Computational duality and the sequent calculus

Paul Downen
University of Oregon
pdownen@cs.uoregon.edu

Abstract

The correspondence between proving theorems and writing programs, known as the *Curry-Howard isomorphism* or the *proofs-as-programs* paradigm, has spurred the development of new tools like proof assistants, new fields like certified programming, and new advanced type features like dependent types. The isomorphism gives us a correspondence between Gentzen's natural deduction, a system for formalizing common mathematical reasoning, and Church's λ-calculus, one of the first models of computation and the foundation for functional programming languages. However, concepts found in modern programming, such as the provision of computational effects, combination of different programming paradigms and evaluation strategies, and definition of infinite objects still lack a strong foundation.

In the hopes of gaining a better understanding of these issues, we will look at an alternate form of logic, called the *sequent calculus*, through the Curry-Howard lens. Doing so leads us to a method of computation that heavily emphasizes the concept of *duality* and the interaction between opposites — production interacts with consumption; construction interacts with deconstruction. The symmetry of this framework naturally explains more complicated features of programming languages through relatively familiar concepts — binding a value to a variable is dual to manipulation over the flow of control in a program; case analysis over data structures in the functional paradigm is dual to dynamic dispatch in the object-oriented paradigm.

In this report, we survey the background of the connections between proofs and programs along with more modern advances. We begin by reviewing the traditional form of the Curry-Howard isomorphism that connect proof theory, category theory, and type theory, and has roots in the beginnings of computer science. We introduce the sequent calculus as an alternative form of logic, and show how the Curry-Howard isomorphism still applies and presents a different basis for expressing computation. We then show how the fundamental dilemma in the sequent calculus gives rise to a duality between *evaluation strategies*: strict languages are dual to lazy languages. Next, we discuss concepts that were developed in the setting of proof search, *polarization* and *focalization*, which give a foundation for data structures and pattern matching in programming languages. Taking a step back, we give a retrospective look at the history of these developments, and illustrate a generalized theory that unifies the different systems for the sequent calculus and more closely matches actual programming languages. Finally, we consider the long road ahead for using the sequent calculus as a basis for understanding programming, and discuss various future directions remaining in this line of work.

1. Introduction

As Alcmaeon (510BC) once said, most things come in pairs: left and right, even and odd, true and false, good and bad. Duality is a guiding force that reveals the existence of a counter entity with a totally different form, yet the two are still strongly related through their mutual opposition. Duality appears in proof theory: "true" and "false" are dual, "and" and "or" are dual. Duality appears in category theory: the opposite of a category is found by reversing its arrows, sums naturally arise as the dual of products. However, even though the theory of programming languages is closely connected to logic and categories, this kind of duality does not appear to arise so readily in the practice of programming. For example, sum types (disjoint unions) and pair types (structures) are related to dual concepts in logic and category theory. But in the realm of programming languages, the duality between these two concepts is not readily apparent, and languages typically present the two as unrelated features and often with different levels of expressiveness.

The situation is even worse for more complicated language features, where we have two concepts connected by duality that are both important in the theory and practice of programming, but one is well understood while the other is enigmatic and underdeveloped. In the case of recursion and looping, inductive data types (like lists and trees of arbitrary, but finite, size) are known to be dual to co-inductive[1] infinite processes (like streams of input or servers that are indefinitely available) [58]. However, whereas the programming languages behind proof assistants like Coq [67] have a sophisticated notion of induction, a lack of understanding about the computational foundations of co-induction is problematic. McBride [107] notes how the poor foundation for the computational interpretation of co-induction is a road block for the goal of program verification and correctness:

> We are obsessed with foundations partly because we are aware of a number of significant foundational problems that we've got to get right before we can do anything realistic. The thing I would think of in particular in that respect is co-induction and reasoning about co-recursive processes. That's currently, in all major implementations of type theory, a disaster. And if we're going to talk about real systems, we've got to actually have something sensible to say about that.

Not only is this mismatch between the foundations of the two features aesthetically unpleasing, but it has real consequences on the application of theory for solving practical problems. Perhaps the reason for this disparity and disconnect between related features is a symptom of the fact that they are not considered with the right frame of mind.

Our main philosophy for approaching these questions is known as the *Curry-Howard isomorphism* [25, 30, 65] or *proofs-as-programs* paradigm. The Curry-Howard isomorphism reveals a deep and profound connection between three different fields of study — logic, category theory, and programming languages — where quite literally mathematical proofs *are* algorithmic pro-

[1] In general, adding the prefix "co-" to a term or concept means "the dual of that thing."

grams. The canonical example of the isomorphism is the correspondence between Gentzen's natural deduction [47], a system for formalizing common mathematical reasoning, and Church's λ-calculus [19], one of the first models of computation and the foundation for functional programming languages. However, the λ-calculus not an ideal setting for studying duality in computation. Dualities that are simple in other settings, like the De Morgan duals in logic or products and sums in category theory, are far from obvious in the λ-calculus. The problem is related to the lack of symmetry in natural deduction: in natural deduction all the emphasis is on concluding true statements and in the λ-calculus all the emphasis is on producing results.

In contrast, the sequent calculus, introduced by Gentzen [47] simultaneously with natural deduction, is a system logic about dualities. In these formal systems of logic, equal attention is given to falsity and truth, to assumptions and conclusions, such that there is perfect symmetry. These symmetries are even more apparent in the related system of linear logic by Girard [51]. When interpreted as a programming language, the sequent calculus reveals 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* which is representative of a program that creates information, and a *consumer* which is representative of an environment or context that observes information. Computation then occurs as a protocol of communication that allows a producer and consumer to speak to one another, and as a negotiation for resolving conflicts between the two. This two-party, protocol-based style of computation lends itself to naturally describe a lower-level view of computation than expressed by the λ-calculus. In particular, programs in the sequent calculus can also be seen as configurations of an abstract machine, in which the evaluation context is reified as a syntactic object that may be directly manipulated. The sequent calculus is also a natural language for expressing effects, as its low-level nature inherently gives a language for manipulating control flow equivalent in power to `callcc` from Scheme [74].

In this report, we will survey the dualities in computation from the perspective of the Curry-Howard isomorphism, with a particular focus on the connections between logic and programming. We will begin with a review of the λ-calculus and its connection with natural deduction and Cartesian closed categories (Section 2). Next, we will introduce the sequent calculus as an alternate logic to natural deduction and an alternate language to the λ-calculus (Section 4). We will discuss the duality between *evaluation strategies*, showing the relationship between programs in strict (like ML) and lazy (like Haskell) languages (Section 5). We will look at how the concepts of *polarization* and *focalization* that arose in the area of proof search have an impact on the meaning of programs and the definition of programming languages (Section 6), particularly on the notions of case analysis and pattern matching from functional programming languages. Finally, we will give a retrospective look at the current state of the art in the field and present some current work on generalizing the various theories from a semantic perspective (Section 7), and then consider some open problems in the field (Section 8).

2. Natural deduction: logic, programming, and categories

The roots of the connection between the foundations of mathematics and computation go back to the early 1900s, when Hilbert posed the *decision problem* by asking if there is an effectively calculable procedure which can decide whether a logical statement is true or false. This problem, and its negative answer, prompted for a rigorous definition of "effectively computable" from Church [20], Turing [110], and Gödel [55]. Later on, a much deeper connection between models of computation and formalized logic was independently discovered many times [25, 30, 65]. The most typical form of this connection, now known as the *Curry-Howard isomorphism*, gives a structural isomorphism between Church's λ-calculus [19], a system for computing with functions, and Gentzen's natural deduction [47], a system for formalizing mathematical logic and reasoning. Additionally, this correspondence also includes an algebraic structure known as *cartesian closed categories* [77]. To illustrate the connection between logic, programming, and categories, we will review the three systems and show how they reveal core concepts in different ways. In particular, we will see how two principles important for characterizing the meaning of various structures, which we will call β and η, arise independently in each field of study.

