. Notice that the sub-syntax of LKT instead prevents us from writing co-terms like $\mathrm{out}_1[\tilde{\mu}y.c]$ and $[\tilde{\mu}y_1.c_1, \tilde{\mu}y_2.c_2]$ because a $\tilde{\mu}$-abstraction is not a co-value; instead such co-terms can only be written indirectly as $\tilde{\mu}x.\langle\mu\alpha.\langle x \| \mathrm{out}_1[\alpha]\rangle \| \tilde{\mu}y.c\rangle$ and $\tilde{\mu}x.\langle\mu\alpha_1.\langle\mu\alpha_2.\langle x \| [\alpha_1, \alpha_2]\rangle \| \tilde{\mu}y_2.c_2\rangle \| \tilde{\mu}y_1.c_1\rangle$, which is symmetric to the explicit bindings forced by LKQ.

The associated type systems separate the restricted notions of (co-)values from general (co-)terms through a new form of *focused* sequent with a stricter sense of active formula held in a *stoup* (Girard, 1991). LKQ introduces *values* in the focus of a stoup on the right ($\Gamma \vdash V : A \;;\; \Delta$) and LKT introduces *co-values* in the focus of a stoup on the left ($\Gamma \;;\; E : A \vdash \Delta$), which differ from the more general sequents ($\Gamma \vdash v : A \mid \Delta$ and $\Gamma \mid e :\vdash \Delta$) that allow for any (co-)term and not just (co-)values. The new form of sequent calls for additional focusing structural rules *FR* (in LKQ) and *FL* (in LKT), which acknowledge that every value is a term and every co-value is a co-term. However, the reverse of the focusing rules—which would say that every (co-)term is a (co-)value—are omitted in LKQ and LKT because they would collapse the distinction between (co-)values and (co-)terms enforced by the stoup. As a consequence of the fact that the stoup is one-way, the focus of the inference rules is forcibly maintained through type checking: working bottom-up, once a (co-)value is in focus in the stoup, our active attention cannot move to any other type in the sequent, thereby limiting the derivations we can build on top of focused sequents.

As it turns out (Curien and Munch-Maccagnoni, 2010), distinguishing (co-)values in type systems like LKQ and LKT correspond with the technique of focusing in proof theory developed by Andreoli (1992), Girard (1993; 2001), and Laurent (2002). If we erase the program-level annotations of typing derivations, the active position in a sequent disappears but the one-way stoup remains giving us two different sub-logics of the LK sequent calculus corresponding to LKQ in Figure 14 and LKT

$$A,B,C \in \mathit{Type} ::= X \mid 1 \mid 0 \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$v \in \mathit{Term} ::= V \mid \mu\alpha\,.c$$
$$V \in \mathit{Value} ::= x \mid () \mid (V,V) \mid \mathsf{in}_1(V) \mid \mathsf{in}_2(V) \mid \mathsf{not}(e) \mid \lambda\, x.v \mid e \cdot V \mid \Lambda X.v \mid B@V$$
$$e \in \mathit{CoTerm} ::= \alpha \mid \tilde{\mu}\, x.c \mid [] \mid \mathsf{out}_1[e] \mid \mathsf{out}_2[e] \mid [e,e] \mid \mathsf{not}[v] \mid V \cdot e \mid \tilde{\lambda}\alpha\,.e \mid B@e \mid \tilde{\Lambda} X.e$$
$$c \in \mathit{Command} ::= \langle v \| e \rangle$$
$$\mathit{Sequent} ::= (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \vdash V : A\,;\Delta) \mid (\Gamma \mid e : A \vdash \Delta) \mid c : (\Gamma \vdash \Delta)$$

Core rules:

$$\frac{}{x : A \vdash x : A\,;}\ VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha\,.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}\, x.c : A \vdash \Delta}\ AL$$

$$\frac{\Gamma \vdash V : A\,;\Delta}{\Gamma \vdash V : A \mid \Delta}\ FR \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : A \vdash \Delta'}{\langle v \| e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ Cut$$

Structural rules:

$$\frac{\Gamma \vdash V : C\,;\Delta}{\Gamma \vdash V : C\,;\alpha : A, \Delta}\ WR \qquad \frac{\Gamma \vdash V : C\,;\Delta}{\Gamma, x : A \vdash V : C\,;\Delta}\ WL$$

$$\frac{\Gamma \vdash V : C\,;\beta : A, \alpha : A, \Delta}{\Gamma \vdash V\{\alpha/\beta\} : C\,;\alpha : A, \Delta}\ CR \qquad \frac{\Gamma, x : A, y : A \vdash V : C\,;\Delta}{\Gamma, x : A \vdash V\{x/y\} : C\,;\Delta}\ CL$$

$$\frac{\Gamma \vdash V : C\,;\Delta, \alpha : A, \beta : B, \Delta'}{\Gamma \vdash V : C\,;\Delta, \beta : B, \alpha : A, \Delta'}\ XR \qquad \frac{\Gamma', y : B, x : A, \Gamma \vdash V : C\,;\Delta}{\Gamma', x : A, y : B, \Gamma \vdash V : C\,;\Delta}\ XL$$

And the same weakening (WL, WR), contraction (CL, CR), and exchange (XL, XR) as in Fig. 6.

Logical rules:

$$\frac{}{\Gamma \vdash () : 1\,;\Delta}\ 1R \qquad \text{no } 1L \text{ rule} \qquad \text{no } 0R \text{ rule} \qquad \frac{}{\Gamma \mid [] : 0 \vdash \Delta}\ 0L$$

$$\frac{\Gamma \vdash V : A\,;\Delta \quad \Gamma \vdash V' : B\,;\Delta}{\Gamma \vdash (V,V') : A \times B\,;\Delta}\ \times R \qquad \frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \mathsf{out}_1[e] : A \times B \vdash \Delta}\ \times L_1 \qquad \frac{\Gamma \mid e : B \vdash \Delta}{\Gamma \mid \mathsf{out}_2[e] : A \times B \vdash \Delta}\ \times L_2$$

$$\frac{\Gamma \vdash V : A\,;\Delta}{\Gamma \vdash \mathsf{in}_1(V) : A + B\,;\Delta}\ +R_1 \qquad \frac{\Gamma \vdash V : B\,;\Delta}{\Gamma \vdash \mathsf{in}_2(V) : A + B\,;\Delta}\ +R_2 \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid e' : B \vdash \Delta}{\Gamma \mid [e,e'] : A + B \vdash \Delta}\ +L$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A\,;\Delta}\ \neg R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid \mathsf{not}[v] : \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda\, x.v : A \to B\,;\Delta}\ \to R \qquad \frac{\Gamma \vdash V : A\,;\Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma, \Gamma' \mid V \cdot e : A \to B \vdash \Delta, \Delta'}\ \to L$$

$$\frac{\Gamma \vdash V : A\,;\Delta \quad \Gamma' \mid e : B \vdash \Delta'}{\Gamma', \Gamma \vdash e \cdot V : A - B\,;\Delta', \Delta}\ -R \qquad \frac{\Gamma \mid e : A \vdash \alpha : B, \Delta}{\Gamma \mid \tilde{\lambda}\alpha\,.e : A - B \vdash \Delta}\ -L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A\,;\Delta}\ \forall R \qquad \frac{\Gamma \mid e : A\{B/X\} \vdash \Delta}{\Gamma \mid B@e : \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash V : A\{B/X\}\,;\Delta}{\Gamma \vdash B@V : \exists X.A\,;\Delta}\ \exists R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\Lambda} X.e : \exists X.A \vdash \Delta}\ \exists L$$

Fig. 12. LKQ: The focalized sub-syntax and types for the call-by-value dual calculus.

$$A,B,C \in \mathit{Type} ::= X \mid 1 \mid 0 \mid A \times B \mid A + B \mid \neg A \mid A \to B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$v \in \mathit{Term} ::= x \mid \mu\alpha.c \mid () \mid (v,v) \mid \mathsf{in}_1(v) \mid \mathsf{in}_2(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid E \cdot v \mid \Lambda X.v \mid B@v$$
$$e \in \mathit{CoTerm} ::= E \mid \tilde{\mu} x.c$$
$$E \in \mathit{CoValue} ::= \alpha \mid [] \mid \mathsf{out}_1[E] \mid \mathsf{out}_2[E] \mid [E,E] \mid \mathsf{not}[v] \mid v \cdot E \mid \tilde{\lambda}\alpha.e \mid B@E \mid \tilde{\Lambda} X.e$$
$$c \in \mathit{Command} ::= \langle v \| e \rangle$$
$$\mathit{Sequent} ::= (\Gamma \vdash v : A \mid \Delta) \mid (\Gamma \mid e : A \vdash \Delta) \mid (\Gamma ; E : A \vdash \Delta) \mid c : (\Gamma \vdash \Delta)$$

Core rules:

$$\frac{}{x : A \vdash x : A ;}\ VR \qquad \frac{}{\mid \alpha : A \vdash \alpha : A}\ VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu} x.c : A \vdash \Delta}\ AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' \mid e : A \vdash \Delta'}{\langle v \| e \rangle : (\Gamma', \Gamma \vdash \Delta', \Delta)}\ \mathit{Cut} \qquad \frac{\Gamma ; E : A \vdash \Delta}{\Gamma \mid E : A \vdash \Delta}\ FL$$

Structural rules:

$$\frac{\Gamma ; E : C \vdash \Delta}{\Gamma ; E : C \vdash \alpha : A, \Delta}\ WR \qquad \frac{\Gamma ; E : C \vdash \Delta}{\Gamma, x : A ; E : C \vdash \Delta}\ WL$$

$$\frac{\Gamma ; E : C \vdash \beta : A, \alpha : A, \Delta}{\Gamma ; E\{\alpha/\beta\} : C \vdash \alpha : A, \Delta}\ CR \qquad \frac{\Gamma, x : A, y : A ; E : C \vdash \Delta}{\Gamma, x : A ; E\{x/y\} : C \vdash \Delta}\ CL$$

$$\frac{\Gamma ; E : C \vdash \Delta, \alpha : A, \beta : B, \Delta'}{\Gamma ; E : C \vdash \Delta, \beta : B, \alpha : A, \Delta'}\ XR \qquad \frac{\Gamma', y : B, x : A, \Gamma ; E : C \vdash \Delta}{\Gamma', x : A, y : B, \Gamma ; E : C \vdash \Delta}\ XL$$

And the same weakening (WL, WR), contraction (CL, CR), and exchange (XL, XR) as in Fig. 6.

Logical rules:

$$\frac{}{\Gamma \vdash () : 1 \mid \Delta}\ 1R \qquad \text{no } 1L \text{ rule} \qquad \text{no } 0R \text{ rule} \qquad \frac{}{\Gamma ; [] : 0 \vdash \Delta}\ 0L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash v' : B \mid \Delta}{\Gamma \vdash (v,v') : A \times B \mid \Delta}\ \times R \qquad \frac{\Gamma ; E : A \vdash \Delta}{\Gamma ; \mathsf{out}_1[E] : A \times B \vdash \Delta}\ \times L_1 \qquad \frac{\Gamma ; E : B \vdash \Delta}{\Gamma ; \mathsf{out}_2[E] : A \times B \vdash \Delta}\ \times L_2$$

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \mathsf{in}_1(v) : A + B \mid \Delta}\ +R_1 \qquad \frac{\Gamma \vdash v : B \mid \Delta}{\Gamma \vdash \mathsf{in}_2(v) : A + B \mid \Delta}\ +R_2 \qquad \frac{\Gamma ; e : A \vdash \Delta \quad \Gamma ; e' : B \vdash \Delta}{\Gamma ; [E,E'] : A + B \vdash \Delta}\ +L$$

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(e) : \neg A \mid \Delta}\ \neg R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma ; \mathsf{not}[v] : \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, x : A \vdash v : B \mid \Delta}{\Gamma \vdash \lambda x.v : A \to B \mid \Delta}\ \to R \qquad \frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' ; E : B \vdash \Delta'}{\Gamma, \Gamma' ; v \cdot E : A \to B \vdash \Delta, \Delta'}\ \to L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma' ; E : B \vdash \Delta'}{\Gamma', \Gamma \vdash e \cdot v : A - B \mid \Delta', \Delta}\ -R \qquad \frac{\Gamma \mid e : A \vdash \alpha : B, \Delta}{\Gamma ; \tilde{\lambda}\alpha.e : A - B \vdash \Delta}\ -L$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \Lambda X.v : \forall X.A \mid \Delta}\ \forall R \qquad \frac{\Gamma ; E : A\{B/X\} \vdash \Delta}{\Gamma ; B@E : \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash v : A\{B/X\} \vdash \Delta}{\Gamma \vdash B@v : \exists X.A \mid \Delta}\ \exists R \qquad \frac{\Gamma \mid e : A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma ; \tilde{\Lambda} X.e : \exists X.A \vdash \Delta}\ \exists L$$

Fig. 13. LKT: The focalized sub-syntax and types for the call-by-name dual calculus.

$$A,B,C \in \mathit{Proposition} ::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid \neg A \mid A \supset B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$\Gamma \in \mathit{Hypothesis} ::= A_1, \ldots, A_n$$
$$\Delta \in \mathit{Consequence} ::= A_1, \ldots, A_n$$
$$\mathit{Sequent} ::= \Gamma \vdash \Delta \mid \Gamma \vdash A \,;\, \Delta$$

Core rules:

$$\frac{}{A \vdash A}\ Ax \qquad \frac{}{A \vdash A \,;}\ Ax_F \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut \qquad \frac{\Gamma \vdash A \,;\, \Delta}{\Gamma \vdash A, \Delta}\ FR$$

Structural rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ WR \qquad \frac{\Gamma \vdash C \,;\, \Delta}{\Gamma \vdash C \,;\, A, \Delta}\ WR_F \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ WL \qquad \frac{\Gamma \vdash C \,;\, \Delta}{\Gamma, A \vdash C \,;\, \Delta}\ WL_F$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ CR \qquad \frac{\Gamma \vdash C \,;\, A, A, \Delta}{\Gamma \vdash C \,;\, A, \Delta}\ CR_F \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ CL \qquad \frac{\Gamma, A, A \vdash C \,;\, \Delta}{\Gamma, A \vdash C \,;\, \Delta}\ CL_F$$

$$\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\ XR \qquad \frac{\Gamma \vdash C \,;\, \Delta, A, B, \Delta'}{\Gamma \vdash C \,;\, \Delta, B, A, \Delta'}\ XR_F \qquad \frac{\Gamma', B, A, \Gamma \vdash \Delta}{\Gamma', A, B, \Gamma \vdash \Delta}\ XL \qquad \frac{\Gamma', B, A, \Gamma \vdash C \,;\, \Delta}{\Gamma', A, B, \Gamma \vdash C \,;\, \Delta}\ XL_F$$