2.1 Logic

In 1935, Gentzen [47] formalized an intuitive model of logical reasoning called *natural deduction*, as it aimed to symbolically model the "natural" way that mathematicians reason about proofs. A proof in natural deduction is a tree-like structure made up of several *inferences*:

$$\frac{\begin{array}{cccc}\vdots & \vdots & & \vdots \\ H_1 & H_2 & \ldots & H_n\end{array}}{J}$$

where we conclude the *deduction* J from proofs of the *premises* $H_1, H_2, \ldots, H_3$. The conclusion J and premises H_i are all *judgments* that make statement about logical *propositions*, such as "0 is greater than 1." For example, if we let $A, B, C, \ldots$ range over the set of propositions, then we may make the basic judgment that A is true. We may also make more complex judgments, such as a *hypothetical* judgment:

$$A_1, A_2, \ldots, A_n \vdash B$$

pronounced "A_1, A_2, ... and A_n entail B," which states that assuming each of $A_1, A_2, \ldots, A_n$ are true then B must be true. The local hypothesis, to the left of entailment, are collectively referred to as Γ. With this form of hypothetical judgments, we may give our first *inference rule* for forming proofs by concluding that the proposition A must be true if was assumed to be true among our hypothesis:

$$\frac{}{\Gamma, A \vdash A}\ Axiom$$

Note that the order of hypothesis does not matter,[2] so A may appear anywhere in the list of hypothesis to the left of entailment. From here on, in a slight deviation from Gentzen's original presentation of natural deduction, we will favor working with hypothetical judgments.

The system of natural deduction forms new propositions by putting together other existing propositions with *connectives*, which are the logical glue for putting together the basic building blocks. For example, the idea of logical conjunction can be expressed formally as a connective, written $A \wedge B$ and read "A and B," along with some associated rules of inference for building proofs out of conjunction. On the one hand, in order to deduce that $A \wedge B$ is true (under some hypothesis Γ) we may use the *introduction rule* $\wedge I$:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge I$$

[2] More specifically, the hypothesis form a *multiset*, so that their order is irrelevant but the number of occurrences of the same proposition matters.

$$A, B, C \in \mathit{Proposition} ::= X \mid A \wedge B \mid A \supset B \mid \top$$
$$\Gamma \in \mathit{Hypothesis} ::= A_1, \ldots, A_n$$
$$\mathit{Judgment} ::= \Gamma \vdash A$$

$$\frac{}{\Gamma, A \vdash A}\ Ax$$

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

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset I \qquad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \supset E$$

$$\frac{}{\Gamma \vdash \top}\ \top I \qquad \text{no } \top E \text{ rule}$$

Figure 1. Natural deduction with conjunction, implication, and truth.

That is to say, if we have a proof that A is true and a proof that B is true (both under the hypothesis Γ), then we have a proof that $A \wedge B$ is true (also under the hypothesis Γ). On the other hand, in order to use the fact that $A \wedge B$ is true we may use one of the *elimination rules* $\wedge E_1$ or $\wedge E_2$:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge E_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge E_2$$

That is to say, if we have a proof that $A \wedge B$ is true (under the hypothesis Γ), then it must be the case that A is true (also under the hypothesis Γ), and similarly for B.

As another example, we can also give an account of logical implication as a connective in natural deduction, written $A \supset B$ and read "A implies B" or "if A then B," in a similar fashion. In order to deduce that $A \supset B$ is true we may use the introduction rule $\supset I$ for implication:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset I$$

Notice that the introduction rule for implication has a more interesting interaction with the local hypothesis. If we can prove that B is true using A as part of our local hypothesis, then we can conclude that $A \supset B$ is true under the hypothesis Γ alone. Once we have a proof of $A \supset B$, we may make use of it with the elimination rule $\supset E$ for implication:

$$\frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \supset E$$

This is a formulation of the traditional reasoning principle *modus ponens*: from the fact that A implies B is true and we have a proof that A is true as well, then B must be true.

Finally, we can give a connective that internalizes the notion of truth or validity into the system, written $\top$ and pronounced "true." The rules for $\top$ are very basic compared to the other connectives. We may always deduce that $\top$ is true, regardless of our local hypothesis, using the introduction $\top I$ rule:

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

On the other hand, there is nothing we can do with a proof that $\top$ is true. In other words, "nothing in, nothing out." The rules presented so far are summarized in Figure 1. The propositions may be some variable, X, which stands in for some unknown proposition, or the propositions formed by the $\wedge$, $\supset$, and $\top$ connectives. Hypothesis are an unordered list of propositions, and the judgments are hypothetical.

Example 1. As an example of how inference trees in natural deduction correspond to proofs of propositions, consider how we might build a proof of $((A \wedge B) \wedge C) \supset (B \wedge A)$. To start searching for a proof, we may begin with our goal $\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)$ at the bottom of the proof tree, and then try to simplify the goal by applying the implication introduction rule "bottom up:"

$$\frac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax \\ \vdots \\ (A \wedge B) \wedge C \vdash B \wedge A}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\ \supset I$$

This move adds the assumption $(A \wedge B) \wedge C$ to our local hypothesis for the duration of the proof, which we may make use of to finish off the proof at the top by the Ax rule. We are still obligated to fill in the missing gap between Ax and $\supset I$, but our job is now a bit easier, since we have gotten rid of the $\supset$ connective from the consequence in the goal. Next, we can try to simplify the goal again by applying the conjunction introduction rule to get rid of the $\wedge$ in the goal:

$$\frac{\dfrac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax \quad \dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax}{\begin{array}{cc} \vdots & \vdots \\ (A \wedge B) \wedge C \vdash B & (A \wedge B) \wedge C \vdash A \end{array}}\ \wedge I}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\ \supset I$$

We now have two sub-proofs to complete: a deduction of B and a deduction of A from our local hypothesis $(A \wedge B) \wedge C$. At this point, the consequences of our goals are as simple as they can be — they no longer contain any connectives for us to work with. Therefore, we instead switch to work "top down" from our assumptions. We are allowed to assume $(A \wedge B) \wedge C$, so let's eliminate the unnecessary proposition C using a conjunction elimination rule in both

sub-proofs:

$$\dfrac{\dfrac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax}{(A \wedge B) \wedge C \vdash A \wedge B}\ \wedge E_1}{\begin{array}{c}\vdots \\ (A \wedge B) \wedge C \vdash B\end{array}} \quad \dfrac{\dfrac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax}{(A \wedge B) \wedge C \vdash A \wedge B}\ \wedge E_1}{\begin{array}{c}\vdots \\ (A \wedge B) \wedge C \vdash A\end{array}}$$

$$\dfrac{(A \wedge B) \wedge C \vdash B \qquad (A \wedge B) \wedge C \vdash A}{\dfrac{(A \wedge B) \wedge C \vdash B \wedge A}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\ \supset I}\ \wedge I$$

We can now finish off the entire proof by using conjunction elimination "top down" in both sub-proofs, closing the gap between assumptions and conclusions:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax}{(A \wedge B) \wedge C \vdash A \wedge B}\ \wedge E_1}{(A \wedge B) \wedge C \vdash B}\ \wedge E_2 \qquad \dfrac{\dfrac{\dfrac{}{(A \wedge B) \wedge C \vdash (A \wedge B) \wedge C}\ Ax}{(A \wedge B) \wedge C \vdash A \wedge B}\ \wedge E_1}{(A \wedge B) \wedge C \vdash A}\ \wedge E_1}{\dfrac{(A \wedge B) \wedge C \vdash B \wedge A}{\vdash ((A \wedge B) \wedge C) \supset (B \wedge A)}\ \supset I}\ \wedge I$$

Since there are no unjustified branches at the top of the tree (every leaf is closed off by the Ax rule) and there are no longer any gaps in the proof, we have completed the deduction of our goal. *End example* 1.

Remark 1. Some presentations of hypothetical natural deduction prefer to be more explicit about the use of hypothesis in the structure of a proof. Most typically, they will replace the Ax rule in Figure 1 with a simpler rule that concludes A only when we assumed that A alone holds:

$$\dfrac{}{A \vdash A}\ Ax$$

By restricting the use of our assumptions, we need to add a few more *structural rules* that add back the additional structural properties of the hypothesis so that we may prove the same things. In particular, we need to allow for the addition of unnecessary hypothesis, called *weakening*:

$$\dfrac{\Gamma \vdash C}{\Gamma, A \vdash C}\ Weakening$$

which says that if we could deduce C from Γ, then surely we can still deduce C when we add A to Γ. From these two rules, the more general Ax rule of Figure 1 is derivable by applying weakening as many times as necessary to delete Γ. We also need the ability to merge multiple assumptions of the same proposition, called *contraction*:

$$\dfrac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ Contraction$$

Finally, some presentations of formal logic are even more explicit, and make Γ a list, so that the order of hypothesis matters. In this case, we would also need a rule that swaps propositions in Γ so that they occur in the correct place, called *exchange*:

$$\dfrac{\Gamma, B, A, \Gamma' \vdash C}{\Gamma, A, B, \Gamma' \vdash C}\ Exchange$$

For our purposes, we will not be considering explicit use of the exchange rule, and will assume that the order of a list of premises in a hypothesis does not matter.

Note that in the definition of natural deduction given in Figure 1, the weakening, contraction, and exchange rules are all derivable as global properties of the system. For example, if we have a proof tree $\mathcal{D}$ that deduces the judgment:

$$\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \vdash C\end{array}$$

then the same proof, using the rules of Figure 1, deduces the weakened judgment:

$$\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma, A \vdash C\end{array}$$

For our purposes, we will prefer the more implicit presentation of hypothetical judgments. *End remark* 1.

Now that we have some connectives and their rules of inference in our system of natural deduction, we would like to have some assurance that what we have defined is sensible in some way. Dummett [33] introduced a notion of *logical harmony* which guarantees that the inference rules are meaningful. Just like Goldilocks, we want rules that are neither too strong (leading to an inconsistent logic) nor too weak (leading to gaps in our knowledge), but are instead just right. The notion of logical harmony for a particular connective can be broken down into two properties of that connective's inference rules: *local soundness* and *local completeness* [90].

For a single logical connective, we need to check that its inference rules are not too strong, meaning that they are *locally sound*, so that the results of the elimination rules are always justified. In other words, everything we deduce can be justified by some proof so that we cannot get out more than what we put in. This notion is expressed in terms of proof manipulations: a proof in which an introduction is immediately followed by an elimination can be transformed to a more direct proof. On the one hand, in the case of conjunction, if we follow $\wedge I$ with $\wedge E_1$, then we can perform the following reduction:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \vdots \\ \Gamma \vdash A\end{array} \quad \begin{array}{c}\mathcal{D}_2 \\ \vdots \\ \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\ \wedge I}{\Gamma \vdash A}\ \wedge E_1 \quad \Rightarrow \quad \begin{array}{c}\mathcal{D}_1 \\ \vdots \\ \Gamma \vdash A\end{array}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ stand for proofs that deduce $\Gamma \vdash A$ and $\Gamma \vdash B$, respectively. If we had forgotten to include the first premise $\Gamma \vdash A$ in the $\wedge I$ rule, then this soundness reduction would have no proof to justify its conclusion. On the other hand, if we follow $\wedge I$ with $\wedge E_2$, then we have a similar reduction:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \vdots \\ \Gamma \vdash A\end{array} \quad \begin{array}{c}\mathcal{D}_2 \\ \vdots \\ \Gamma \vdash B\end{array}}{\Gamma \vdash A \wedge B}\ \wedge I}{\Gamma \vdash B}\ \wedge E_2 \quad \Rightarrow \quad \begin{array}{c}\mathcal{D}_2 \\ \vdots \\ \Gamma \vdash B\end{array}$$

Additionally, we should also ensure that the rules are not too weak, so all the information that goes into a proof can still be accessed somehow. In this respect, we say that the inference rules for a logical connective are *locally complete* if they are strong enough to break an arbitrary proof ending with that connective into pieces and then put it back together again. For the rules given for conjunction, this is expressed as the following proof transformation:

$$\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \vdash A \wedge B\end{array} \quad \Rightarrow \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \vdash A \wedge B\end{array}}{\Gamma \vdash A}\ \wedge E_1 \quad \dfrac{\begin{array}{c}\mathcal{D} \\ \vdots \\ \Gamma \vdash A \wedge B\end{array}}{\Gamma \vdash B}\ \wedge E_2}{\Gamma \vdash A \wedge B}\ \wedge I$$

If we had forgotten the elimination rule $\wedge E_2$, then local completeness would fail because we would not have enough information to satisfy the premise of the $\wedge I$ introduction rule. As a result, the rules will still be sound but we would be unable to prove a basic tautology like $A \wedge B \vdash B \wedge A$, which should hold by our intuitive interpretation of $A \wedge B$.

We also have local soundness and completeness for the inference rules of logical implication, although they require a few properties about the system as a whole. For local soundness, we can reduce $\supset I$ immediately followed by $\supset E$ by using the meaning of the hypothetical judgment:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash B\end{array}}{\Gamma \vdash A \supset B}\supset I \qquad \begin{array}{c}\mathcal{E}\\ \vdots\\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\supset E \quad \Rightarrow \quad \begin{array}{c}\mathcal{D}\{\mathcal{E}/A\}\\ \vdots\\ \Gamma \vdash B\end{array}$$

where $\mathcal{D}\{\mathcal{E}/A\}$ is the *substitution* of the proof $\mathcal{E}$ for any uses of the local hypothesis A in $\mathcal{D}$. The substitution gives us a modified proof that no longer needs that particular local assumption of A, since any time the $Axiom$ rule was used to deduce A, we instead use $\mathcal{E}$ as a proof of A. For local completeness, we can expand an arbitrary proof of $A \supset B$ as follows:

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma \vdash A \supset B\end{array} \quad \Rightarrow \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash A \supset B\end{array} \qquad \dfrac{}{\Gamma, A \vdash A}\,Axiom}{\Gamma, A \vdash B}\supset E}{\Gamma \vdash A \supset B}\supset I$$

Notice that on the right hand side, we had to use the *weakening* principle (see Remark 1) to add A as an additional, unused hypothesis to the proof $\mathcal{D}$.

Demonstrating local soundness and completeness for the inference rules of logical truth may be deceptively basic. Since there is no $\top E$ rule, local soundness is trivially true: there is no possible way to have a proof where $\top I$ is followed by $\top E$, and so local soundness is vacuous. On the other hand, we still have to demonstrate local completeness by transforming an arbitrary proof of $\top$ into one ending in the $\top I$ rule. However, because the $\top I$ rule is always available, this transformation just throws away the original, unnecessary proof:

$$\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma \vdash \top\end{array} \quad \Rightarrow \quad \dfrac{}{\Gamma \vdash \top}\top I$$

Therefore, we can be sure that the rules for logical truth are sensible.

2.2 Programming languages

The λ-calculus, defined by Church [19] in the 1930s, is a remarkably simple yet powerful model of computation. The original language is defined by only three constructions: abstracting a program with respect to a parameter (*i.e.,* a function term: $\lambda x.M$), reference to a parameter (*i.e.,* a variable term: x), and applying a program to an argument (*i.e.,* a function application term: $M\ N$). Despite this simple list of features, the untyped λ-calculus is a complete model of computation, equivalent to a Turing machine. It is often used as a foundation for understanding the static and dynamic semantics of programming languages as well as a platform to experiment with new features. In particular, functional programming languages are sometimes thought of as notational convenience that desugars to an underlying core language based on the λ-calculus.

The dynamic behavior of the λ-calculus is defined by three principles. The most basic principle is called α equivalence, and it asserts that the particular choice of names for variables does not matter; the defining characteristic for a variable is where it was introduced, enforcing a notion of static scope. For instance, the identity function that immediately returns its argument unchanged may be written as either $\lambda x.x$ or $\lambda y.y$, both of which are considered α equivalent. The next principle is called β equivalence, and it provides the primary computational force of the λ-calculus. Given a λ-abstraction (*i.e.,* a term of the form $\lambda x.M$) that is applied to an argument, we may calculate the result by substituting the argument for every reference to the λ-abstraction's parameter:

$$(\lambda x.M)\ N =_\beta M\,\{N/x\}$$

The term $M\,\{N/x\}$ is notation for performing the usual notion of capture-avoiding substitution of N for the variable x in M, such that the static bindings of variables are preserved. The final principle is called η equivalence, and it gives a notion of extensionality for functions. In essence, a λ-abstraction that does nothing but forward its parameter to another function is the same as that original function:

$$M =_\eta (\lambda x.M\ x)$$

Note that this rule is restricted so that M may not refer to the variable x introduced by the abstraction, again to preserve static binding.