Logical rules:

$$\frac{}{\Gamma \vdash \top \,;\, \Delta}\ \top R \qquad \text{no } \top L \text{ rule} \qquad \text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \vdash \Delta}\ \bot L$$

$$\frac{\Gamma \vdash A \,;\, \Delta \quad \Gamma \vdash B \,;\, \Delta}{\Gamma \vdash A \wedge B \,;\, \Delta}\ \wedge R \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_1 \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge L_2$$

$$\frac{\Gamma \vdash A \,;\, \Delta}{\Gamma \vdash A \vee B \,;\, \Delta}\ \vee R_1 \qquad \frac{\Gamma \vdash B \,;\, \Delta}{\Gamma \vdash A \vee B \,;\, \Delta}\ \vee R_2 \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee L$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A \,;\, \Delta}\ \neg R \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B \,;\, \Delta}\ \supset R \qquad \frac{\Gamma \vdash A \,;\, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma, A \supset B \vdash \Delta', \Delta}\ \supset L \qquad \frac{\Gamma \vdash A \,;\, \Delta \quad \Gamma', B \vdash \Delta'}{\Gamma', \Gamma \vdash A - B \,;\, \Delta', \Delta}\ -R \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma, A - B \vdash \Delta}\ -L$$

$$\frac{\Gamma \vdash A, \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \forall X.A \,;\, \Delta}\ \forall R \qquad \frac{\Gamma, A\{B/X\} \vdash \Delta}{\Gamma, \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash A\{B/X\} \,;\, \Delta}{\Gamma \vdash \exists X.A \,;\, \Delta}\ \exists R \qquad \frac{\Gamma, A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma, \exists X.A \vdash \Delta}\ \exists L$$

Fig. 14. The sub-logic of the LK sequent calculus corresponding to LKQ.

in Figure 15. Notice how, even without the explicit notion of values and co-values, the stoup still manages to restrict the possible derivations that might be built on top of it. For example, a proposition in the stoup cannot be subject to structural rules. These restrictions imposed by focusing can help guide the bottom-up development of a proof tree, cutting out unneeded flexibility from the inference rules that encourage proof development to "fail early." In the LKQ sub-logic, when the $\vee R_1$ rule is applied to the conclusion $\Gamma \vdash A \vee B \,;\, \Delta$ we get the premise $\Gamma \vdash A \,;\, \Delta$ where A is still in focus, forcing us to keep working with A to see if we made the correct choice (perhaps B was the correct disjunct to prove). Dually in the LKT sub-logic, when

$$A,B,C \in \mathit{Proposition} ::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid \neg A \mid A \supset B \mid A - B \mid \forall X.A \mid \exists X.A$$
$$\Gamma \in \mathit{Hypothesis} ::= A_1, \ldots, A_n$$
$$\Delta \in \mathit{Consequence} ::= A_1, \ldots, A_n$$
$$\mathit{Sequent} ::= \Gamma \vdash \Delta \mid \Gamma ; A \vdash \Delta$$

Core rules:

$$\frac{}{A \vdash A}\ Ax \qquad \frac{}{; A \vdash A}\ Ax_F \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma', A \vdash \Delta'}{\Gamma', \Gamma \vdash \Delta', \Delta}\ Cut \qquad \frac{\Gamma ; A \vdash \Delta}{\Gamma, A \vdash \Delta}\ FL$$

Structural rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ WR \qquad \frac{\Gamma ; C \vdash \Delta}{\Gamma ; C \vdash A, \Delta}\ WR_F \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ WL \qquad \frac{\Gamma ; C \vdash \Delta}{\Gamma, A ; C \vdash \Delta}\ WL_F$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ CR \qquad \frac{\Gamma ; C \vdash A, A, \Delta}{\Gamma ; C \vdash A, \Delta}\ CR_F \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ CL \qquad \frac{\Gamma, A, A ; C \vdash \Delta}{\Gamma, A ; C \vdash \Delta}\ CL_F$$

$$\frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\ XR \qquad \frac{\Gamma ; C \vdash \Delta, A, B, \Delta'}{\Gamma ; C \vdash \Delta, B, A, \Delta'}\ XR_F \qquad \frac{\Gamma', B, A, \Gamma \vdash \Delta}{\Gamma', A, B, \Gamma \vdash \Delta}\ XL \qquad \frac{\Gamma', B, A, \Gamma ; C \vdash \Delta}{\Gamma', A, B, \Gamma ; C \vdash \Delta}\ XL_F$$

Logical rules:

$$\frac{}{\Gamma \vdash \top, \Delta}\ \top R \qquad \text{no } \top L \text{ rule} \qquad \text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma ; \bot \vdash \Delta}\ \bot L$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\ \wedge R \qquad \frac{\Gamma ; A \vdash \Delta}{\Gamma ; A \wedge B \vdash \Delta}\ \wedge L_1 \qquad \frac{\Gamma ; B \vdash \Delta}{\Gamma ; A \wedge B \vdash \Delta}\ \wedge L_2$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vee R_1 \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta}\ \vee R_2 \qquad \frac{\Gamma ; A \vdash \Delta \quad \Gamma ; B \vdash \Delta}{\Gamma ; A \vee B \vdash \Delta}\ \vee L$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \neg R \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma ; \neg A \vdash \Delta}\ \neg L$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\ \supset R \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' ; B \vdash \Delta'}{\Gamma', \Gamma ; A \supset B \vdash \Delta', \Delta}\ \supset L \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma' ; B \vdash \Delta'}{\Gamma', \Gamma \vdash A - B, \Delta', \Delta}\ - R \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma ; A - B \vdash \Delta}\ - L$$

$$\frac{\Gamma \vdash A, \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \forall X.A, \Delta}\ \forall R \qquad \frac{\Gamma ; A\{B/X\} \vdash \Delta}{\Gamma ; \forall X.A \vdash \Delta}\ \forall L$$

$$\frac{\Gamma \vdash A\{B/X\}, \Delta}{\Gamma \vdash \exists X.A, \Delta}\ \exists R \qquad \frac{\Gamma, A \vdash \Delta \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma ; \exists X.A \vdash \Delta}\ \exists L$$

Fig. 15. The focused sub-logic of the LK sequent calculus corresponding to LKT.

the $\wedge L_1$ rule is applied to the conclusion $\Gamma ; A \wedge B \vdash \Delta$ we get the premise $\Gamma ; A \vdash \Delta$ which forces us to keep working with the A in focus in case it was the wrong choice (perhaps B was the correct assumption to use). So in proof search, focusing makes the search algorithm more efficient by cutting down on the search space, whereas in calculi, focusing identifies a well-behaved sub-syntax for the operational semantics.

Dynamic focusing

For the dynamic method of focusing, consider which steps were missing from the operational semantics. So instead of ruling out troublesome corners of the syntax, we

$$F \in \mathit{FocusCxt} ::= (\Box, v) \mid (V, \Box) \mid \mathsf{in}_1(\Box) \mid \mathsf{in}_2(\Box) \mid e \cdot \Box \mid B@\Box$$
$$L \in \mathit{CoFocusCxt} ::= \Box \cdot e$$

Operational rules:

$$(s_v) \qquad \langle F[v] \| E \rangle \mapsto \langle v \| \tilde{\mu} x.\langle F[x] \| E \rangle \rangle \qquad (v \notin \mathit{Value}_v, x \notin FV(F) \cup FV(E))$$
$$(s_v) \qquad \langle V \| L[v] \rangle \mapsto \langle v \| \tilde{\mu} x.\langle V \| L[x] \rangle \rangle \qquad (v \notin \mathit{Value}_v, x \notin FV(L) \cup FV(V))$$

Rewriting rules:

$$(s_v) \qquad F[v] \to \mu\beta.\langle v \| \tilde{\mu} x.\langle F[x] \| \beta \rangle \rangle \qquad (v \notin \mathit{Value}_v, x \notin FV(F), \beta \notin FV(F) \cup FV(v))$$
$$(s_v) \qquad L[v] \to \tilde{\mu} y.\langle v \| \tilde{\mu} x.\langle y \| L[x] \rangle \rangle \qquad (v \notin \mathit{Value}_v, x \notin FV(L),\ y \notin FV(L) \cup FV(v))$$

Fig. 16. The ς semantics for the call-by-value (v) half of the dual calculi.

will add additional steps to kick-start stuck commands. Recall that in our stuck call-by-value command, $\langle((\mu\beta.\langle f \| 1 \cdot \beta\rangle), 2) \| \mathrm{out}_1[\alpha]\rangle$, the $\beta_v^\times$ operational step was stuck because a pair with a non-value component needs to interact with a projection. One thing we can do in this situation is lift the non-value component out of the pair and assign it a name via an input abstraction. Such a step reveals a hidden μ_v reduction and lets the computation continue to bring the application of f to the top:

$$\begin{aligned}\langle((\mu\beta.\langle f \| 1 \cdot \beta\rangle), 2) \| \mathrm{out}_1[\alpha]\rangle &\mapsto_? \langle \mu\beta.\langle f \| 1 \cdot \beta\rangle \| \tilde{\mu} x.\langle (x, 2) \| \mathrm{out}_1[\alpha]\rangle\rangle \\ &\mapsto_{\mu_v} \langle f \| 1 \cdot \tilde{\mu} x.\langle (x, 2) \| \mathrm{out}_1[\alpha]\rangle\rangle\end{aligned}$$

Now, assuming that the call to f returns the result 3, the computation can continue along to present 3 as the result to α, yielding the desired answer:

$$\begin{aligned}\langle f \| 1 \cdot \tilde{\mu} x.\langle (x, 2) \| \mathrm{out}_1[\alpha]\rangle\rangle &\mapsto\!\!\mapsto \langle 3 \| \tilde{\mu} x.\langle (x, 2) \| \mathrm{out}_1[\alpha]\rangle\rangle \\ &\mapsto_{\tilde{\mu}_v} \langle (3, 2) \| \mathrm{out}_1[\alpha]\rangle \\ &\mapsto_{\beta_v^\times} \langle 3 \| \alpha\rangle\end{aligned}$$

That one extra lifting step was all that was needed to continue the computation and get to the final command. Likewise, the stuck call-by-name command $\langle(1, 2) \| \mathrm{out}_1[\tilde{\mu} x.\langle 0 \| \alpha\rangle]\rangle$ has a non-co-value component in the projection, so we can similarly lift the component out of the projection and assign it a name via an output abstraction:

$$\begin{aligned}\langle(1, 2) \| \mathrm{out}_1[\tilde{\mu} x.\langle 0 \| \alpha\rangle]\rangle &\mapsto_? \langle \mu\beta.\langle (1, 2) \| \mathrm{out}_1[\beta]\rangle \| \tilde{\mu} x.\langle 0 \| \alpha\rangle\rangle \\ &\mapsto_{\tilde{\mu}_n} \langle 0 \| \alpha\rangle\end{aligned}$$

Lifting non-(co-)value components out of introduction forms of (co-)terms seems to be the missing step in β-stuck commands.

The full set of such lifting rules are given in Figure 16 for the call-by-value semantics and Figure 17 for the call-by-name semantics. These operational rules give a minimal set of extra steps required to reduce hidden computations nested deeply inside terms and co-terms in a way that matches the call-by-value and call-by-name semantics for the λ-calculus. Additionally, the rewriting rules are generalized to operate on terms and co-terms directly, making it possible to lift the appropriate sub-computations out of (co-)terms in any context, rather than only in commands.

$$F \in \mathit{FocusCxt} ::= \Box \cdot v$$
$$L \in \mathit{CoFocusCxt} ::= \mathsf{out}_1[\Box] \mid \mathsf{out}_2[\Box] \mid [\Box, e] \mid [E, \Box] \mid v \cdot \Box \mid B@\Box$$

Operational rules:

$$(\varsigma_n) \qquad \langle F[e] \| E \rangle \mapsto \langle \mu\alpha.\langle F[\alpha] \| E \rangle \| e \rangle \qquad (e \notin \mathit{CoValue}_n, \alpha \notin FV(F) \cup FV(E))$$
$$(\varsigma_n) \qquad \langle V \| L[e] \rangle \mapsto \langle \mu\alpha.\langle V \| L[\alpha] \rangle \| e \rangle \qquad (e \notin \mathit{CoValue}_n, \alpha \notin FV(L) \cup FV(V))$$

Rewriting rules:

$$(\varsigma_n) \qquad F[e] \to \mu\beta.\langle \mu\alpha.\langle F[\alpha] \| \beta \rangle \| e \rangle \qquad (e \notin \mathit{CoValue}_n, \alpha \notin FV(F), \beta \notin FV(F) \cup FV(v))$$
$$(\varsigma_n) \qquad L[e] \to \tilde{\mu}y.\langle \mu\alpha.\langle y \| L[\alpha] \rangle \| e \rangle \qquad (e \notin \mathit{CoValue}_n, \alpha \notin FV(L),\ y \notin FV(L) \cup FV(v))$$

Fig. 17. The ς semantics for the call-by-name (n) half of the dual calculi.

For example, the call-by-value operational rule ς_v lets us lift out the non-value v in the command $\langle \mathsf{in}_1(v) \| \alpha \rangle$, whereas the generalized rewriting rule lets us lift out v directly in the term $\mathsf{in}_1(v)$ itself by abstracting over the co-variable α:

$$\mathsf{in}_1(v) \to_{\varsigma_v} \mu\alpha.\langle v \| \tilde{\mu}x.\langle \mathsf{in}_1(x) \| \alpha \rangle \rangle$$

This extra generality is necessary when we want to use the rewriting theory to aggressively perform lifting reductions in advance, as we soon will in the following subsection. Furthermore, note that extending the semantics of the dual calculi with the ς rules preserves determinism of the operational semantics and confluence of the rewriting theory, since there are no critical pairs between the ς rules and $\mu\tilde{\mu}\eta_\mu\eta_{\tilde{\mu}}\beta$ rules in either the call-by-value or call-by-name calculus.