Even though the λ-calculus with functions alone is sufficient for modeling all computable functions, it is often useful to enrich the language with other constructs. For instance, we may add pairs to the λ-calculus by giving a way to build a pair out of two terms other, (M, N), as well as projecting out the first and second components from a pair, $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$. We may define the dynamic behavior of pairs in the λ-calculus similar to the way we did for functions. Since pairs do not introduce any parameters, they are a bit simpler than functions. The main computational principle, by analogy called β equivalence for pairs, extracts a component out of a pair when it is demanded:

$$\mathsf{fst}(M, N) =_\beta M \qquad\qquad \mathsf{snd}(M, N) =_\beta N$$

The extensionality principle, called η equivalence for pairs, equates a term M with the pair formed out of the first and second components of M:

$$M =_\eta (\mathsf{fst}\ M, \mathsf{snd}\ M)$$

We can also add a unit value to the λ-calculus, which is a nullary form of pair that contains no elements, written $()$, that expresses lack of any interesting information. On the one hand, since the unit value contains no elements, there are no projections out of it, and therefore there it has meaningful β equivalence. On the other hand, the extensionality principle is quite strong, and the η equivalence for the unit equates a term M with the canonical unit value:

$$M =_\eta ()$$

This rule can be read as the nullary version of the η rule for pairs, where M did not contain any interesting information, and so it is irrelevant.

So far, we have only considered the dynamic meaning of the λ-calculus without any mention of its static properties. In particular, now that we have both functions and pairs, we may want to statically check and rule out programs that might "go wrong" during calculation. For instance, if we apply a pair to an argument, $(M, M')\ N$, then there is nothing we can do reduce this program any further. Likewise, it is nonsensical to ask for the first component of a function, $\mathsf{fst}(\lambda x.M)$. We may rule out such ill-behaved programs by using a *type system* that guarantees that such situations never occur by assigning a type to programming constructs and ensuring that programs are used in accordance to their types. For instance, we may give a function type, $A \to B$ to λ-abstractions as follows:

$$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\to I$$

where $M : B$ means that M has type B, and the environment Γ keeps track of the variables in scope along with their types.

$$
\begin{aligned}
A, B, C \in Type &::= X \mid A * B \mid A \to B \mid 1 \\
M, N \in Term &::= x \mid (M, N) \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \lambda x.M \mid M\ N \\
\Gamma \in Environment &::= x_1 : A_1, \ldots, x_n : A_n \\
Judgment &::= \Gamma \vdash M : A
\end{aligned}
$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ Var$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A * B}\ *I \qquad \frac{\Gamma \vdash M : A * B}{\Gamma \vdash \mathsf{fst}\, M : A}\ *E_1 \qquad \frac{\Gamma \vdash M : A * B}{\Gamma \vdash \mathsf{snd}\, M : B}\ *E_2$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to I \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to E$$

$$\frac{}{\Gamma \vdash () : 1}\ 1I \qquad \text{no } 1E \text{ rule}$$

(β^*)	$\begin{cases} \mathsf{fst}(M, N) & = M \\ \mathsf{snd}(M, N) & = N \end{cases}$	(η^*)	$M = (\mathsf{fst}\, M, \mathsf{snd}\, M)$
$(\beta^{\to})$	$(\lambda x.M)\ N = M\,\{N/x\}$	$(\eta^{\to})$	$M = (\lambda x.M\ x)$
(β^1)	no β^1 rule	(η^1)	$M = ()$

Figure 2. The simply typed λ-calculus with pairs and units.

Notice that if $\lambda x.M$ is a function taking an argument A, then by the premise of this rule, the body of the function M sees x in its environment. Having given a rule for introducing a term of function type, we can now restrict application to only occur for terms of the correct type:

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \to E$$

This rule ensures that if we apply a term M to an argument, then M must be of function type. Likewise, we may give a pair type, $A * B$, to the introduction of a pairs of terms:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A * B}\ *I$$

as well as restricting first and second project to only be used on terms of a pair type:

$$\frac{\Gamma \vdash M : A * B}{\Gamma \vdash \mathsf{fst}\, M : A}\ *E_1 \qquad \frac{\Gamma \vdash M : A * B}{\Gamma \vdash \mathsf{snd}\, M : B}\ *E_2$$

The unit type, 1, is a degenerate form of the pair type with a single canonical introduction:

$$\frac{}{\Gamma \vdash () : 1}\ 1I$$

and no other typing rules. And finally, we need a rule that allows us to refer to a variable in scope:

$$\frac{}{\Gamma, x : A \vdash x : A}\ Var$$

With all these rules in place, nonsensical programs like $\mathsf{fst}(\lambda x.M)$ are now ruled out, since they cannot be given a type. The static and semantics of this simply typed λ-calculus are summarized in Figure 2.

Remark 2. We should note that some care needs to be taken during a type derivation to make sure that the distinction between variables in different scopes is clear. For example, consider the following derivation of the identity function $(\lambda x.x)$ in an environment that already introduces the variable x:

$$\frac{\dfrac{}{x : A, x : A \vdash x : A}\ Var}{x : A \vdash \lambda x.x : A \to A}\ \to I$$

In the use of the Var rule, it is not clear which variable in the environment $x : A, x : A$ is being referred to. In the setting of programming, this distinction matters; whether the function returns its argument or something else from the outside environment is an important difference! Therefore, when we are dealing with variable environments, we maintain the convention that all the variables in the environment are distinct, so that it is unambiguous which one we wish to refer to.

There are two solutions to avoid the problem of identically named variables in the environment. The first is to forbid any inference rule from adding a variable to an environment that already contains a variable of the same name. To work around this restriction, we may use α equivalence on the term in question so that the locally bound variable is unique. By converting $\lambda x.x$ to $\lambda y.y$, we have:

$$\frac{\dfrac{}{x : A, y : A \vdash y : A}\ Var}{x : A \vdash \lambda y.y : A \to A}\ \to I$$

which is no longer ambiguous. On the other hand, we may state that the operation of adding a variable to an environment, $\Gamma, x : A$, *overrides* any variable with the same name in Γ. This enforces a notion of variable *shadowing* from programming languages, and ensures that environments never contain duplicate variables without the need for α equivalence. For example, the typing derivation

using shadowing would proceed as follows:

$$\dfrac{\dfrac{}{x : A \vdash x : A}\ Var}{x : A \vdash \lambda x.x : A \to A} \to I$$

where it is clear that the x referred to by the Var is the one introduced by the λ abstraction, since it must have overrode the x that came in from the environment in the deduction. *End remark* 2.

Example 2. For an example of how to program in the λ-calculus, consider the following function which takes a nested pair, of type $(A * B) * C$, and swaps the inner first and second components:

$$\lambda x.(\mathsf{snd}(\mathsf{fst}(x)), \mathsf{fst}(\mathsf{fst}(x)))$$

We can check that this function is indeed well-typed, using the typing rules given in Figure 2, by completing the typing derivation in Figure 3. Notice how the type derivation bears a structural resemblance to the proof of $((A \wedge B) \wedge C) \supset (B \wedge A)$ given in Example 1. In addition, we can check that this function behaves the way we intended by applying it to a nested pair, $((M_1, M_2), M_3)$, and evaluate it using the equations given in Figure 2:

$$\begin{aligned}
&(\lambda x.(\mathsf{snd}(\mathsf{fst}(x)), \mathsf{fst}(\mathsf{fst}(x)))) \ ((M_1, M_2), M_3) \\
&=_{\beta\to} \ (\mathsf{snd}(\mathsf{fst}((M_1, M_2), M_3)), \mathsf{fst}(\mathsf{fst}((M_1, M_2), M_3))) \\
&=_{\beta\times} \ (\mathsf{snd}(M_1, M_2), \mathsf{fst}(M_1, M_2)) \\
&=_{\beta\times} \ (M_2, M_1)
\end{aligned}$$

which confirms that this is the correct function. *End example* 2.

2.3 Category theory

Category theory is an abstract field of study that formalizes mathematics, from set theory to topology to abstract algebra. The main focus is on *categories*, which are made up of:

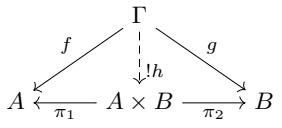
- some objects ("points"),
- some morphisms between those objects ("arrows"),
- a trivial morphism for every object ("identity"), and
- the ability to compose together any connected morphisms ("composition"),

along with some laws about identity and composition. For example, if we have a category with:

- objects A, B, and C,
- a morphism f from A to B (written $f : A \to B$), and
- a morphism g from B to C (written $g : B \to C$),

then $g \circ f$ is a morphism from A to C in that category. Of particular interest for our purposes is a categorical structure known as a *cartesian closed category*, but in order to talk about this concept we must first introduce cartesian products and exponentials in a categorical setting.