For the $\mu_v\tilde{\mu}_v\beta_v\varsigma_v$ call-by-value operational semantics, the net effect is that the final commands are always a *value* yielded to a co-variable or a *simple co-value* (that is, a co-variable or a left introduction co-term) applied to a variable as follows:

$$\mathit{FinalCommand}_v ::= \langle V \| \alpha \rangle \mid \langle x \| E_s \rangle$$
$$V \in \mathit{Value}_v ::= x \mid (V, V') \mid \mathsf{in}_1(V) \mid \mathsf{in}_2(V) \mid \mathsf{not}(e) \mid \lambda x.v \mid e \cdot V \mid \Lambda X.v \mid B@V$$
$$E_s \in \mathit{SimpleCoValue}_v ::= \alpha \mid \mathsf{out}_1[e] \mid \mathsf{out}_2[e] \mid [e, e'] \mid \mathsf{not}[v] \mid V \cdot e \mid \tilde{\lambda}\alpha.e \mid B@e \mid \tilde{\Lambda}X.e$$

Dually for the $\mu_n\tilde{\mu}_n\beta_n\varsigma_n$ call-by-name operational semantics, the final commands are always a *simple value* (a variable or an introduction term) yielded to a co-variable or a *co-value* applied to a variable as follows:

$$\mathit{FinalCommand}_n ::= \langle V_s \| \alpha \rangle \mid \langle x \| E \rangle$$
$$V_s \in \mathit{SimpleValue}_n ::= x \mid (v, v') \mid \mathsf{in}_1(v) \mid \mathsf{in}_2(v) \mid \mathsf{not}(e) \mid \lambda x.v \mid E \cdot v \mid \Lambda X.v \mid B@v$$
$$E \in \mathit{CoValue}_n ::= \alpha \mid \mathsf{out}_1[E] \mid \mathsf{out}_2[E] \mid [E, E'] \mid \mathsf{not}[v] \mid v \cdot E \mid \tilde{\lambda}\alpha.e \mid B@E \mid \tilde{\Lambda}X.e$$

If we only take well-typed commands into consideration, then we get a standard type safety theorem which says that well-typed commands always reduce to a final command, and do not get stuck on any interacting (and potentially mismatched) introduction forms:

Theorem 3 (Type safety)
For any command $c : (\Gamma \vdash \Delta)$*:*

- *if* $c \mapsto\!\!\!\!\!\rightarrow c'$ *by the* $\mu_v\tilde{\mu}_v\beta_v\varsigma_v$ *operational semantics, then* $c' : (\Gamma \vdash \Delta)$ *and* c' *is irreducible (*i.e., $c' \not\mapsto$*) if and only if* c' *is a call-by-value final command, and*
- *if* $c \mapsto\!\!\!\!\!\rightarrow c'$ *by the* $\mu_n\tilde{\mu}_n\beta_n\varsigma_n$ *operational semantics, then* $c' : (\Gamma \vdash \Delta)$ *and* c' *is irreducible (*i.e., $c' \not\mapsto$*) if and only if* c' *is a call-by-name final command.*

This statement of big-step type safety follows from the small-step lemmas of *progress* and *preservation* (Wright and Felleisen, 1994), which can easily be confirmed by induction on typing derivations and inversion on the possible operational steps.

Lemma 1 (Progress and preservation)
For any command $c : (\Gamma \vdash \Delta)$*:*

Progress: either c *is a call-by-value (respectively, call-by-name) final command or there is a command* c' *such that* $c \mapsto c'$ *by the call-by-value* $\mu_v\tilde{\mu}_v\beta_v\varsigma_v$ *(respectively, call-by-name* $\mu_n\tilde{\mu}_n\beta_n\varsigma_n$*) operational semantics, and*

Preservation: if $c \mapsto c'$ *by either the call-by-value* $\mu_v\tilde{\mu}_v\beta_v\varsigma_v$ *or call-by-name* $\mu_n\mu_n\beta_n\varsigma_n$ *operational semantics, then* $c' : (\Gamma \vdash \Delta)$*.*

Recall that $\bar{\lambda}\mu\tilde{\mu}$-calculus originally used a different β rule for functions, namely

$$(\beta^{\rightarrow}) \qquad \langle \lambda x.v \| v' \cdot e \rangle \mapsto \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle \qquad x \notin FV(e)$$

This $\beta^{\rightarrow}$ works the same for both call-by-name and call-by-value reduction; since the argument v' is bound to x with an input abstraction, the rules of the core $\mu\tilde{\mu}$-calculus take over to determine whether or not the argument is evaluated now (by a μ_v reduction, for example) or later (by a $\tilde{\mu}_n$ reduction). Furthermore, this form of $\beta^{\rightarrow}$ reduction applies more often than the strategy-specific $\beta_v^{\rightarrow}$ and $\beta_n^{\rightarrow}$, so we might ask if it avoids the need of focusing for functions altogether. Unfortunately, the general $\beta^{\rightarrow}$ rule still suffers a similar, if more subtle, fate as the strategy-specific β rules. For example, consider the command $\langle f \| \mu\beta.\langle 1 \| \alpha \rangle \cdot \tilde{\mu}x.\langle 0 \| \alpha \rangle \rangle$ that corresponds to the expression $\mathbf{let}\, x = f\ (\mathsf{abort1})\mathbf{in}\, 0$ in a functional language containing the control operator abort that halts the current computation and yields its argument as the result. In call-by-value this expression should evaluate to 1, and in call-by-name it should evaluate to 0, but the $\beta^{\rightarrow}$ rule does not help us since there is a free variable f instead of a λ-abstraction. In this command, the ς rules are still necessary to get the final result, and unfortunately combining the general $\beta^{\rightarrow}$ rule with $\varsigma^{\rightarrow}$ creates a mild form of non-determinism in the operational semantics since some $\beta^{\rightarrow}$ redexes are also $\varsigma^{\rightarrow}$ redexes (though the associated rewriting theories are still confluent).

As it turns out, though, the combination of lifting and strategy-specific $\beta^{\rightarrow}$ reductions are more powerful than the generalized $\beta^{\rightarrow}$ rule. In call-by-value, the combination of $\varsigma_v^{\rightarrow}$, $\tilde{\mu}_v$, and $\beta_v^{\rightarrow}$ exactly simulate the $\bar{\lambda}\mu\tilde{\mu}$-calculus $\beta^{\rightarrow}$ rule as follows:

$$\begin{aligned}\langle \lambda x.v \| v' \cdot e \rangle &\rightarrow [\varsigma_v]\langle \lambda x.v \| \tilde{\mu}y.\langle v' \| \tilde{\mu}x.\langle y \| x \cdot e \rangle \rangle \rangle \rightarrow [\tilde{\mu}_v]\langle v' \| \tilde{\mu}x.\langle \lambda x.v \| x \cdot e \rangle \rangle \\ &\rightarrow [\beta[\rightarrow][v]]\langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle\end{aligned}$$

In call-by-name, observe that the combination of $\overline{\lambda}\mu\tilde{\mu}$'s $\beta^{\rightarrow}$ and μ_n rules simulate the call-by-name-specific $\beta^{\rightarrow}_n$ even when the call stack is not a co-value,

$$\langle \lambda x.v \| v' \cdot e \rangle \to [\beta^{\rightarrow}] \langle v' \| \tilde{\mu}x.\langle v \| e \rangle \rangle \to [\mu_n] \langle v \{v'/x\} \| e \rangle$$

but together the $\tilde{\mu}_n \eta_\mu \beta^{\rightarrow}_n \varsigma_n$ rules perform the same reduction as follows:

$$\begin{aligned} \langle \lambda x.v \| v' \cdot e \rangle &\to_{\varsigma_n} \langle \lambda x.v \| \tilde{\mu}y.\langle \mu\alpha.\langle y \| v' \cdot \alpha \rangle \| e \rangle \rangle \to_{\tilde{\mu}_n} \langle \mu\alpha.\langle \lambda x.v \| v' \cdot \alpha \rangle \| e \rangle \\ &\to_{\beta^{\rightarrow}_n} \langle \mu\alpha.\langle v \{v'/x\} \| \alpha \rangle \| e \rangle \to_{\eta_\mu} \langle v \{v'/x\} \| e \rangle \end{aligned}$$

So even though type safety (Theorem 3) cannot dispense with the ς rules by adopting the $\overline{\lambda}\mu\tilde{\mu}$-calculus' original $\beta^{\rightarrow}$ rules, we can still rely on the combination of strategy-specific $\beta^{\rightarrow}\varsigma$ rules from Figures 10, 16 and Figures 11, 17 to get all the same results with deterministic operational semantics.

Static versus dynamic focusing

Now that we have two different methods for addressing β-stuck commands, one question still remains: what do the static and dynamic methods have to do with one another? As it turns out, they are compatible and complementary solutions to the same problem—two sides of the same coin—that apply the same essential idea at different times. First, one of the major features of static focusing in proof theories and type systems is that the apparent restriction on inference rules is no real restriction at all: every program (*i.e.,* proof) in the original system has a corresponding program with the same type (*i.e.,* specification) in the focused sub-system. We can make this claim more formally for LKQ and LKT by observing that the syntactic transformations in Figures 18 and 19 translate general dual calculi expressions into the LKQ and LKT sub-syntaxes, respectively, with the same type (which can be confirmed by induction on syntax and typing derivations). These translations are defined in such a way that an expression that happens to already lie in the LKQ sub-syntax is not altered by Q-focusing translation, and likewise LKT expressions are not altered by T-focusing translation.

With the focusing translations and the ς rewriting theory in hand, we can now observe that both the static and dynamic methods of focusing amount to the same thing. In particular, notice that the LKQ sub-syntax is just the ς_v-normal forms from the original dual calculus and the Q-focusing translation performs call-by-value ς_v-normalization, and similarly the T-focusing translation is just call-by-name ς_n-normalization into the LKT sub-syntax of ς_n-normal forms, which can be confirmed by induction on the syntax of (co-)terms and commands.

Theorem 4 (Focusing)

- *In the call-by-value dual calculus, every LKQ command, term, and co-term is a* ς_v*-normal form, and* $c \twoheadrightarrow_{\varsigma_v} [\![c]\!]^Q$, $v \twoheadrightarrow_{\varsigma_v} [\![v]\!]^Q$, *and* $e \twoheadrightarrow_{\varsigma_v} [\![e]\!]^Q$.
- *In the call-by-name dual calculus, every LKT command, term, and co-term is a* ς_n*-normal form, and* $c \twoheadrightarrow_{\varsigma_n} [\![c]\!]^T$, $v \twoheadrightarrow_{\varsigma_n} [\![v]\!]^T$, *and* $e \twoheadrightarrow_{\varsigma_n} [\![e]\!]^T$.

$$[\![\langle v \| e \rangle]\!]^Q \triangleq \langle [\![v]\!]^Q \| [\![e]\!]^Q \rangle$$

$$\left.\begin{array}{rlrl}
[\![x]\!]^Q &\triangleq x & [\![\alpha]\!]^Q &\triangleq \alpha \\
[\![\mu\alpha.c]\!]^Q &\triangleq \mu\alpha.[\![c]\!]^Q & [\![\tilde{\mu}x.c]\!]^Q &\triangleq \tilde{\mu}x.[\![c]\!]^Q \\
[\![()]\!]^Q &\triangleq () & [\![[]]\!]^Q &\triangleq [] \\
[\![(v,v')]\!]^Q &\triangleq \mu\alpha.\langle [\![v]\!]^Q \| \tilde{\mu}x.\langle [\![(x,v')]\!]^Q \| \alpha \rangle \rangle & [\![\mathsf{out}_i[e]]\!]^Q &\triangleq \mathsf{out}_i[[\![e]\!]^Q] \\
[\![(V,v)]\!]^Q &\triangleq \mu\alpha.\langle [\![v]\!]^Q \| \tilde{\mu}x.\langle [\![(V,x)]\!]^Q \| \alpha \rangle \rangle & [\![[e,e']]\!]^Q &\triangleq [[\![e]\!]^Q, [\![e']\!]^Q] \\
[\![(V,V')]\!]^Q &\triangleq ([\![V]\!]^Q, [\![V']\!]^Q) & [\![\mathsf{not}[v']]\!]^Q &\triangleq \mathsf{not}[[\![v']\!]^Q] \\
[\![\mathsf{in}_i(v)]\!]^Q &\triangleq \mu\alpha.\langle [\![v]\!]^Q \| \tilde{\mu}x.\langle [\![\mathsf{in}_i(x)]\!]^Q \| \alpha \rangle \rangle & [\![v \cdot e]\!]^Q &\triangleq \tilde{\mu}x.\langle [\![v]\!]^Q \| \tilde{\mu}y.\langle x \| [\![y \cdot e]\!]^Q \rangle \rangle \\
[\![\mathsf{in}_i(V)]\!]^Q &\triangleq \mathsf{in}_i([\![V]\!]^Q) & [\![V \cdot e]\!]^Q &\triangleq [\![V]\!]^Q \cdot [\![e]\!]^Q \\
[\![\mathsf{not}(e)]\!]^Q &\triangleq \mathsf{not}([\![e]\!]^Q) & \left[\!\left[\tilde{\lambda}\alpha.e\right]\!\right]^Q &\triangleq \tilde{\lambda}\alpha.[\![e]\!]^Q \\
[\![\lambda x.v']\!]^Q &\triangleq \lambda x.[\![v']\!]^Q & [\![B@e]\!]^Q &\triangleq B@[\![e]\!]^Q \\
[\![e \cdot v]\!]^Q &\triangleq \mu\alpha.\langle [\![v]\!]^Q \| \tilde{\mu}x.\langle [\![e]\!]^Q \cdot x \| \alpha \rangle \rangle & [\![\tilde{\Lambda}X.e]\!]^Q &\triangleq \tilde{\Lambda}X.[\![e]\!]^Q \\
[\![e \cdot V]\!]^Q &\triangleq [\![e]\!]^Q \cdot [\![V]\!]^Q & & \\
[\![\Lambda X.v']\!]^Q &\triangleq \Lambda X.[\![v']\!]^Q & & \\
[\![B@v]\!]^Q &\triangleq \mu\alpha.\langle [\![v]\!]^Q \| \tilde{\mu}x.\langle [\![B@x]\!]^Q \| \alpha \rangle \rangle & & \\
[\![B@V]\!]^Q &\triangleq B@[\![V]\!]^Q & &
\end{array}\right\} v \notin \mathit{Value}_v$$

Fig. 18. The Q-focusing translation to the LKQ sub-syntax.

$$[\![\langle v \| e \rangle]\!]^T \triangleq \langle [\![v]\!]^T \| [\![e]\!]^T \rangle$$

$$\left.\begin{array}{rlrl}
[\![x]\!]^T &\triangleq x & [\![\alpha]\!]^T &\triangleq \alpha \\
[\![\mu\alpha.c]\!]^T &\triangleq \mu\alpha.[\![c]\!]^T & [\![\tilde{\mu}x.c]\!]^T &\triangleq \tilde{\mu}x.[\![c]\!]^T \\
[\![()]\!]^T &\triangleq () & [\![[]]\!]^T &\triangleq [] \\
[\![(v,v')]\!]^T &\triangleq ([\![v]\!]^T, [\![v']\!]^T) & [\![\mathsf{out}_i[e]]\!]^T &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| [\![\mathsf{out}_i[\alpha]]\!]^T \rangle \| [\![e]\!]^T \rangle \\
[\![\mathsf{in}_i(v)]\!]^T &\triangleq \mathsf{in}_i([\![v]\!]^T) & [\![\mathsf{out}_i[E]]\!]^T &\triangleq \mathsf{out}_i[[\![E]\!]^T] \\
[\![\mathsf{not}(e)]\!]^T &\triangleq \mathsf{not}([\![e]\!]^T) & [\![[e,e']]\!]^T &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| [\![[\alpha,e']]\!]^T \rangle \| [\![e]\!]^T \rangle \\
[\![\lambda x.v]\!]^T &\triangleq \lambda x.[\![v]\!]^T & [\![[E,e]]\!]^T &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| [\![[E,\alpha]]\!]^T \rangle \| [\![e]\!]^T \rangle \\
[\![e \cdot v]\!]^T &\triangleq \mu\alpha.\langle \mu\beta.\langle \beta \cdot [\![v]\!]^T \| \alpha \rangle \| [\![e]\!]^T \rangle & [\![[E,E']]\!]^T &\triangleq [[\![E]\!]^T, [\![E']\!]^T] \\
[\![E \cdot v]\!]^T &\triangleq [\![E]\!]^T \cdot [\![v]\!]^T & [\![\mathsf{not}[v]]\!]^T &\triangleq \mathsf{not}[[\![v]\!]^T] \\
[\![\Lambda X.v]\!]^T &\triangleq \Lambda X.[\![v]\!]^T & [\![v \cdot e]\!]^T &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| [\![v \cdot \alpha]\!]^T \rangle \| [\![e]\!]^T \rangle \\
[\![B@v]\!]^T &\triangleq B@[\![v]\!]^T & [\![v \cdot E]\!]^T &\triangleq [\![v]\!]^T \cdot [\![E]\!]^T \\
& & \left[\!\left[\tilde{\lambda}\alpha.e'\right]\!\right]^T &\triangleq \tilde{\lambda}\alpha.[\![e']\!]^T \\
& & [\![B@e]\!]^T &\triangleq \tilde{\mu}x.\langle \mu\alpha.\langle x \| [\![B@\alpha]\!]^T \rangle \| [\![e]\!]^T \rangle \\
& & [\![B@E]\!]^T &\triangleq B@[\![E]\!]^T \\
& & [\![\tilde{\Lambda}X.e']\!]^T &\triangleq \tilde{\Lambda}X.[\![e']\!]^T
\end{array}\right\} e \notin \mathit{CoValue}_n$$

Fig. 19. The T-focusing translation to the LKT sub-syntax.

Therefore, the difference between the static and dynamic methods of focusing is not a matter of what but when: do we prefer to leave ς redexes to happen during execution, or would we rather reduce them all up front as a preprocessing pass?

Abstract machines

The operational semantics of the core and dual calculi is relatively straightforward to specify and execute: reduction rules are checked against and applied directly to commands. The situation in the term-based λ-calculus, however, is not so easy; the next step to take may not be found directly at the top of the term itself, but may be buried somewhere deep inside. Therefore, an operational semantics for the λ-calculus must also include a *search* for the next step which is very different from the way that the λ-calculus is implemented on a real machine. To help bridge the gap between the mathematics and the machine, we can instead use an *abstract machine* for evaluating terms. As opposed to an operational semantics that composes together reduction with a recursive search function as separate steps, an abstract machine is an iterative interpreter that weaves both parts of evaluation together. To achieve this iterative structure, an abstract machine for the λ-calculus does not act on terms in isolation, but on a configuration including *both* terms and a representation of their context for evaluation. By having direct access to the evaluation context, it can be built up to search deeper into a term for the next step and then broken down to propagate results back up.

Let us now consider two different abstract machines for the λ-calculus, one implementing call-by-name evaluation and one implementing call-by-value. Although abstract machines usually implement variable binding explicitly with an environment that is part of the machine configuration to be closer to a real implementation, here we will remain more abstract by using substitution-based machines. First, consider the following substitution-based Krivine-style machine (Krivine, 2007) for call-by-name evaluation:

$$\langle v\ v' \| E \rangle \rightsquigarrow \langle v \| E[\Box\ v'] \rangle$$

$$\langle \lambda x.v \| E[\Box\ v'] \rangle \rightsquigarrow \langle v\ \{v'/x\} \| E \rangle$$

The configuration for this machine contains two parts—a λ-calculus term v and an evaluation context E—so that $\langle v \| E \rangle$ can be understood as "the term v found inside the context E." This machine uses two forms of evaluation context—the application of the computation in question to an argument, $E[\Box\ v']$, and the empty context, $\Box$—for finding the next β-redex to perform. The first rule is searching for the next step of the operational semantics; given an application $v\ v'$, the function v must be evaluated first, which is done by looking at v inside the larger context $E[\Box\ v']$. The second rule is performing a function call by β reduction; if an abstraction $\lambda x.v$ is found inside an application to v', then the result $v\ \{v'/x\}$ is returned to the surrounding evaluation context.

Second, consider the substitution-based CEK-style machine (Felleisen and Friedman, 1986) for call-by-value evaluation:

$$\langle v\ v' \| E \rangle \rightsquigarrow \langle v \| E[\Box\ v'] \rangle$$
$$\langle V \| E[\Box\ v] \rangle \rightsquigarrow \langle v \| E[V\ \Box] \rangle$$
$$\langle V \| E[(\lambda x.v)\ \Box] \rangle \rightsquigarrow \langle v\,\{V/x\} \| E \rangle$$

Compared to previous machine, this machine uses one additional form of evaluation context—the application of a function value to the computation in question $E[V\ \Box]$—for finding the next β-redex to perform. The first rule is the same as before. The second rule is new, and reflects the fact that in call-by-value arguments must be evaluated before function calls can be performed; when the function of a call is found to be a value but its argument is not, then our attention must shift to the argument to search for the next step. The third rule is a rephrasing of the β reduction rule from before; if a value V is found inside of an application of the abstraction $\lambda x.v$, then the result $v\,\{V/x\}$ is returned to the surrounding evaluation context.

Since the dual calculi effectively represents evaluation contexts with an explicit syntactic object e, it gives us an abstract language for abstract machines (Ariola *et al.*, 2009). In particular, we may view the syntax of the dual calculi as a higher-level representation of the above substitution-based abstract machines. The λ-calculus term can be represented by a dual calculus term v, the evaluation context can be represented by a co-term e, and the configuration of the machine can be represented by a command c. Interestingly, though, the treatment of focusing in abstract machines tends to be asymmetrical depending on the evaluation strategy: call-by-value abstract machines (like the CEK machine above) tend to rely on dynamic focusing that happens during execution, whereas call-by-name abstract machines (like the Krivine machine above) tend to maintain static focusing.

We can relate the states of the call-by-name Krivine machine to the call-by-name dual calculus by translating the evaluation contexts to co-terms. The empty context can be represented by just an arbitrary co-variable α, and the application to an argument is represented directly as a call stack co-term: $E[\Box\ v'] \triangleq v' \cdot E$. With this interpretation, the first rule of the machine states the relationship between function application in the λ-calculus and call stacks in the dual calculus, and the second rule is exactly the $\beta_n^{\rightarrow}$ operational step:

$$\langle v\ v' \| E \rangle = \langle \mu\alpha.\langle v \| v' \cdot \alpha \rangle \| E \rangle \mapsto [\mu_{[}n]] \langle v \| v' \cdot E \rangle = \langle v \| E[\Box\ v'] \rangle$$
$$\langle \lambda x.v \| E[\Box\ v'] \rangle = \langle \lambda x.v \| v' \cdot E \rangle \mapsto [\beta^{\rightarrow}[n]] \langle v\,\{v'/x\} \| E \rangle$$

Note that if we always start with a co-value in the machine state then the first rule only ever builds co-values in the LKT sub-syntax. For example, by evaluating a term v in the "empty context" as $\langle v \| \alpha \rangle$, the co-term in the machine will always be a chain of call stacks with some number of arguments like $v_1 \cdot v_2 \cdot v_3 \cdot v_4 \cdot \alpha$. Therefore, this Krivine-style machine operates within the statically focused LKT sub-syntax.

Now consider how to apply this relationship to the call-by-value CEK machine and the call-by-value dual calculus. We can extend the previous translation of evaluation

contexts to co-terms so that an applied function value is represented indirectly with an input abstraction: $E[V\ \Box] \triangleq \tilde{\mu}x.\langle V \| x \cdot E\rangle$. With this interpretation, the first rule of the machine relates function application and call stacks as before, the second rule of the machine is exactly the ς_v operational step, and the last rule is a combined $\tilde{\mu}_v\beta_v^{\rightarrow}$ step:

$$\langle v\ v' \| E\rangle = \langle \mu\alpha.\langle v \| v' \cdot \alpha\rangle \| E\rangle \mapsto_{\mu_v} \langle v \| v' \cdot E\rangle = \langle v \| E[\Box\ v']\rangle$$
$$\langle V \| E[\Box\ v]\rangle = \langle V \| v \cdot E\rangle \mapsto_{\varsigma_v} \langle v \| \tilde{\mu}x.\langle V \| x \cdot E\rangle\rangle = \langle v \| E[V\ \Box]\rangle$$
$$\langle V \| E[(\lambda x.v)\ \Box]\rangle = \langle V \| \tilde{\mu}y.\langle \lambda x.v \| y \cdot E\rangle\rangle \mapsto_{\tilde{\mu}_v} \langle \lambda x.v \| V \cdot E\rangle \mapsto [\beta^{\rightarrow}[v]]\langle v\,\{V/x\} \| E\rangle$$

Notice that this machine does not necessarily operate within the focused LKQ sub-syntax: the first rule might push a non-value computation onto a call stack. In this case, the ς_v rule is needed to refocus the machine during execution. Of course, we could avoid the need for ς_v reduction at run-time by changing our interpretation of application to pre-ς_v-normalize the call stack, as in $E[\Box\ v] \triangleq \tilde{\mu}x.\langle v \| \tilde{\mu}y.\langle x \| y \cdot E\rangle\rangle$. However, this is just a matter of taste since the two timings of focusing amount to the same thing (Theorem 4).

5.3 Call-by-value is dual to call-by-name

We now turn to the duality for which the dual calculi are named. We saw how the symmetries of the sequent calculus present a logical duality that captures De Morgan duals in Section 3.3. This duality is carried over by the Curry–Howard isomorphism and presents itself as two dualities in programming languages:

(1) a duality between the *static* semantics (types) of languages, and
(2) a duality between the *dynamic* semantics (reductions) of languages.

These dualities of programming languages were first observed by (Filinski, 1989) from the correspondence with duality in category theory, which was later expanded upon by Selinger (2001; 2003) in the style of natural deduction. Curien and Herbelin (2000) and Wadler (2003; 2005) brought this duality to the language of sequent calculus, and show how it is better reflected in the language as a duality of syntax corresponding to the inherent symmetries in the logic.

The static aspect of duality between types comes directly from the logical duality of the sequent calculus. Since duality spins a sequent around its turnstile, so that assumptions are exchanged with conclusions, we also have a corresponding swap in the programming language. The dual of a term v of type A is a co-term of the dual type and vice versa, so that the term and co-term components of a command are swapped. Likewise, the duality on types lines up directly with the De Morgan duality on logical propositions. For example, since the types for pairs ($\times$) and sums ($+$) correspond to conjunction ($\wedge$) and disjunction ($\vee$), we have the same relationship with the duality operation:

$$(A \times B)^{\perp} \triangleq (A^{\perp}) + (B^{\perp}) \qquad (A + B)^{\perp} \triangleq (A^{\perp}) \times (B^{\perp})$$

Also following the De Morgan duality, negation ($\neg$) is self-dual.

Duality of sequents:

$$(c : (\Gamma \vdash \Delta))^{\perp} \triangleq c^{\perp} : (\Delta^{\perp} \vdash \Gamma^{\perp})$$

$$(\Gamma \vdash v : A \mid \Delta)^{\perp} \triangleq \Delta^{\perp} \mid v^{\perp} : A^{\perp} \vdash \Gamma^{\perp} \qquad (\Gamma \mid e : A \vdash \Delta)^{\perp} \triangleq \Delta^{\perp} \vdash e^{\perp} : A^{\perp} \mid \Gamma^{\perp}$$

$$(x_n : A_n, \ldots, x_1 : A_1)^{\perp} \triangleq x_1^{\perp} : A_1^{\perp}, \ldots, x_n^{\perp} : A_n^{\perp} \qquad (\alpha_1 : A_1, \ldots, \alpha_n : A_n)^{\perp} \triangleq \alpha_n^{\perp} : A_n^{\perp}, \ldots, \alpha_1^{\perp} : A_1^{\perp}$$

Duality of types:

$$(X)^{\perp} \triangleq \overline{X}$$

$$\top^{\perp} \triangleq \bot \qquad \bot^{\perp} \triangleq \top$$

$$(A \times B)^{\perp} \triangleq (A^{\perp}) + (B^{\perp}) \qquad (A + B)^{\perp} \triangleq (A^{\perp}) \times (B^{\perp})$$

$$(A \to B)^{\perp} \triangleq (B^{\perp}) - (A^{\perp}) \qquad (B - A)^{\perp} \triangleq (A^{\perp}) \to (B^{\perp})$$

$$(\forall X.A)^{\perp} \triangleq \exists X.(A^{\perp}) \qquad (\exists X.A)^{\perp} \triangleq \forall X.(A^{\perp})$$

$$(\neg A)^{\perp} \triangleq \neg(A^{\perp})$$

Duality of programs:

$$\langle v \| e \rangle^{\perp} \triangleq \langle e^{\perp} \| v^{\perp} \rangle$$

$$(x)^{\perp} \triangleq \overline{x} \qquad [\alpha]^{\perp} \triangleq \overline{\alpha}$$

$$(\mu\alpha.c)^{\perp} \triangleq \tilde{\mu}\overline{\alpha}.c^{\perp} \qquad [\tilde{\mu}x.c]^{\perp} \triangleq \mu\overline{x}.c^{\perp}$$

$$()^{\perp} \triangleq [] \qquad []^{\perp} \triangleq ()$$

$$(v_1, v_2)^{\perp} \triangleq [v_1^{\perp}, v_2^{\perp}] \qquad [e_1, e_2]^{\perp} \triangleq (e_1^{\perp}, e_2^{\perp})$$

$$\mathsf{in}_i(v)^{\perp} \triangleq \mathsf{out}_i[v^{\perp}] \qquad \mathsf{out}_i[e]^{\perp} \triangleq \mathsf{in}_i(e^{\perp})$$

$$\mathsf{not}(e)^{\perp} \triangleq \mathsf{not}[e^{\perp}] \qquad \mathsf{not}[v]^{\perp} \triangleq \mathsf{not}(v^{\perp})$$

$$(\lambda x.v)^{\perp} \triangleq \tilde{\lambda}\overline{x}.[v^{\perp}] \qquad [\tilde{\lambda}\alpha.e]^{\perp} \triangleq \lambda\overline{\alpha}.(v^{\perp})$$

$$(e \cdot v)^{\perp} \triangleq e^{\perp} \cdot v^{\perp} \qquad [v \cdot e]^{\perp} \triangleq v^{\perp} \cdot e^{\perp}$$

$$(\Lambda X.v)^{\perp} \triangleq \tilde{\Lambda}X.[v^{\perp}] \qquad [\tilde{\Lambda}X.e]^{\perp} \triangleq \Lambda X.(e^{\perp})$$

$$(B@v)^{\perp} \triangleq B@[v^{\perp}] \qquad [B@e]^{\perp} \triangleq B@(e^{\perp})$$

Fig. 20. The duality relation between the dual calculi.