A binary *product* in category theory is an algebraic reformulation of the usual notion of cartesian products from set theory. The goal is to characterize the product of the objects A and B in a category, written $A \times B$. The product object $A \times B$ must have projection morphisms, $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ out of the product. $A \times B$ and its projections form a product if, for any other object Γ and morphisms $f : \Gamma \to A$ and $g : \Gamma \to B$, then there exists a *unique* morphism $h : \Gamma \to A \times B$ such that the following diagram commutes:[3]



By saying the diagram commutes, we mean that $\pi_1 \circ h$ is the same as f and $\pi_2 \circ h$ is the same as g. Since h is uniquely determined by f and g, it may be denoted by the notation (f, g). Therefore, we may alternative express the commutation of the above diagram with the following equations:

$$\pi_1 \circ (f, g) = f$$
$$\pi_2 \circ (f, g) = g$$

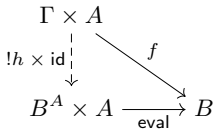
We can also capture the uniqueness condition as an equation which expresses the fact that given an arbitrary arrow into $A \times B$, this morphism must coincide with the morphism we get from the definition of the product:

$$(\pi_1 \circ h, \pi_2 \circ h) = h$$

An *exponential* object in category theory is a bit more complex, and relies on the existence of products in their definition. Intuitively, exponentials internalize morphisms in the category into an object. If we instead write an arrow $A \to B$ as B^A, then the definition of an exponential object may be seen as capturing the following algebraic property of exponents and products:

$$B^{(A_1 \times A_2)} = (B^{A_1})^{A_2}$$

The exponential object of A and B, written B^A, along with an evaluation morphism $\mathsf{eval} : B^A \times A \to B$ is defined so that for any object Γ and morphism $f : \Gamma \times A \to B$, there exists a *unique* morphism $h : \Gamma \to B^A$ such that the following diagram commutes:



The notation $h \times \mathsf{id}$ means to put h in parallel with the identity morphism id,[4] so that the left component of $\Gamma \times A$ is fed through h and the right component is unchanged. As before, since h is uniquely determined by f, we may denote this single morphism by $\mathsf{curry}(f)$. Similarly as with products, we may boil down the diagrammatic definition into a set of equations that express the same meaning. The commutation of the diagram is captured by the following equation:

$$\mathsf{eval} \circ (\mathsf{curry}(f) \times \mathsf{id}) = f$$

and the uniqueness condition on h may be expressed as:

$$\mathsf{curry}(\mathsf{eval} \circ (h \times \mathsf{id})) = h$$

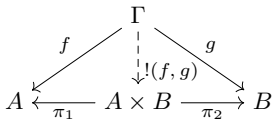
The last ingredient to a cartesian closed category is the existence of a *terminal object*. The terminal object of a category has a unique arrow coming from every other object in that category. All three of these structures are summarized in Figure 4. Any category which has a product and exponent for any pair of objects A and B, $A \times B$ and B^A, along with a terminal object, is called *cartesian closed*. Such a category is "closed" in the sense that every morphism

[3] The dashed arrow means that the morphism h must exist, given the other arrows in the diagram, and the exclamation mark means that there is only one possible arrow h.

[4] The parallel composition of morphisms can be defined in terms of the primitive operations for products, giving us $h_1 \times h_2 = (h_1 \circ \pi_1, h_2 \circ \pi_2)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : (A * B) * C \vdash x : (A * B) * C}\, Var}{x : (A * B) * C \vdash \mathsf{fst}(x) : A * B}\, *E_1}{x : (A * B) * C \vdash \mathsf{snd}(\mathsf{fst}(x)) : B}\, *E_2 \quad \dfrac{\dfrac{\dfrac{}{x : (A * B) * C \vdash x : (A * B) * C}\, Var}{x : (A * B) * C \vdash \mathsf{fst}(x) : A * B}\, *E_1}{x : (A * B) * C \vdash \mathsf{fst}(\mathsf{fst}(x)) : A}\, *E_1}{\dfrac{x : (A * B) * C \vdash (\mathsf{snd}(\mathsf{fst}(x)), \mathsf{fst}(\mathsf{fst}(x))) : B * A}{\vdash \lambda x.(\mathsf{snd}(\mathsf{fst}(x)), \mathsf{fst}(\mathsf{fst}(x))) : ((A * B) * C) \to (B * A)}\, \to I}\, *I$$

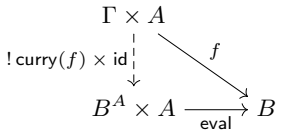
Figure 3. Type derivation of the λ-calculus term: $\lambda x.(\mathsf{snd}(\mathsf{fst}(x)), \mathsf{fst}(\mathsf{fst}(x)))$.



$$\pi_1 \circ (f, g) = f \qquad \pi_2 \circ (f, g) = g$$
$$h = (\pi_1 \circ h, \pi_2 \circ h)$$

$$\pi_1 : A \times B \to A \quad \pi_2 : A \times B \to B$$

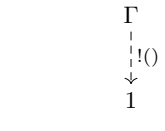
$$\frac{f : \Gamma \to A \quad g : \Gamma \to B}{(f, g) : \Gamma \to A \times B}$$



$$\mathsf{eval} \circ (\mathsf{curry}(f) \times \mathsf{id}) = f$$
$$h = \mathsf{curry}(\mathsf{eval} \circ (h \times \mathsf{id}))$$

$$\mathsf{eval} : B^A \times A \to B$$

$$\frac{f : \Gamma \times A \to B}{\mathsf{curry}(f) : \Gamma \to B^A}$$



$$h = ()$$

$$\frac{}{() : \Gamma \to 1}$$

Figure 4. Definitions of categorical products ($A \times B$), exponentials (B^A), and the terminal object (1).

$A \to B$ in the category is represented by an exponential object B^A.

Example 3. As an example of how to work with product and exponential objects, we can build a morphism that swaps the inner components of a product, $((A \times B) \times C) \to (B \times A)$, similar to the λ-calculus term in Example 2. First, observe that we have the projection morphism π_1 on the following objects:

$$\pi_1 : ((A \times B) \times C) \to (A \times B)$$

and additionally, we may construct a morphism that swaps a product as follows:

$$\frac{\pi_2 : (A \times B) \to B \quad \pi_1 : (A \times B) \to A}{(\pi_2, \pi_1) : (A \times B) \to (B \times A)}$$

In order to push the swapping into the nested pair, we can compose the two morphisms together, giving us the following:

$$\frac{\pi_1 : ((A \times B) \times C) \to (A \times B) \quad (\pi_2, \pi_1) : (A \times B) \to (B \times A)}{(\pi_2, \pi_1) \circ \pi_1 : ((A \times B) \times C) \to (B \times A)}$$

We also need one more equation about products that is derivable from the definition given in Figure 4:

$$\begin{aligned}(f, g) \circ h &= (\pi_1 \circ (f, g) \circ h, \pi_2 \circ (f, g) \circ h) \\ &= (f \circ h, g \circ h)\end{aligned}$$

Notice that this equation guarantees that our original morphism, that discards C and then swaps A and B, is the same as one that builds a product by projecting twice into its input:

$$(\pi_2, \pi_1) \circ \pi_1 = (\pi_2 \circ \pi_1, \pi_1 \circ \pi_1)$$

In order to see how this morphism behaves when given input, we can compose it with another morphism that will build a nested product:

$$\frac{f : \Gamma \to A \quad g : \Gamma \to B \quad h : \Gamma \to C}{((f, g), h) : \Gamma \to (A \times B) \times C}$$

When put together, this morphism simplifies as follows:

$$\begin{aligned}(\pi_2, \pi_1) \circ \pi_1 \circ ((f, g), h) &= (\pi_2, \pi_1) \circ (f, g) \\ &= (\pi_2 \circ (f, g), \pi_1 \circ (f, g)) \\ &= (g, f)\end{aligned}$$

Finally, if we want to represent this morphism in the exponential object $(B \times A)^{(A \times B) \times C}$, we are in a bit of trouble since the input of our morphism is missing some extra Γ. Luckily, we can throw a redundant 1 into the input, since $1 \times A$ is isomorphic to A, as follows:

$$(\pi_2 \circ \pi_1, \pi_1 \circ \pi_1) \circ \pi_2 : 1 \times ((A \times B) \times C) \to B \times A$$

which gives us a morphism of the right shape for building the exponential object:

$$\mathsf{curry}((\pi_2 \circ \pi_1, \pi_1 \circ \pi_1) \circ \pi_2) : 1 \to (B \times A)^{(A \times B) \times C}$$

Therefore, the presence of the terminal object 1 in the category allows us to package any morphism $A \to B$ into a morphism that builds the corresponding exponential $1 \to B^A$. *End example* 3.

2.4 The Curry-Howard isomorphism

Amazingly, despite their different origins and presentations, each of these systems are all isomorphic to one another. Examples 1, 2, and 3 all correspond to various ways of expressing the same idea. Each of natural deduction, the λ-calculus, and cartesian closed categories end up revealing the same underlying concepts in different ways. The connection is perhaps most directly apparent between natural deduction and the λ-calculus. The propositions of natural deduction are isomorphic to the types of the λ-calculus, where conjunction is

the same as pair types, implication is the same as function types, and logical truth is the same as the unit type. Furthermore, the proofs of natural deduction are isomorphic to the (typed) terms of the λ-calculus. This structural similarity between the two systems gives us the slogan, "proofs as programs and propositions as types." From this point of view, natural deduction may be seen as the essence of the type system for the λ-calculus and the λ-calculus may be seen as a more concise term language for expressing proofs in natural deduction. For this reason, we may say that the λ-calculus is a natural deduction language.

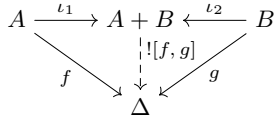
Likewise, cartesian closed categories are isomorphic to the other two systems. The product and exponential objects correspond to pair and function types in the λ-calculus, and to conjunction and implication in natural deduction. Furthermore, the terminal object corresponds to the unit type and to logical truth. Cartesian closed categories may be seen as variable-free presentation of the λ-calculus, where λ-abstractions are replaced by primitive functions, and as the internalization of hypothetical natural deduction. Intuitively, types and propositions correspond to objects in the category, so that conjunction, $A \wedge B$, pair types, $A * B$, are represented by products, $A \times B$, and implication, $A \supset B$, and function types, $A \to B$, are represented by exponentials, B^A, and similarly for truth, $\top$, unit type, 1, and the terminal object. Furthermore, the judgments $\Gamma \vdash A$ and $\Gamma \vdash M : A$ correspond to morphisms $f : \Gamma \to A$ in the cartesian closed category.

The correspondence between these three systems is not just an isomorphism between their static structures, but also extends to the dynamic properties as well. The notions of local soundness and completeness in natural deduction are exactly the same as β and η equivalence of terms in the λ-calculus, respectively, for each of pair, function, and unit types. Furthermore, the β rules in the λ-calculus correspond to the commutations of the respective diagrams in the cartesian closed category, and the η rules correspond to the uniqueness property for each construct. Therefore, it is no coincidence that the original β and η rules for functions in the λ-calculus appeared as they did, and in general this treatment of connectives extends to other programming language constructs.

3. A critical look at the λ-calculus

The Curry-Howard isomorphism lead to striking discoveries and developments that likely would not have arisen otherwise. The connection between logic and programming languages led to the development of mechanized proof assistants, notably the Coq [21] system, which are used in both the security and verification communities for validating the correctness of programs. The connection between category theory and programming languages suggested a new compilation technique for ML [22]. However, let us now look at λ-calculus with a more critical eye. There are some defining principles and computational phenomena that are important to programming languages, but are not addressed by the λ-calculus. For example, what about:

- *Duality?* The concept of duality is important in category where it comes for free as a consequence of the presentation. Since the morphisms in category theory have a direction, we may just "flip all the arrows." This action gives us a straightforward method to find the dual of any category or diagram. For example, consider the diagram for products, $A \times B$, in Figure 4. For free, we may obtain the dual structure of a *sum object* by just turning this diagram around:



Now, the two projections have become two injections, ι_1 and ι_2, into the sum object, and definition guarantees a unique way out of the sum for any two morphisms f and g from both of its possible components into a common result.

Duality also appears in logic as well, for example in the traditional De Morgan laws $\neg(A \wedge B) = (\neg A) \vee (\neg B)$. Predictably, the corresponding concept of a sum object (the dual of a product) in logic is disjunction (the dual of a conjunction). If we look at the common treatment of logical disjunction in natural deduction, the introduction rules for $A \vee B$ bear a resemblance the elimination rules for $A \wedge B$:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee I_2$$

However, the elimination rule for disjunction is quite different from the introduction for conjunction:

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee E$$

This dissimilarity comes from the asymmetry in natural deduction. We may have many hypothesis, but only a single consequence. We have a rich set of tools for working with the logical meaning of conclusions, but nothing for assumptions. It seems like a more symmetrical system of logic would be easier to methodically determine duality in the same way we can in category theory.

Likewise, this form of duality is not readily apparent in the λ-calculus. Since the λ-calculus is isomorphic to natural deduction, it shares the same biases and lack of symmetry. The emphasis of the language is entirely on the *production* of information: a λ-abstraction *produces* a function, a function application *produces* the result of the function, *etc*. For this reason, the relationship between a pair, (M, N), and case analysis for sum types in functional programming languages:

$$\mathbf{case}\, M\, \mathbf{of}\ \mathsf{inl}(x) \Rightarrow N_1 \,|\, \mathsf{inr}(y) \Rightarrow N_2$$

is not entirely obvious. Nor is it entirely obvious how the categorical definition of sums is meant to correspond with sum types in functional languages. In particular, what is the corresponding η rule for functional programming languages that captures the uniqueness condition of a sum object? For this reason, we would like to study a language which expresses duality "for free," and which corresponds to a more symmetrical system of logic.

- *Other evaluation strategies?* Reynolds [97] observed that while functional or applicative languages may be based on the λ-calculus, the true λ-calculus implies a lazy (call-by-name) evaluation order, whereas many languages are evaluated by a strict (call-by-value) order that first reduces arguments before performing a function call.

  To resolve this mismatch between the λ-calculus and strict programming languages, Plotkin [91] defined a call-by-value variant of the λ-calculus along with a *continuation-passing style* (CPS) transformation that embeds evaluation order into the program itself. Sabry and Felleisen [101] give a set of equations for reasoning about the call-by-value λ-calculus based on Fischer's call-by-value CPS transformation [46], and which is related to Moggi's computational λ-calculus [85]. The equations are later refined into a theory for call-by-value reduction by Sabry and Wadler [102]. More recently, there has been work on a theory for reasoning about call-by-need evaluation of the λ-calculus [6, 9, 81], which is the strategy commonly employed by Haskell implementations, and the development of the call-

by-push-value [79] framework which includes both call-by-value and call-by-name evaluation but not call-by-need.

What we would ultimately want is not just another calculus, but instead a framework in which these distinctions in evaluation strategy come out as special cases, and where the relationships between strategies is naturally expressed. Can we have a logical foundation for programming languages that is naturally strict, in the same way that the λ-calculus is naturally lazy? Is it possible to have a better indication on which rules are affected by the choice of an evaluation strategy?

- *Object-oriented programming?* The object-oriented paradigm has become a prominent concept in the mainstream programming landscape. Unfortunately, what is meant by an "object" in the object-oriented sense is fuzzy, since the exact details of "what is an object" depend on choices made by the programming language under consideration. One concept of objects that is universal across every programming language is the notion of *dynamic dispatch* that is used to select the behavior of a method call based on the value or type of an object, and is emphasized by Kay [73] in the form of *message passing* in the design of Smalltalk. Abadi and Cardelli [1] give a theoretical formulation for the many features of object-oriented languages, wherein dynamic dispatch plays a central role. Can we give an account of the essence of objects, and in particular messages and dispatch that is connected to logic and category theory in the same way as the λ-calculus? Even more, can this foundation for objects refer back to basic principles discovered independently in the field of logic?
- *Computational effects?* Most modern programming languages contain some notion of *computational effect*, such as mutable state, input and output to the file system, or exception handling. These types of effects step outside the simple call-and-return protocol for functions, and lie outside of the expressive power of the λ-calculus [37]. One effect that has been recently connected with logic is expressed by callcc-like control operators from Scheme [74], which corresponds with classical logic [7, 56]. However, the traditional study of these control operators is to add new primitives to the λ-calculus. We would rather understand these effects in a setting where they are naturally expressed as a consequence of the language, rather than added on as an afterthought.