With the dual counterpart to functions in place, the full duality relationship of types and programs of the dual calculi is defined in Figure 20, where we assume an underlying bijection, denoted by $\overline{x}$ and $\overline{\alpha}$, between variables and co-variables.This relationship is not just a syntactic word game, but it gives us a duality between the typing derivations of terms and co-terms (Curien and Herbelin, 2000; Wadler, 2003):

Theorem 5 (Static duality)

- *The command* $c : (\Gamma \vdash \Delta)$ *is well-typed if and only if the command* $c^{\perp} : (\Delta^{\perp} \vdash \Gamma^{\perp})$ *is.*
- *The term* $\Gamma \vdash v : A \mid \Delta$ *is well-typed if and only if the co-term* $\Delta^{\perp} \mid v^{\perp} : A^{\perp} \vdash \Gamma^{\perp}$ *is.*
- *The co-term* $\Gamma \mid e : A \vdash \Delta$ *is well-typed if and only if the term* $\Delta^{\perp} \vdash e^{\perp} : A^{\perp} \mid \Gamma^{\perp}$ *is.*

Furthermore, if a command, term, or co-term lies in the LKQ sub-syntax, its dual lies in the LKT sub-syntax and vice versa.

Also, notice that the duality operation is involutive on the nose: the dual of the dual is exactly the same as the original.

Theorem 6 (Involution)
For all commands c, terms v, and co-terms e of the dual calculi, $c^{\perp\perp} \triangleq c$, $v^{\perp\perp} \triangleq v$, and $e^{\perp\perp} \triangleq e$.

The dynamic aspect of duality takes form as a relationship between the two reduction systems for evaluating programs: call-by-value reduction is dual to call-by-name reduction. That is, if we have a command c that behaves a certain way according to the call-by-value calculus, then the dual command $c^{\perp}$ behaves in a correspondingly dual way according to the call-by-name calculus, and vice versa. The two operational and rewriting semantics mirror each other exactly, rule for rule.

Theorem 7 (Dynamic duality)
$c \mapsto_{\mu_v \tilde{\mu}_v \beta_v \varsigma_v} c'$ if and only if $c^{\perp} \mapsto_{\mu_n \tilde{\mu}_n \beta_n \varsigma_n} c'^{\perp}$, and dually $c \mapsto_{\mu_n \tilde{\mu}_n \beta_n \varsigma_n} c'$ if and only if $c^{\perp} \mapsto_{\mu_v \tilde{\mu}_v \beta_v \varsigma_v} c'^{\perp}$. And analogously for the rewriting rules.

This duality relationship inherent to computational interpretations of the sequent calculus is a useful vehicle for exploring programming language design and implementation. Because duality is so syntactic in this language, once the general pattern is set up no cleverness is needed to exploit it: terms are mirrored by co-terms, and so we can always ask what happens when they switch places. For example, even though we have presented LK and the dual calculi with subtraction from the start, it was actually developed after the fact as a means to complete duality (Curien and Herbelin, 2000). Once a sequent-based language with functions is developed, there is a glaring gap of symmetry begging one to ask "what happens when λ-abstractions and call-stacks switch places?" Similarly, this syntactic form of duality was used to ask (Wadler, 2005), and subsequently answer (Ariola *et al.*, 2011), the question "if call-by-value is dual to call-by-name, then what is dual to call-by-need (Ariola *et al.*, 1995)?" By figuring out what is a call-by-need sequent calculus, the dual to call-by-need comes for free.

6 Conclusion

We have now seen how the sequent calculus gives us a programming language for classical logic by using the Curry–Howard isomorphism to derive another view of computation. This view lets us look at functional programming from a lower level using a language tailored for representing abstract machines. The important role of *contexts* is always in the background of the λ-calculus and functional languages—for example, when studying the semantics of the λ-calculus, contexts explicitly arise in abstract machines, operational semantics, and CPS—and the sequent calculus gives a first-class body to the essence of contexts. The language of the sequent calculus also lets us see the computational meaning of dualities in logic as it is expressed directly in the syntax of programs, showing us

- the duality between call-by-value and call-by-name evaluation,
- the duality between manipulating values and manipulating contexts, and
- the duality between types in programming languages, like products and sums.

And in the context of functional programming, these kinds of dualities can be used to tackle difficult issues like deriving well-founded principles of co-induction from the more intuitive principles of induction (Downen *et al.*, 2015). We also saw how the concept of focusing from proof search can be used to maintain type safety in the sequent calculus, so that we avoid getting prematurely stuck while keeping computation at the top of a command. The two approaches to focusing in the syntax or in the reductions amount to the same end, and just differ in their timing: static focusing (*i.e.,* translating to the LKT and LKQ sub-syntaxes (Curien and Herbelin, 2000)) occurs during "compile-time" and dynamic focusing (*i.e.,* performing ς-reductions (Wadler, 2003)) occurs during "run-time." For an alternative view and introduction to the sequent calculus from the perspective of proof search rather than computation see (Pfenning, 2010b), and for an application of focusing for deciding equivalence of typed λ-calculus terms see (Scherer, 2016).

In our experience, we have found that the sequent calculus provides an enlightening and practical alternative perspective to functional programming that complements the foundations based on the λ-calculus. For example, variations on the λ-calculus are popularly used as an intermediate language in real-world compilers of functional programming languages, so that the compiler can reason about and optimize programs. Given the machine-like nature of the sequent calculus, perhaps which would make for a good intermediate language, too? To answer this question, we designed such an intermediate language and implemented it as a plugin for the GHC (Downen *et al.*, 2016), and obtained a representation that was a compromise combining the advantages of the λ-calculus in both direct style and CPS. Of particular note, we learned how join points—which are a useful feature in both CPS (Kennedy, 2007) and static single assignment (Cytron *et al.*, 1991) intermediate languages—are still important and can be incorporated in a direct style language. From this experiment, we used the connection between natural deduction and the sequent calculus (Gentzen, 1935b) to develop a minimal extension to GHC's existing intermediate language that incorporates the join points from our sequent-based language. As a result, we found that real functional programs benefited from the extension of GHC with join points (see `https://ghc.haskell.org/trac/ghc/wiki/SequentCore` for more details of GHC's use of join points in practice), confirming that the sequent calculus can serve as a catalyst in the practice and implementation of functional languages.

In this paper, we only covered the basics of using the sequent calculus as the core for a programming language. One topic that we did not cover, but is of increasing importance for the foundations of functional programming, is the concept of *polarity*. In terms of computation, polarity takes into consideration not just the operational meaning of each type (*i.e., β*-conversion in the λ-calculus) but also the observational meaning of types (*i.e., η*-conversion in the λ-calculus). Instead of deciding on a single strategy for the language once and for all, the programs of each type are evaluated according to their "optimal" strategy which maximizes their observational

properties. For example, η-conversion for functions types happens on terms in the sequent calculus, so expressions of function type should be evaluated with the call-by-name strategy to let η-conversion be as strong as possible. In contrast, η-conversion for sums types happens on co-terms in the sequent calculus, so expressions of sum types should be evaluated with the call-by-value strategy for the same reason. This difference in η comes from properties of the inference rules for types and lets us divide types into two camps: the *positive* types like sums that warrant a call-by-value interpretation and the *negative* types like functions that warrant a call-by-name interpretation. Lecture materials by Zeilberger (2013), Pfenning (2010a), and Graham-Lengrand (2016) give an introduction to the idea of polarity in logic and languages.

The polarized approach shows how we can design languages that incorporate both eager and lazy evaluation to take advantage of the strengths of both evaluation strategies without bias as to which one must be the "default" throughout programs. But even for functional programming languages which (largely) use a single default strategy, polarity still gives us new insights. For example, polarity gives a logical reconstruction of pattern-matching as found in functional programming languages (Zeilberger, 2009) and shows us how to better reason about the equivalence of functional programs using sum types (Munch-Maccagnoni and Scherer, 2015). Polarity first arose hand-in-hand with focusing in the study of proof search (Andreoli, 1992; Laurent, 2002), and interestingly it too says something important about computation. Whereas focusing tells us how to focus attention on sub-computations, polarity tells us how to adapt the dynamic meaning of types (*i.e.,* how programs are evaluated) to match the static meaning of types (*i.e.,* how programs are type-checked). Since the logic of the sequent calculus is the lingua franca of proof search, the sequent calculus serves as an intermediate common language which lets us discover the surprising connections between proof search and programming languages.

Acknowledgments

We would like to thank Luke Maurer, Philip Johnson-Freyd, Matthias Felleisen, and the anonymous reviewers for their thorough and helpful feedback for improving this paper. This work has been supported by the National Science Foundation under Grant no. CCF-1423617 and Grant no. CCF-1719158.

References

Andreoli, J.-M. (1992) Logic programming with focusing proofs in linear logic. *J. Log. Comput.* **2**(3), 297–347. doi: 10.1093/logcom/2.3.297.

Appel, A. W. (1992) *Compiling with Continuations.* New York, NY, USA, 1992: Cambridge University Press. ISBN 0-521-41695-7.

Ariola, Z. M. & Herbelin, H. (2003) Minimal classical logic and control operators. In Proceedings of 30th International Colloquium in Automata, Languages and Programming:(ICALP 2003). Berlin, Heidelberg:Springer, p. 871–885. ISBN 978-3-540-45061-0. doi:10.1007/3-540-45061-0 68.

Ariola, Z. M., Bohannon, A. & Sabry, A. (2009) Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.* **31**(4), 13:1–13:48. ISSN 0164-0925. doi: 10.1145/1516507.1516508.

Ariola, Z. M., Herbelin, H. & Saurin, A. (2011) Classical call-by-need and duality. In Proceedings of 10th International Conference in Typed Lambda Calculi and Applications (TLCA '11). Berlin, Heidelberg: Springer, pp. 27–44. ISBN 978-3-642-21690-9. doi: 10.1007/978-3-642-21691-6_6.

Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M., & Wadler, P. (1995) A call-by-need lambda calculus. In Proceedings of 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95). New York, NY, USA: ACM, pp. 233–246. ISBN 0-89791-692-1. doi: 10.1145/199448.199507.

Church, A. (1932) A set of postulates for the foundation of logic. *Ann. Math.* **33**(2), 346–366. doi: 10.2307/1968337.

Curien, P.-L. & Herbelin, H. (2000) The duality of computation. In Proceedings of 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00). New York, NY, USA: ACM, pp. 233–243. ISBN 1-58113-202-6. doi: 10.1145/351240.351262.

Curien, P.-L. & Munch-Maccagnoni, G. (2010) The duality of computation under focus. In Proceedings of 6th IFIP TC 1/WG 2.2 International Conference in Theoretical Computer Science (TCS '10). Held as Part of WCC 2010, TCS, Berlin Heidelberg: Springer, pp. 165–181. ISBN 978-3-642-15240-5. doi: 10.1007/978-3-642-15240-5_13.

Curry, H. B., Feys, R. & Craig, W. (1958) *Combinatory Logic*, vol. 1. North-Holland Publishing Company.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991) Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* **13**(4), 451–490. ISSN 0164-0925. doi: 10.1145/115372.115320.

de Bruijn, N. (1968) *AUTOMATH, A Language for Mathematics*. Technical Report 66-WSK-05, Technological University Eindhoven.

Downen, P. & Ariola, Z. M. (2014) The duality of construction. In Proceedings of 23rd European Symposium on Programming in Programming Languages and Systems (ESOP '14). Held as Part of the European Joint Conferences on Theory and Practice of Software, Lecture Notes in Computer Science, vol. 8410, Berlin Heidelberg: Springer, pp. 249–269. ISBN 978-3-642-54832-1. doi: 10.1007/978-3-642-54833-8_14.

Downen, P., Johnson-Freyd, P. & Ariola, Z. M. (2015) Structures for structural recursion. In Proceedings of 20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15). New York, NY, USA: ACM, pp. 127–139. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784762.

Downen, P., Maurer, L., Ariola, Z. M. & Peyton Jones, S. (2016) Sequent calculus as a compiler intermediate language. In Proceedings of 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16). New York, NY, USA: ACM, pp. 74–88. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951931.

Felleisen, M. & Friedman, D. P. (1986) Control operators, the SECD machine, and the λ-calculus. In Proceedings of the IFIP TC 2/WG2.2 Working Conference on Formal Descriptions of Programming Concepts Part III, pp. 193–219.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.* **103**(2), 235–271. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90014-7.

Filinski, A. (1989) *Declarative Continuations and Categorical Duality*. Master's thesis, Computer Science Department, University of Copenhagen, 1989.

Gentzen, G. (1935a) Untersuchungen über das logische schließen. I. *Math. Z.* **39**(1), 176–210. ISSN 0025-5874. doi: 10.1007/BF01201353.

Gentzen, G. (1935b) Untersuchungen über das logische schließen. II. *Math. Z.* **39**(1), 405–431. ISSN 0025-5874. doi: 10.1007/BF01201363.

Girard, J.-Y. (1987) Linear logic. *Theor. Comput. Sci.* **50**(1):1–101. ISSN 0304-3975. doi:10.1016/0304-3975(87)90045-4.

Girard, J.-Y. (1991) A new constructive logic: Classical logic. *Math. Struct. Comput. Sci.* **1**(3), 255–296. doi: 10.1017/S0960129500001328.

Girard, J.-Y. (1993) On the unity of logic. *Ann. Pure Appl. Log.* **59**(3), 201–217. ISSN 0168-0072. doi: 10.1016/0168-0072(93)90093-S.

Girard, J.-Y. (2001) Locus solum: From the rules of logic to the logic of rules. *Math. Struct. Comput. Sci.* **11**(3), 301–506. ISSN 0960-1295. doi: 10.1017/S096012950100336X.

Girard, J.-Y., Taylor, P. & Lafont, Y. (1989) *Proofs and Types* . New York, USA: Cambridge University Press. ISBN 0-521-37181-3.