  Furthermore, what about other common computational effects like mutable state and exceptions, or more advanced effects like delimited control [27, 28, 36, 39] or algebraic effects [16, 18, 92, 94]? In the case of delimited control, there has been a proliferation of different primitive operators, and the connection between all of them is not fully developed. Rather, we need a framework where delimited control comes out as a natural consequence, instead of inventing different operators for different needs. Can these other computational effects also be included in the correspondence, in the same way that classical logic corresponds to programming with callcc?

With the aim of answering each of these reasons, we will put the λ-calculus aside and we look to another logical framework instead of natural deduction. Most surprisingly, we do not have to look very far, since Genzten [47] introduced the sequent calculus along side natural deduction as an alternative system of formal logic. Gentzen developed sequent calculus in order to better understand the properties of natural deduction. Therefore, to answers these questions about programming, we will look for the computational interpretation of the sequent calculus and its corresponding programming language.

4. The sequent calculus

Gentzen [47] developed the sequent calculus as a tool to reason about the properties of natural deduction. As such, the sequent calculus is still a system for formalizing symbolic logic, but one that turns the structure of natural deduction on its head. The first change when moving to the sequent calculus is that we may generalize the form of judgments from their hypothetical form in natural deduction. In the sequent calculus, in addition to carrying multiple local hypotheses, we may also have multiple consequences:

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

pronounced "A_1, A_2, ..., and A_n entail B_1, B_2, ..., or B_m," which states that assuming each of $A_1, A_2, \ldots, A_n$ are true then at least *one* of $B_1, B_2, \ldots, B_m$ must be true. By convention, the consequences to the right of entailment are collectively referred to as Δ, and a judgment of the form $\Gamma \vdash \Delta$ is called a *sequent*. Note that, opposite from the fact that an empty collection of assumptions in $\vdash B$ means that B is unconditionally true, an empty collection of conclusions in $A \vdash$ means that A is contradictory.

The next difference in the sequent calculus is the style of the inference rules compared to the rules of natural deduction so that the connective under consideration always appears on the bottom of the rule. The introduction rules of natural deduction are represented similarly in the sequent calculus, where they are called *right rules* since they work on a proposition on the right of entailment. For example, the right rule for conjunction in the sequent calculus is given as follows:

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

where the only difference between $\wedge R$ and $\wedge I$ is that $\wedge R$ allows for additional conclusions Δ. In contrast, the elimination rules of natural deduction appear quite different in the sequent calculus, where they are called *left rules* since they work on a proposition to the left of entailment. Continuing the example, the left rules for conjunction in the sequent calculus may be given as:

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

which say that if we can prove Δ from the assumption A and Γ, then we can just as well prove Δ from the more informative assumption $A \wedge B$ and Γ (and similarly for B). This style of inference rules, where the connective under consideration always occurs at the bottom of the rule, gives the sequent calculus an entirely "bottom up" style of building proofs.

Example 4. To illustrate the "bottom up" style of the sequent calculus, let us reconsider the proof of the proposition $((A \wedge B) \wedge C) \supset (B \wedge A)$ from Example 1. As before, we may begin the proof similarly with the right implication rule, corresponding with implication introduction from natural deduction:

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

Next, we may continue as we did before with the right conjunction rule, splitting the proof into two parts:

$$\frac{\begin{array}{cc}\vdots & \vdots\\ (A \wedge B) \wedge C \vdash B & (A \wedge B) \wedge C \vdash A\end{array}}{\begin{array}{c}(A \wedge B) \wedge C \vdash B \wedge A\\ \hline \vdash (A \wedge B) \wedge C \supset B \wedge A\end{array} \supset R} \wedge R$$

However, at this point in the proof, we diverge from Example 1. This is because in natural deduction we would now switch from a

$$A, B, C \in Proposition ::= X \mid A \wedge B \mid A \vee B \mid A \supset B \mid \neg A$$
$$\Gamma \in Hypothesis ::= A_1, \ldots, A_n$$
$$\Delta \in Consequence ::= A_1, \ldots, A_n$$
$$Judgment ::= \Gamma \vdash \Delta$$

Axiom and cut:

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

Logical rules:

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

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

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

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

Figure 5. The sequent calculus with conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and negation ($\neg$).

"bottom up" to a "top down" use of the rules. Instead in the sequent calculus, we will continue with the "bottom up" application of the rules, and use a left conjunction rule in both sub-proofs to discard C from our assumption:

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

We may use another application of a left conjunction rule to select the correct proposition for both sub-proofs:

$$\frac{\dfrac{\dfrac{\dfrac{\vdots}{B \vdash B}\ \wedge L_2}{A \wedge B \vdash B}\ \wedge L_1}{(A \wedge B) \wedge C \vdash B}\ \wedge L_1 \quad \dfrac{\dfrac{\dfrac{\vdots}{A \vdash A}\ \wedge L_1}{A \wedge B \vdash A}\ \wedge L_1}{(A \wedge B) \wedge C \vdash A}\ \wedge L_1}{(A \wedge B) \wedge C \vdash B \wedge A}\ \wedge R}{\vdash (A \wedge B) \wedge C \supset B \wedge A}\ \supset R$$

And finally, we are now able to close off the both sub-proofs by using the Ax rule, which is similar to the corresponding Ax rule in natural deduction:

$$\frac{\dfrac{\dfrac{\dfrac{}{B \vdash B}\ Ax}{A \wedge B \vdash B}\ \wedge L_2}{(A \wedge B) \wedge C \vdash B}\ \wedge L_1 \quad \dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \wedge B \vdash A}\ \wedge L_1}{(A \wedge B) \wedge C \vdash A}\ \wedge L_1}{(A \wedge B) \wedge C \vdash B \wedge A}\ \wedge R}{\vdash (A \wedge B) \wedge C \supset B \wedge A}\ \supset R$$

End example 4.

The definitions and rules of the sequent calculus are summarized in Figure 5, where we consider connectives for logical conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and negation ($\neg$). In addition to the logical rules and the single axiom, we also have the Cut rule, which allows us to connect an assumption and a conclusion in two different proofs. The intuition behind Cut is that if we are able to prove either A or Δ from Γ, and we know how to use Γ and A to prove Δ alone, then either way we have a proof of just Δ. Notice that the rules for disjunction are a mirror image of the rules for conjunction. Also, as we saw with conjunction, the right rules for disjunction and implication in the sequent calculus are similar to the corresponding introduction rules in natural deduction, but the left rules are quite different from the elimination rules. In particular, the left rule for implication may seem a bit unintuitive as compared to its presentation in natural deduction.

Example 5. In order to convince ourselves that the left implication rule makes sense, consider how we can eliminate implication in natural deduction:

$$\frac{\dfrac{}{A, A \supset B \vdash A \supset B}\ Ax \quad \dfrac{}{A, A \supset B \vdash A}\ Ax}{A, A \supset B \vdash B}\ \supset E$$

In the sequent calculus, we may begin a proof of the same goal by using the $\supset L$ rule:

$$\frac{\dfrac{\vdots}{A \vdash A, B} \quad \dfrac{\vdots}{A, B \vdash B}}{A, A \supset B \vdash B}\ \supset L$$

when then allows us to directly conclude the first sub-proof by connecting the A we assumed with the conclusion of A produced by $\supset L$, and the second sub-proof by connecting the hypothesis B produced by the $\supset L$ with the conclusion from our goal:

$$\frac{\dfrac{}{A \vdash A, B}\ Ax \quad \dfrac{}{A, B \vdash B}\ Ax}{A, A \supset B \vdash B}\ \supset L$$

Intuitively, $A \supset B$ means that if A is true then B must be true. Therefore, if we independently know that A is true, then assuming

$A \supset B$ is just as good as assuming B, since we can always apply the hypothesis $A \supset B$ to our proof of A to recover B. *End example* 5.