Graham-Lengrand, S. (2016) The Curry-Howard view of classical logic: A short introduction. Lecture Notes for the MPRI course on Curry-Howard correspondence for Classical Logic. URL `http://www.lix.polytechnique.fr/~lengrand/Work/Teaching/MPRI/Notes.pdf`. Unpublished Manuscript.

Griffin, T. G. (1990) A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, New York, NY, USA: ACM, pp. 47–58. ISBN 0-89791-343-4. doi: 10.1145/96709.96714.

Herbelin, H. (1995) *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes.* PhD thesis, Université Paris 7, January 1995.

Herbelin, H. (2005) *C'est maintenant qu'on calcule : Au coeur de la dualité*. Habilitation thesis, Université Paris 11, 2005.

Howard, W. A. (1980) The formulae-as-types notion of constructions. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, pp. 479–490. ISBN 0123490502. Unpublished manuscript of 1969.

Kelsey, R., *et al.* (1998) Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symb. Comput.* **11**(1), 7–105. ISSN 1573-0557. doi: 10.1023/A:1010051815785.

Kennedy, A. (2007) Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, New York, NY, USA: ACM, pp. 177–190. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291179.

Krivine, J.-L. (2007) A call-by-name lambda-calculus machine. *Higher-Order Symb. Comput.* **20**(3), 199–207. ISSN 1388-3690. doi: 10.1007/s10990-007-9018-9.

Laurent, O. (2002) *Étude de la polarisation en logique.* PhD thesis, Université de la Méditerranée - Aix-Marseille II.

Munch-Maccagnoni, G. (2009) Focalisation and classical realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL*, CSL 2009, Berlin Heidelberg: Springer, pp. 409–423. ISBN 978-3-642-04027-6. doi: 10.1007/978-3-642-04027-6_30.

Munch-Maccagnoni, G. & Scherer, G. (2015) Polarised intermediate representation of lambda calculus with sums. In *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS 2015, pp. 127–140. doi: 10.1109/LICS.2015.22.

Ohori, A. (1999) The logical abstract machine: A Curry-Howard isomorphism for machine code. In *Functional and Logic Programming: 4th Fuji International Symposium*, FLOPS '99, Berlin, Heidelberg: Springer, pp. 300–318. ISBN 978-3-540-47950-5. doi: 10.1007/10705424_20.

Ohori, A. (2003) Register allocation by proof transformation. In *Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003 Held as Part of the*

Joint European Conferences on Theory and Practice of Software, ETAPS 2003, ESOP 2003, Berlin Heidelberg: Springer, pp. 399–413. ISBN 978-3-540-36575-4. doi: 10.1007/3-540-36575-3_27.

Parigot, M. (1992) l m-calculus: An algorithmic interpretation of classical natural deduction. In Logic Programming and Automated Reasoning: International Conference, LPAR '92, pages 190–201, Berlin, Heidelberg, July 1992. Springer Berlin Heidelberg. ISBN 978- 3-540-47279-7. doi:10.1007/BFb0013061.

Peyton Jones, S., Tolmach, A. & Hoare, T. (2001) Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*. ACM SIGPLAN.

Pfenning, F. (2010a) Lecture notes on focusing. Lecture notes for the Oregon Programming Languages Summer School 2010 course on Proof Theory Foundations, Lecture 4. URL `http://www.cs.cmu.edu/~fp/courses/oregon-m10/04-focusing.pdf`. Unpublished Manuscript.

Pfenning, F. (2010b) Lecture notes on sequent calculus. Lecture Notes for the Carnegie Mellon University course 15-816 on Modal Logic, Lecture 8. URL `http://www.cs.cmu.edu/~fp/courses/15816-s10/lectures/08-seqcalc.pdf`. Unpublished Manuscript.

Emmanuel, P. (2004) *Explicit Substitutions, Logic and Normalization*. PhD thesis, Université Paris-Diderot – Paris VII, Jun. 2004.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In Proceedings of the IFIP 9th World Computer Congress, Information Processing 83. Amsterdam: Elsevier Science Publishers B. V. (North-Holland), pp. 513–523.

Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbol. Comput.* **6**(3–4), 233–248. ISSN 0892-4635. doi: 10.1007/BF01019459. URL `http://dx.doi.org/10.1007/BF01019459`.

Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. *Higher-Order Symbol. Comput.* **11**(4), 363–397. ISSN 1388-3690. doi: 10.1023/A:1010027404223.

Scherer, G. (2016) *Which Types Have a Unique Inhabitant? Focusing on Pure Program Equivalence*. PhD thesis, Université Paris-Diderot.

Selinger, P. (2001) categories, Control and duality: On the categorical semantics of the lambda-mu calculus. *Math. Struct. Comput. Sci.* **11**(2), 207–260. ISSN 0960-1295. doi: 10.1017/S096012950000311X.

Selinger, P. (2003) Some remarks on control categories, 2003. URL `http://mathstat.dal.ca/~selinger/papers/controlremarks.pdf`. Unpublished Manuscript.

Singh, S., Peyton Jones, S., Norell, U., Pottier, F., Meijer, E., & McBride, C. (2011) Sexy types—are we done yet? Software Summit. URL `https://www.microsoft.com/en-us/research/video/sexy-types-are-we-done-yet/`.

Wadler, P. (2003) Call-by-value is dual to call-by-name. In Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, pp. 189–201. ISBN 1-58113-756-7. doi: 10.1145/944705.944723.

Wadler, P. (2005) Call-by-value is dual to call-by-name, reloaded. In Proceedings of 16th International Conference in Term Rewriting and Applications (RTA '05)., Berlin Heidelberg: Springer, pp. 185–203. ISBN 978-3-540-32033-3. doi: 10.1007/978-3-540-32033-3_15. URL `dx.doi.org/10.1007/978-3-540-32033-3_15`.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inform. Comput.* **115**(1), 38–94. ISSN 0890-5401. doi: 10.1006/inco.1994.1093.

Zeilberger, N. (2009) *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University.

Zeilberger, N. (2013) Polarity in proof theory and programming. Lecture Notes for the Summer School on Linear Logic and Geometry of Interaction in Torino, Italy. URL `http://noamz.org/talks/logpolpro.pdf`. Unpublished Manuscript.

Appendix A. Classical versus intuitionistic logic and computation

The fact that the LK sequent calculus can prove the law of excluded middle ($A \vee (\neg A)$), assured by duality of the law of contradiction ($\neg(A \wedge (\neg A))$) in Section 3.3, means that it is a proof system for *classical logic*. In contrast, *intuitionistic logic* is missing duality since it accepts non-contradiction in general, but rejects the universal truth of laws like excluded middle or double negation elimination ($(\neg(\neg A)) \supset A$), only allowing for specialized proofs depending on the particular A in question. Intuitionistic logic also only validates three of the four De Morgan laws for commuting negation with conjunction and disjunction, rejecting $\neg(A \wedge B) \supset (\neg A) \vee (\neg B)$ in particular, showing another break of duality.

(Gentzen, 1935a) introduced another sequent calculus called LJ for formalizing intuitionistic logic instead of classical logic. Notice that the LK proof of excluded middle made critical use of multiple consequences and contraction on the right of the sequent in order to apply both $\vee R_1$ and $\vee R_1$ to the same original consequence. Without the ability to manipulate sequents with multiple consequences, the general proof that $A \vee (\neg A)$ is true for any A would not be possible. Such a restriction would break the symmetry of LK—as multiple hypotheses cannot be mirrored by multiple consequences—and destroy the duality that let us convert the general proof of non-contradiction into a proof of the excluded middle. LJ is thus defined as the restriction of LK where sequents contain *exactly one* consequence at all times. Note that with this restriction, LJ does not allow for the right structural rules WR, CR, and XR since they necessarily involve sequents with more than one consequence. For the same reason, LJ does not include the logical connectives for negation ($\neg$) and subtraction ($-$), since the introduction rules for these connectives do not fit within the single-consequence discipline. In their place, they can be encoded in terms of the other connectives in LJ as $\neg A = A \to \bot$ and $A - B = A \wedge (\neg B)$.

The LJ sequent calculus has a close relationship with (Gentzen, 1935a) system NJ of natural deduction, which is naturally a proof system for intuitionistic logic already. More specifically, NJ proofs can be converted to equivalent LJ proofs, and vice versa. The NJ system of natural deduction is shown in Figure A1, which corresponds to the polymorphic λ-calculus with products, sums, and existential types shown in Figure A2. We call a leaf of an NJ proof tree that is not closed off by an axiom (an inference rule with no premise) a *free assumption* of that proof tree and call an NJ proof tree without any free assumptions a *closed proof*. Similarly, a variable found in a λ-calculus term that is not under a matching binder of that variable (introduced by a λ or *case* term) is called a *free variable*, and a term without any free variables is called a *closed term*. With this terminology in mind, the correspondence between the two is that there is an NJ proof derivation of B with free assumptions $A_1, \ldots, A_n$ if and only if there is a typing derivation for some term $M : B$ with free variables $x_1 : A_1, \ldots, x_n : A_n$. By taking advantage of the correspondence

$$A,B,C \in \mathit{Proposition} ::= X \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid A \supset B \mid \forall X.A \mid \exists X.A$$

$$\frac{}{\top}\ \top I \qquad \text{no } \top E \text{ rule} \qquad \text{no } \bot I \text{ rule} \qquad \frac{\bot}{C}\ \bot E$$

$$\frac{A \quad B}{A \wedge B}\ \wedge I \qquad \frac{A \wedge B}{A}\ \wedge E_1 \qquad \frac{A \wedge B}{B}\ \wedge E_2$$

$$\frac{A}{A \vee B}\ \vee I_1 \qquad \frac{B}{A \vee B}\ \vee I_2 \qquad \frac{A \vee B \quad \begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ C\end{array} \quad \begin{array}{c}\overline{B}^{\,y}\\ \vdots\\ C\end{array}}{C}\ \vee I_{x,y}$$

$$\frac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ B\end{array}}{A \supset B}\ \supset I_x \qquad \frac{A \supset B \quad A}{B}\ \supset E$$

$$\frac{\begin{array}{c}\vdots\ (X \notin FV(*))\\ A\end{array}}{\forall X.A}\ \forall I_X \qquad \frac{\forall X.A}{A\{B/X\}}\ \forall E$$

$$\frac{A\{B/X\}}{\exists X.A}\ \exists I \qquad \frac{\exists X.A \quad \begin{array}{c}\overline{A}^{\,x}\\ \vdots\ (X \notin FV(*))\\ C\end{array}}{C}\ (X \notin FV(C))\ \exists E_{X,x}$$

Fig. A1. Gentzen's NJ system of natural deduction.

between NJ and the polymorphic λ-calculus and between the LJ restriction of LK and the analogously restricted dual calculi with a single output, we can demonstrate the correspondence between NJ and LJ as a translation on programs. Consider the mutual translations between the polymorphic λ-calculus and single-output dual calculi shown in Figure A3. The fact that these two translations preserve types means that the logics of LJ and NJ prove the same propositions true.

Theorem 8 (Well-typed translation)

(a) If $M : A$ is a well-typed λ-calculus term with free variables of type Γ, then $\Gamma \vdash [\![M]\!]^{LJ} : A \mid$ is a well-typed pure dual calculi term.

(b) If $\Gamma \vdash v : A \mid$ is a well-typed pure dual calculi term then $[\![v]\!]^{NJ} : A$ is a well-typed λ-calculus term with free variables of type Γ.

(c) If $c : (\Gamma \vdash \alpha : A)$ is a well-typed pure dual calculi command then $[\![c]\!]^{NJ} : A$ is a well-typed λ-calculus term with free variables of type Γ.

(d) If $\Gamma \mid e : B \vdash \alpha : A$ is a well-typed pure dual calculi co-term, then for all well-typed λ-calculus terms $M : B$ with free variables of type Γ, it follows that $[\![e]\!]^{NJ}[M] : A$ is a well-typed λ-calculus term with free variables of type Γ.

Theorem 9 (LJ–NJ provability)

A proof of $A_1, \ldots, A_n \vdash B$ is derivable in LJ if and only if a proof of B with free assumptions $A_1, \ldots, A_n$ is derivable in NJ.

$$
\begin{aligned}
A,B,C \in \mathit{Type} &::= X \mid 1 \mid 0 \mid A \times B \mid A + B \mid A \to B \mid \forall X.A \mid \exists X.A \\
M,N,P \in \mathit{Term} &::= x \\
&\mid () \mid (\mathbf{case}\, M\, \mathbf{of}\,) \\
&\mid (M,N) \mid \mathsf{out}_1(M) \mid \mathsf{out}_2(M) \\
&\mid \mathsf{in}_1(M) \mid \mathsf{in}_2(M) \mid (\mathbf{case}\, M\, \mathbf{of}\, \mathsf{in}_1(x) \Rightarrow N \mid \mathsf{in}_2(y) \Rightarrow P) \\
&\mid \lambda\, x.M \mid M\, N \\
&\mid \Lambda X.M \mid M\, B \\
&\mid B@M \mid (\mathbf{case}\, M\, \mathbf{of}\, X@x \Rightarrow N)
\end{aligned}
$$

Typing rules:

$$\frac{}{() : 1}\ 1I \qquad \text{no } 1E \text{ rule} \qquad \text{no } \bot I \text{ rule} \qquad \frac{v : 0}{\mathbf{case}\, M\, \mathbf{of}\, : C}\ 0E$$

$$\frac{M : A \quad N : B}{(M,N) : A \times B}\ \times I \qquad \frac{M : A \times B}{\mathsf{out}_1(M) : A}\ \times E_1 \qquad \frac{M : A \times B}{\mathsf{out}_2(M) : B}\ \times E_2$$

$$\frac{M : A}{\mathsf{in}_1(M) : A + B}\ +I_1 \qquad \frac{M : B}{\mathsf{in}_2(M) : A + B}\ +I_2 \qquad \frac{M : A + B \quad \begin{array}{c}\overline{x : A}\ x \\ \vdots \\ N : C\end{array} \quad \begin{array}{c}\overline{y : B}\ y \\ \vdots \\ P : C\end{array}}{(\mathbf{case}\, M\, \mathbf{of}\, \mathsf{in}_1(x) \Rightarrow N \mid \mathsf{in}_2(y) \Rightarrow P) : C}\ +I_{x,y}$$

$$\frac{\begin{array}{c}\overline{x : A}\ x \\ \vdots \\ M : B\end{array}}{\lambda\, x.M : A \to B}\ \to I_x \qquad \frac{M : A \to B \quad N : A}{M\, N : B}\ \to E$$

$$\frac{\begin{array}{c}\vdots\ (X \notin FV(*)) \\ M : A\end{array}}{\Lambda X.M : \forall X.A}\ \forall I_X \qquad \frac{M : \forall X.A}{M\, B : A\{B/X\}}\ \forall E$$

$$\frac{M : A\{B/X\}}{B@M : \exists X.A}\ \exists I \qquad \frac{M : \exists X.A \quad \begin{array}{c}\overline{x : A}\ x \\ \vdots\ (X \notin FV(*)) \\ N : C\end{array} \quad (X \notin FV(C))}{(\mathbf{case}\, M\, \mathbf{of}\, X@x \Rightarrow N) : C}\ \exists E_{X,x}$$

Call-by-name rewriting rules:

$$
\begin{aligned}
(\beta^{\times}) \quad & \mathsf{out}_i(M_1, M_2) \to M_i \\
& \mathbf{case}\ \mathsf{in}_i(M)\ \mathbf{of} \\
(\beta^{+}) \quad & \quad \mathsf{in}_1(x_1) \Rightarrow N_1 \to N_i\{M/x_i\} \\
& \quad \mathsf{in}_2(x_2) \Rightarrow N_2 \\
(\beta^{\to}) \quad & (\lambda\, x.M)\, N \to M\{N/x\} \\
(\beta^{\forall}) \quad & (\Lambda X.M)\, B \to M\{B/X\} \\
(\beta^{\exists}) \quad & \mathbf{case}\, B@M\, \mathbf{of}\ X@y \Rightarrow N \to N\{B/X\}\{M/y\}
\end{aligned}
$$

Fig. A2. The polymorphic λ-calculus.

Translation from NJ to LJ:

$$[\![x]\!]^{LJ} \triangleq x$$

$$[\![()]\!]^{LJ} \triangleq ()$$

$$[\![(M,N)]\!]^{LJ} \triangleq ([\![M]\!]^{LJ}, [\![N]\!]^{LJ})$$

$$[\![\mathsf{in}_i(M)]\!]^{LJ} \triangleq \mathsf{in}_i([\![M]\!]^{LJ})$$

$$[\![\lambda x.M]\!]^{LJ} \triangleq \lambda x.[\![M]\!]^{LJ}$$

$$[\![\Lambda X.M]\!]^{LJ} \triangleq \Lambda X.[\![M]\!]^{LJ}$$

$$[\![B@M]\!]^{LJ} \triangleq B@[\![M]\!]^{LJ}$$

$$[\![\mathbf{case}\, M\, \mathbf{of}]\!]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| [] \rangle$$

$$[\![\mathsf{out}_i(M)]\!]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| \mathsf{out}_i[\alpha] \rangle$$

$$\left[\!\!\left[\begin{array}{l} \mathbf{case}\, M\, \mathbf{of} \\ \quad \mathsf{in}_1(x) \Rightarrow N \\ \quad \mathsf{in}_2(y) \Rightarrow P \end{array} \right]\!\!\right]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| [\tilde{\mu}x.\langle [\![N]\!]^{LJ} \| \alpha \rangle, \tilde{\mu}y.\langle [\![P]\!]^{LJ} \| \alpha \rangle] \rangle$$

$$[\![M\, N]\!]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| [\![N]\!]^{LJ} \cdot \alpha \rangle$$

$$[\![M\, B]\!]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| B@\alpha \rangle$$

$$\left[\!\!\left[\begin{array}{l} \mathbf{case}\, M\, \mathbf{of} \\ \quad X@x \Rightarrow N \end{array} \right]\!\!\right]^{LJ} \triangleq \mu\alpha.\langle [\![M]\!]^{LJ} \| \tilde{\Lambda}X.\tilde{\mu}x.\langle N \| \alpha \rangle \rangle$$

Translation from LJ to NJ:

$$[\![\langle v \| e \rangle]\!]^{NJ} \triangleq [\![e]\!]^{NJ}[[\![v]\!]^{NJ}]$$

$$[\![x]\!]^{NJ} \triangleq x$$

$$[\![\mu\alpha.c]\!]^{NJ} \triangleq [\![c]\!]^{NJ}$$

$$[\![()]\!]^{NJ} \triangleq ()$$

$$[\![(v_1,v_2)]\!]^{NJ} \triangleq ([\![v_1]\!]^{NJ}, [\![v_2]\!]^{NJ})$$

$$[\![\mathsf{in}_i(v)]\!]^{NJ} \triangleq \mathsf{in}_i([\![v]\!]^{NJ})$$

$$[\![\lambda x.v]\!]^{NJ} \triangleq \lambda x.[\![v]\!]^{NJ}$$

$$[\![\Lambda X.v]\!]^{NJ} \triangleq \Lambda X.[\![v]\!]^{NJ}$$

$$[\![B@v]\!]^{NJ} \triangleq B@[\![v]\!]^{NJ}$$

$$[\![\alpha]\!]^{NJ} \triangleq \Box$$

$$[\![\tilde{\mu}x.c]\!]^{NJ} \triangleq \mathbf{let}\, x = \Box\, \mathbf{in}\, [\![c]\!]^{NJ}$$

$$[\![[]]\!]^{NJ} \triangleq \mathbf{case}\, \Box\, \mathbf{of}$$

$$[\![\mathsf{out}_i[e]]\!]^{NJ} \triangleq [\![e]\!]^{NJ}[\mathsf{out}_i(\Box)]$$

$$[\![[e_1,e_2]]\!]^{NJ} \triangleq \begin{array}{l} \mathbf{case}\, \Box\, \mathbf{of} \\ \quad \mathsf{in}_1(x) \Rightarrow [\![K_1]\!]^{NJ}[x] \\ \quad \mathsf{in}_2(y) \Rightarrow [\![K_2]\!]^{NJ}[y] \end{array}$$

$$[\![v \cdot e]\!]^{NJ} \triangleq [\![e]\!]^{NJ}[\Box\ [\![v]\!]^{NJ}]$$

$$[\![B@e]\!]^{NJ} \triangleq [\![e]\!]^{NJ}[\Box\ B]$$

$$[\![\tilde{\Lambda}X.e]\!]^{NJ} \triangleq \begin{array}{l} \mathbf{case}\, \Box\, \mathbf{of} \\ \quad X@x \Rightarrow [\![e]\!]^{NJ}[x] \end{array}$$

Fig. A3. Translations between the polymorphic λ-calculus and the pure dual calculi.

Furthermore, because of the consistency of LJ (coming from the consistency of LK by cut elimination in Theorem 1), the correspondence between LJ and NJ means NJ is also consistent. Because NJ does not use sequents, we cannot state its consistency in terms of the contradictory sequent $\bullet \vdash \bullet$. Instead, we can say that NJ is consistency because the provability of propositions is not a trivial predicate: there exist some propositions with proofs and some propositions without proofs. For example, $\top$ is axiomatically true in NJ, whereas $\bot$ does not proof because that would mean that $\bullet \vdash \bot$ can be proved in LJ which would lead to an impossible contradiction caused by $\bot L$ and *Cut*.

Corollary 2 (Consistency)

There are propositions A and B such that a closed proof of A is derivable in NJ and a closed proof of B is not derivable in NJ.

This gives us a close relationship between the two alternative formalizations of intuitionistic logic: NJ and LJ. If we want to find a system of natural deduction that corresponds with the full classical LK sequent calculus, we would have to extend the NJ basis to include proofs of classical reasoning principles. If we are only interested in provability, a direct way to extend the intuitionistic natural deduction NJ to classical logic is to add a sufficiently expressive classical reasoning principle as an axiom to the system. For example, we could add the law of excluded middle to NJ to get NK as Gentzen (1935a) did.

However, there is a more programmatic way of looking at the difference between intuitionistic and classical logic. It turns out that μ-abstractions let programs manipulate their own control flow similar to Scheme's (Kelsey *et al.*, 1998) callcc control operator, or Felleisen's (1992) $\mathscr{C}$ operator. Intuitively, a use of callcc or an abort can be read in terms of an output abstraction that duplicates or deletes its bound co-variable, respectively, to perform contraction or weakening on the active type of the term as seen in Section 4:

$$\mathsf{callcc}(\lambda\alpha.v) \triangleq \mu\alpha.\langle v \| \alpha \rangle \qquad \mathsf{abort}c \triangleq \mu\delta.c \qquad (\delta \notin FV(c))$$

This phenomenon is a consequence of Griffin (1990) observation that under the Curry–Howard correspondence, classical logic corresponds to control flow manipulation, along with the fact that the LK sequent calculus formalizes classical logic. Under this interpretation, multiple consequences in the sequent calculus correspond to multiple available co-variables that give the program multiple possible exit paths. The weakening and contraction rules on the right for these multiple consequences correspond to deleting or copying an exit path, respectively. Indeed, multiple consequences with right-handed structural rules may be seen as the logical essence for this "classical" form of control effects (so called for the connection to classical logic as well as callcc being the traditional control operator), since extending natural deduction with multiple consequences, as in (Parigot, 1992) $\lambda\mu$-calculus. This gives rise to a programming language with control effects equivalent to the λ-calculus with a primitive callcc operator given the type for Pierce's law $\forall X.\forall Y.((X \to Y) \to X) \to X$ (Ariola and Herbelin, 2003) or with a primitive $\mathscr{C}$ operator given the type for double negation elimination $\forall X.((X \to \bot) \to \bot) \to X$, which uses the empty type $\bot$.

Appendix B. An implicit treatment of structure

The traditional LK sequent calculus from Figure 4 represents the structural properties of sequents—exchange, weakening, and contraction—explicitly in the form of inference rules. However, there are alternate sequent calculi and variations on LK that forgo these structural rules by baking the properties deeper into the logic itself, which is especially common when formalizing the type systems for core programming languages based on the sequent calculus (Curien and Herbelin, 2000; Wadler, 2005; Curien and Munch-Maccagnoni, 2010; Munch-Maccagnoni and Scherer, 2015). The first change along this line is to treat the hypotheses and consequences of sequents as *unordered* collections of propositions, for example building sequents out of sets or multisets. This way, the exchange rules XL and XR do not do anything at all, since

the sequents in the premise and conclusion are considered identical. The second change is to rephrase the core axiom and cut rules in a way that bakes in weakening and contraction as follows:

$$\frac{}{\Gamma, A \vdash A, \delta}\ Ax \qquad\qquad \frac{\Gamma \vdash A, \delta \quad \Gamma, A \vdash \delta}{\Gamma \vdash \delta}\ Cut$$

Contraction is completely implicit when hypothesis and consequences are represented by sets: Γ, A, A and Γ, A are already the same set. And in any case, even if multisets are used, contraction can still be derived from these above new Ax and Cut rules. CL is derived as

$$\frac{\Gamma, A, A \vdash \delta \quad \dfrac{}{\Gamma, A \vdash A, \delta}\ Ax}{\Gamma, A \vdash \delta}\ Cut$$

and the derivation of CR is similar. Weakening, unfortunately, cannot be directly derived in the same manner as contraction, but instead it is *admissible*. That is to say, given any proof of the sequent $\Gamma \vdash \delta$, we can build similar proofs $\Gamma, A \vdash \delta$, and $\Gamma \vdash A, \delta$ by pushing the unused A through the proof until it is finally discarded by the generalized Ax rule or another axiom like $\top L$ or $\bot L$.

In terms of provability—the question of which sequents can conclude a valid proof tree—the versions of LK with explicit and implicit structural rules are the same. In the implicit system, exchange is invisible, contraction is a consequence of axiom and cut, and all weakening is pushed to the leaves. Furthermore, the two different versions of the axiom and cut rules are interderivable with respect to their different logics. The explicit Ax rule in Figure 4 is a special case of the implicit one above, whereas the implicit Ax rule can be expanded into many weakenings followed by the explicit rule. Likewise, the explicit Cut rule can be derived from the implicit rule by weakening the two premises until they match, whereas the implicit Cut rule can be derived from the explicit rule by contracting the result of the conclusion to remove the duplication. Therefore, up to provability, the choice between these two different styles for handling the structural properties of sequents in a classical or intuitionistic logic are a matter of taste.

On the same subject, it is also sensible to consider an alternate version of left implication introduction and right subtraction introduction that duplicates rather than splitting hypotheses and consequences among the premises in the style of our revised Cut above:

$$\frac{\Gamma \vdash A, \delta \quad \Gamma, B \vdash \delta}{\Gamma, A \supset B \vdash \delta}\ \supset L \qquad\qquad \frac{\Gamma \vdash A, \delta \quad \Gamma, B \vdash \delta}{\Gamma \vdash A - B, \delta}\ -R$$

In the presence of structural properties (either explicit or implicit), the two different $\supset L$ and $-R$ rules are equivalent up to provability. However, if we want a more refined view of the structural properties, as in sub-structural logics like linear logic (Girard, 1987), then these differences become more acute and must be considered carefully.

The implicit treatment of structural rules in LK corresponds to the variant of the core $\mu\tilde{\mu}$-calculus type system shown in Figure B1. In this formulation, there is no explicit use of structural rules in a typing derivation, but instead the structural properties of sequents follow from the natural scoping rules for static (co-)variables

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ VR \qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ VL$$

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ AR \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta}\ AL$$

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle v \| e \rangle : (\Gamma \vdash \Delta)}\ Cut$$

Fig. B1. Implicit (co-)variable scope in the core $\mu\tilde{\mu}$ typing.

in the $\mu\tilde{\mu}$-calculus, more closely analogous to the treatment of variable scope in the λ-calculus. During type checking, an output abstraction $\Gamma \vdash \mu\alpha.c : A \mid \delta$ (dually an input abstraction $\Gamma \mid \tilde{\mu}x.c : A \vdash \delta$) signals that the active type A may undergo an arbitrary number of structural rules depending on how α (dually x) is referenced in c. During execution, the behavior of structural rules are implicitly implemented by the substitution operation used by μ and $\tilde{\mu}$ reduction, corresponding to the structural steps of a cut elimination procedure.

As with logic of LK in, the choice between the two formulations of the scoping properties of $\mu\tilde{\mu}$ (co-)variables is somewhat arbitrary and a matter of taste. Since we are dealing with a calculus corresponding to classical logic, both treatments of structural properties are equivalent to each other in a sense—both formulations will admit type checking the same expressions, even in richer extensions of the core language. However, the two formulations have their own advantages. The implicit scoping presented in Figure B1 is concise and forgoes the redundancy of repeated rules, whereas the explicit scoping presented in Figure 6 easily allows for a more refined analysis of the structural properties and exploration of sub-structural calculi (Munch-Maccagnoni, 2009) corresponding to sub-structural logics that forbid certain uses of structural rules. The most important thing, though, is that *something* is done to express the scope of (co-)variables in the classical language $\mu\tilde{\mu}$.

Appendix C. Terminology and notation

Here, we give definitions for the common notations and terminology used in this article, namely: free variables (Section C1), substitution and α renaming (Section C2), and (deterministic) operational semantics, and (confluent) rewriting theory (Section C3).

To avoid too much redundancy, we will only consider the definitions for the dual calculi explicitly. The corresponding definitions for the core $\mu\tilde{\mu}$-calculus follow the appropriate subset of the dual calculi, and the relevant definitions for LK come from the following relationship between logical connectives and type constructors:

$$\top = 1 \qquad \bot = 0 \qquad A \wedge B = A \times B \qquad A \vee B = A + B \qquad A \supset B = A \to B$$

The rest of the logical connectives ($\neg$, $-$, $\forall$, $\exists X$) correspond to the type constructor of the same name. So the definitions of free variables, substitution, and α renaming for LK propositions can be translated from the corresponding definitions for dual calculi types.

C1 Free variables

The set of *free variables* of a type are defined by the following function:

$$FV : Type \to \wp(TypeVariable)$$

$$\begin{aligned} FV(X) &\triangleq \{X\} & FV(\neg A) &\triangleq FV(A) \\ FV(A \times B) &\triangleq FV(A) \cup FV(B) & FV(A + B) &\triangleq FV(A) \cup FV(B) \\ FV(A \to B) &\triangleq FV(A) \cup FV(B) & FV(A - B) &\triangleq FV(A) \cup FV(B) \\ FV(\forall X.A) &\triangleq FV(A) - \{X\} & FV(\exists X.A) &\triangleq FV(A) - \{X\} \end{aligned}$$

The main lines of note is that X has exactly itself in its set of free variables, and the quantifiers $\forall$ and $\exists$ *bind* their given type variable, thereby removing it from their set of free variables. In contrast, the free type variables of sequents is defined pointwise in terms of the above function by collecting together the union of all the free variables in each type:

$$\begin{aligned} &FV(x_n : A_n, \ldots, x_1 : A_1 \vdash \alpha_1 : B_1, \ldots, \alpha_m : B_m) \\ &\triangleq FV(A_n) \cup \cdots \cup FV(A_1) \cup FV(B_1) \cup \cdots \cup FV(B_m) \end{aligned}$$

The set of free variables in commands, terms, and co-terms follows a similar logic to the free variables in types, where (co-)variables are their own free sets, and binders (μ, $\tilde{\mu}$, λ, $\tilde{\lambda}$, Λ, $\tilde{\Lambda}$) remove their bound variable from the free set. For the purpose of representing the result of the FV function on commands and (co-)terms, we use the set *AnyVariable* which is the union of all variables ($x, \ldots$), co-variables ($\alpha, \ldots$), and type variables ($X, \ldots$):

$$FV : Command \to \wp(AnyVariable)$$

$$FV(\langle v \| e \rangle) \triangleq FV(v) \cup FV(e)$$

$$FV : Term \to \wp(AnyVariable) \qquad FV : CoTerm \to \wp(AnyVariable)$$

$$\begin{aligned} FV(x) &\triangleq \{x\} & FV(\alpha) &\triangleq \{\alpha\} \\ FV(\mu\alpha.c) &\triangleq FV(c) - \{\alpha\} & FV(\tilde{\mu}x.c) &\triangleq FV(c) - \{x\} \\ FV((v, v')) &\triangleq FV(v) \cup FV(v') & FV([e, e']) &\triangleq FV(e) \cup FV(e') \\ FV(\mathsf{in}_i(v)) &\triangleq FV(v) & FV(\mathsf{out}_i[e]) &\triangleq FV(e) \\ FV(\mathsf{not}(e)) &\triangleq FV(e) & FV(\mathsf{not}[v]) &\triangleq FV(v) \\ FV(\lambda x.v) &\triangleq FV(v) - \{x\} & FV(\tilde{\lambda}\alpha.e) &\triangleq FV(e) - \{\alpha\} \\ FV(e \cdot v) &\triangleq FV(e) \cup FV(v) & FV(v \cdot e) &\triangleq FV(v) \cup FV(e) \\ FV(\Lambda X.v) &\triangleq FV(v) - \{X\} & FV(\tilde{\Lambda}X.e) &\triangleq FV(e) - \{X\} \\ FV(B@v) &\triangleq FV(B) \cup FV(v) & FV(B@e) &\triangleq FV(B) \cup FV(e) \end{aligned}$$

C2 Substitution

The main obstacle in capture-avoiding substitution is to ensure that, when substituting underneath a binding form, the bound variable is not free in the expression being substituted under the binder, which is the action known as *capture*. The capture-avoiding substitution of C for Z in a type A, written $A\{C/Z\}$, is defined as the following partial function:

$$X\{C/Z\} \triangleq \begin{cases} X & \text{if } X \neq Z \\ C & \text{if } X = Z \end{cases} \qquad (\neg A)\{C/Z\} \triangleq \neg(A\{C/Z\})$$

$$(A \times B)\{C/Z\} \triangleq (A\{C/Z\}) \times (B\{C/Z\}) \qquad (A + B)\{C/Z\} \triangleq (A\{C/Z\}) + (B\{C/Z\})$$

$$(A \to B)\{C/Z\} \triangleq (A\{C/Z\}) \to (B\{C/Z\}) \qquad (A - B)\{C/Z\} \triangleq (A\{C/Z\}) - (B\{C/Z\})$$

$$(\forall X.A)\{C/Z\} \triangleq \forall X.(A\{C/Z\}) \quad \text{if } X \notin FV(C) \qquad (\exists X.A)\{C/Z\} \triangleq \exists X.(A\{C/Z\}) \quad \text{if } X \notin FV(C)$$

The main lines of interest is what happens during $X\{C/Z\}$, in which we must check whether X and Z are the same type variable to determine whether X is left unchanged or replaced with C, and during $(\forall X.A)\{C/Z\}$ and $(\exists X.A)\{C/Z\}$, in which we must check that X is not a free variable of C to avoid capture and fail to produce any result in that case.

At the level of programs, capture-avoiding substitution follows a similar pattern. Substituting values for variables is defined as:

$$\langle v \| e \rangle\{V/z\} \triangleq \langle v\{V/z\} \| e\{V/z\} \rangle$$

$$x\{V/z\} \triangleq \begin{cases} x & \text{if } x \neq z \\ V & \text{if } x = z \end{cases} \qquad \alpha\{V/z\} \triangleq \alpha$$

$$(\mu\alpha.c)\{V/z\} \triangleq \mu\alpha.(c\{V/z\}) \quad \text{if } \alpha \notin FV(V) \qquad [\tilde{\mu}x.c]\{V/z\} \triangleq \tilde{\mu}x.(c\{V/z\}) \quad \text{if } x \notin FV(V)$$

$$(v, v')\{V/z\} \triangleq (v\{V/z\}, v'\{V/z\}) \qquad [e, e']\{V/z\} \triangleq [e\{V/z\}, e'\{V/z\}]$$

$$\mathsf{in}_i(v)\{V/z\} \triangleq \mathsf{in}_i(v\{V/z\}) \qquad \mathsf{out}_i[e]\{V/z\} \triangleq \mathsf{out}_i[e\{V/z\}]$$

$$\mathsf{not}(e)\{V/z\} \triangleq \mathsf{not}(e\{V/z\}) \qquad \mathsf{not}[v]\{V/z\} \triangleq \mathsf{not}[v\{V/z\}]$$

$$(\lambda x.v)\{V/z\} \triangleq \lambda x.(v\{V/z\}) \quad \text{if } x \notin FV(V) \qquad [\tilde{\lambda}\alpha.e]\{V/z\} \triangleq \tilde{\lambda}\alpha.[e\{V/z\}] \quad \text{if } \alpha \notin FV(V)$$

$$(e \cdot v)\{V/z\} \triangleq [e\{V/z\}] \cdot (v\{V/z\}) \qquad [v \cdot e]\{V/z\} \triangleq (v\{V/z\}) \cdot [e\{V/z\}]$$

$$(\Lambda X.v)\{V/z\} \triangleq \Lambda X.(v\{V/z\}) \quad \text{if } X \notin FV(V) \qquad [\tilde{\Lambda}X.e]\{V/z\} \triangleq \tilde{\Lambda}X.[e\{V/z\}] \quad \text{if } X \notin FV(V)$$

$$(B@v)\{V/z\} \triangleq B@(v\{V/z\}) \qquad [B@e]\{V/z\} \triangleq B@[e\{V/z\}]$$

Here, we must decide whether x is replaced in $x\{V/z\}$, and be careful to avoid capture when going under the binders μ, $\tilde{\mu}$, λ, $\tilde{\lambda}$, Λ, $\tilde{\Lambda}$ by failing to produce a result in the worst case. Substituting co-values for co-variables is defined analogously to the above with the same free variable checks for capture-avoidance and where two

base cases for variables and co-variables are changed to the following:

$$x\{E/\gamma\} \triangleq x \qquad \alpha\{E/\gamma\} \triangleq \begin{cases} \alpha & \text{if } \alpha \neq \gamma \\ E & \text{if } \alpha = \gamma \end{cases}$$

Additionally, substituting types for type variables in commands and (co-)terms is also analogous to the above, where we have fewer places that we need to check for capture (because variables and co-variables cannot appear in types), but also need to distribute substitution into existential hiding ($B@v$) and universal instantiation ($B@e$) as follows:

$$x\{C/Z\} \triangleq x \qquad \alpha\{C/Z\} \triangleq \alpha$$

$$(\lambda x.v)\{C/Z\} \triangleq \lambda x.(v\{C/Z\}) \qquad (\tilde{\lambda}\alpha.e)\{C/Z\} \triangleq \tilde{\lambda}\alpha.(e\{C/Z\})$$

$$(B@v)\{C/Z\} \triangleq (B\{C/Z\})@(v\{C/Z\}) \qquad [B@e]\{C/Z\} \triangleq (B\{C/Z\})@[e\{C/Z\}]$$

To get around the partiality of the above substitution operations, we can use α renaming to replace bound variables and avoid undefined cases. Intuitively, for any instance of substitution, the primary expression being substituted into (be it a type, command, term, or co-term) can always be α renamed into an equivalent expression for which substitution has a definite result: for all A, Z, and C there is a $B =_\alpha A$ such that $B\{C/Z\}$ is defined. The α renaming rules of types are:

$$\forall X.A =_\alpha \forall Y.(A\{Y/X\}) \qquad \exists X.A =_\alpha \exists Y.(A\{Y/X\})$$

Similarly, the α renaming rules of terms and co-terms are:

$$\mu\alpha.c =_\alpha \mu\beta.(c\{\beta/\alpha\}) \qquad \tilde{\mu}x.c =_\alpha \tilde{\mu}y.[c\{y/x\}]$$

$$\lambda x.v =_\alpha \lambda y.(v\{y/x\}) \qquad \tilde{\lambda}\alpha.v =_\alpha \tilde{\lambda}\beta.[e\{\beta/\alpha\}]$$

$$\Lambda X.v =_\alpha \Lambda Y.(v\{Y/X\}) \qquad \tilde{\Lambda}X.e =_\alpha \tilde{\Lambda}Y.[e\{Y/X\}]$$

C3 Rewriting and operational semantics

The single-step operational relation ($c \mapsto c'$) is a relation between commands defined by the operational rules stated previously: $c \mapsto c'$ if any of the individual rules apply. The operational semantics ($c \mapsto\!\!\!\!\!\rightarrow c'$) is the reflexive, transitive closure of the single-step operational relation defined by the following inference rules:

$$\frac{c \mapsto c'}{c \mapsto\!\!\!\!\!\rightarrow c'}\ \textit{Inclusion} \qquad \frac{}{c \mapsto\!\!\!\!\!\rightarrow c}\ \textit{Reflexivity} \qquad \frac{c \mapsto\!\!\!\!\!\rightarrow c' \quad c' \mapsto\!\!\!\!\!\rightarrow c''}{c \mapsto\!\!\!\!\!\rightarrow c''}\ \textit{Transitivity}$$

An operational semantics is *deterministic* when each command can step to at most one other command, *i.e.*, $c \mapsto c_1$ and $c \mapsto c_2$ implies that c_1 and c_2 are identical commands.

The single-step rewriting relation ($\rightarrow$) is a family of relations between commands, terms, and co-terms, respectively, defined by the rewriting rules stated previously

and closed under the following *compatibility* inference rules:

$$\frac{c \to c'}{\mu\alpha.c \to \mu\alpha.c'} \qquad \frac{v \to v'}{\langle v \| e \rangle \to \langle v' \| e \rangle} \qquad \frac{e \to e'}{\langle v \| e \rangle \to \langle v \| e' \rangle} \qquad \frac{c \to c'}{\tilde{\mu}x.c \to \tilde{\mu}x.c'}$$

$$\frac{v_1 \to v_1'}{(v_1, v_2) \to (v_1', v_2)} \qquad \frac{v_2 \to v_2'}{(v_1, v_2) \to (v_1, v_2')} \qquad \frac{e_1 \to e_1'}{[e_1, e_2] \to [e_1', e_2]} \qquad \frac{e_2 \to e_2'}{[e_1, e_2] \to [e_1, e_2']}$$

$$\frac{v \to v'}{\mathsf{in}_i(v) \to \mathsf{in}_i(v')} \qquad \frac{e \to e'}{\mathsf{not}(e) \to \mathsf{not}(e')} \qquad \frac{v \to v'}{\mathsf{not}[v] \to \mathsf{not}[v']} \qquad \frac{e \to e'}{\mathrm{out}_i[e] \to \mathrm{out}_i[e']}$$

$$\frac{v \to v'}{\lambda x.v \to \lambda x.v'} \qquad \frac{v \to v'}{e \cdot v \to e \cdot v'} \qquad \frac{e \to e'}{e \cdot v \to e' \cdot v} \qquad \frac{v \to v'}{v \cdot e \to v' \cdot e} \qquad \frac{e \to e'}{v \cdot e \to v \cdot e'} \qquad \frac{e \to e'}{\tilde{\lambda}\alpha.e \to \tilde{\lambda}\alpha.e'}$$

$$\frac{v \to v'}{\Lambda X.v \to \Lambda X.v'} \qquad \frac{v \to v'}{B@v \to B@v'} \qquad \frac{e \to e'}{B@e \to B@e'} \qquad \frac{e \to e'}{\tilde{\Lambda}X.e \to \tilde{\Lambda}X.e'}$$

The rewriting theory ($\twoheadrightarrow$) is the reflexive, transitive closure of the family of single-step rewriting relation defined by analogous inference rules as for the operational semantics: an inclusion, reflexivity, and transitivity inference rule for each of commands, terms, and co-terms. A rewriting theory is *confluent* if any two divergent, many-step reductions join back together, *i.e.,* $c \twoheadrightarrow c_1$ and $c \twoheadrightarrow c_2$ implies that $c_1 \twoheadrightarrow c'$ and $c_2 \twoheadrightarrow c'$ for some c', and similarly for reductions on terms and co-terms.