Remark 3. Recall the structural rules *weakening* and *contraction* in natural deduction from Remark 1, which were derivable from the definition given in Figure 1. We should also check whether these rules still apply in the sequent calculus defined in Figure 5, even though they are not given as explicit rules. The single weakening rule from natural deduction is reflected as two rules in the sequent calculus — one rule for working on each side of the sequent:

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

These two weakening rules are an implicit global property of the system defined in Figure 5, since we may apply the Ax axiom with any number of extra assumptions and conclusions. On the other hand, the two corresponding contraction rules:

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

are not quite so simple to prove, since many of the logical rules make a destructive choice. For example, working from the bottom up, the $\wedge L_1$ rule takes an assumption $A \wedge B$ and breaks it down by choosing to use A while simultaneously discarding the knowledge that B is true. Similar problems occur for the right disjunction rules. However, contraction is still derivable from the Cut rule as follows (parenthesis have been added to highlight the active proposition being cut):

$$\frac{(\Gamma, A), A \vdash \Delta \quad \overline{(\Gamma, A) \vdash A, \Delta}\ Ax}{(\Gamma, A) \vdash \Delta}\ Cut$$

Therefore, the implicit presentation in Figure 5 still admits the explicit structural rules as sound reasoning steps, which may be expanded into the above derivation. *End remark* 3.

4.1 Consistency and cut elimination

The motivation behind the original development of the sequent calculus was to prove the *consistency* of natural deduction. When natural deduction first came about, there was no guarantee that it actually captured the notion of logical reasoning that it intended to. What if the rules of natural deduction are so strong that they allowed for a proof of every single proposition? This would make natural deduction proofs absolutely useless, as they would deduce both true and false propositions, alike. For example, if we can prove the judgment:

$$\begin{array}{c} \mathcal{D} \\ \vdots \\ \vdash X \end{array}$$

which says that an unknown proposition X is provable without using any assumptions, then we may substitute any proposition for X and use $\mathcal{D}$ as a generic proof of anything. If we try to show that such a proof is impossible by a "bottom up" induction on $\mathcal{D}$, then the elimination rules of natural deduction cause us trouble. The problem with the elimination rules is that, when read from the bottom up, they cause new propositions to appear out of thin air. The worst offender is the implication elimination rule, because who is to say that there isn't some very clever A such that we can fill in the following proof:

$$\frac{\begin{array}{c}\mathcal{D}\\ \vdots \\ \vdash A \supset X\end{array} \quad \begin{array}{c}\mathcal{E}\\ \vdots \\ \vdash A\end{array}}{\vdash X}\ \supset E$$

Instead of trying to deal with the problem of consistency in natural deduction, one trick is to translate every proof in natural deduction into a proof in the sequent calculus. For example, we can translate the problematic $\supset E$ rule into an alternate proof in the sequent as follows:

$$\frac{\begin{array}{c}\mathcal{D}\\ \vdots \\ \Gamma \vdash A \supset B\end{array} \quad \begin{array}{c}\mathcal{E}\\ \vdots \\ \Gamma \vdash A\end{array}}{\Gamma \vdash B}\ \supset E$$

$$\Downarrow$$

$$\frac{\begin{array}{c}\mathcal{D}'\\ \vdots \\ \Gamma \vdash A \supset B, B\end{array} \quad \frac{\begin{array}{c}\mathcal{E}'\\ \vdots \\ \Gamma \vdash A, B\end{array} \quad \overline{\Gamma, B \vdash B}\ Ax}{\Gamma, A \supset B \vdash B}\ \supset L}{\Gamma \vdash B}\ Cut$$

It follows that if we can prove that the sequent calculus is consistent, we know that natural deduction must also be consistent.

Observe that in all the logical rules of sequent calculus presented in Figure 5, every proposition above the line of inference appears somewhere below. For example, in the implication left rule, both the propositions A and B above the line come from the original proposition $A \supset B$ from below. We can say that the logical rules of the sequent calculus enjoy the *sub-formula property*, which states that every proposition (formula) in the premise appears somewhere (as a subformula) below the line. Therefore, if we are given a proof of $\vdash X$ in the sequent calculus:

$$\begin{array}{c} \mathcal{D} \\ \vdots \\ \vdash X \end{array}$$

then we know that the final rule used to deduce $\vdash X$ could not have been any of the logical rules, since X is an unknown proposition not containing any connectives and with no sub-formulas. In addition, the Ax rule also does not apply. The only rule in the sequent calculus that does not follow the subformula property is the cut rule, which allows us to invent an arbitrary proposition A out of thin air. For example, we need to make sure that there is no very clever proof:

$$\frac{\begin{array}{c}\mathcal{D}\\ \vdots \\ \vdash A, X\end{array} \quad \begin{array}{c}\mathcal{E}\\ \vdots \\ A \vdash X\end{array}}{\vdash X}\ Cut$$

Notice how translating the problematic implication elimination rule into the sequent calculus reveals a hidden use of Cut, and similarly for the other elimination rules. It follows that if we can show that the cut rule is superfluous, so that any proof in the sequent calculus can be rewritten into an equivalent proof that doesn't make use of Cut, then we can prove that there is no the sequent calculus derivation of $\vdash X$ and the system is consistent. Therefore, the so-called *cut elimination* theorem of the sequent calculus is a key ingredient to showing that both the sequent calculus and natural deduction are consistent.

Theorem 1 (Cut elimination)**.** *For all proofs $\mathcal{D}$ of a judgment $\Gamma \vdash \Delta$ in the sequent calculus, there exists an alternate proof $\mathcal{D}'$ of $\Gamma \vdash \Delta$ does not contain any use of the Cut rule.*

4.2 Logical duality in the sequent calculus

In contrast to natural deduction, observe the deep symmetries apparent in the sequent calculus, giving rise to a notion of *duality*. The form of judgment, $\Gamma \vdash \Delta$, allows for multiple conclusions as well as multiple assumptions. The logical rules for conjunction and

Duality of judgments:

$$(\Gamma \vdash \Delta)^\circ = \Delta^\circ \vdash \Gamma^\circ$$

Duality of propositions:

$$(A \wedge B)^\circ = (A^\circ) \vee (B^\circ) \qquad (A \vee B)^\circ = (A^\circ) \wedge (B^\circ)$$
$$(A \supset B)^\circ = (B^\circ) - (A^\circ) \qquad (B - A)^\circ = (A^\circ) \supset (B^\circ)$$
$$(\neg A)^\circ = \neg(A^\circ)$$

Figure 6. The duality of logic in the sequent calculus.

disjunction are exactly mirror images of one another across entailment: the $\wedge R$ rule is opposite to $\vee L$, and the $\wedge L_1$ and $\wedge L_2$ rules are opposite to $\vee R_1$ and $\vee R_2$. Additionally, the logical negation is the internal realization of this duality, as its left and right rules are the mirror images of one another. Logical duality in the sequent calculus captures a relationship between the connectives that follows the familiar De Morgan's laws about the distribution of negation over conjunction and disjunction:

$$\neg(A \wedge B) \approx (\neg A) \vee (\neg B)$$
$$\neg(A \vee B) \approx (\neg B) \wedge (\neg B)$$

But what about implication? It is missing a logical connective that serves as its dual counterpart. Recall that implication is logically equivalent to an encoding in terms of conjunction and negation:

$$A \supset B \approx (\neg A) \vee B$$

so that A implies B holds if and only if either B is true or A is false. Then we may compute the dual counterpart to implication using De Morgan's laws:

$$\begin{aligned}\neg(A \supset B) &\approx \neg(\neg A \vee B) \\ &\approx (\neg(\neg A)) \wedge (\neg B) \\ &\approx (\neg B) - (\neg A)\end{aligned}$$

That is to say, the opposite of the implication $A \supset B$ is the subtraction $B - A$, which states that B is true and A is false:

$$B - A = B \wedge (\neg A)$$

Now, how do we express inference rules for forming proofs using the relatively unfamiliar concept of subtraction? Luckily, we do not have to be overly clever, and the symmetry of the sequent calculus gives us a mechanical way of generating the rules: flip the rules of implication across the left and right sides of the sequent:

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

In order to check that these rules are correct, we can translate them in terms of the rules for conjunction and negation according to our original encoding: $B - A = B \wedge (\neg A)$. The right rule for subtraction is derived from the other right rules:

$$\dfrac{\Gamma \vdash B, \Delta \quad \dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \neg R}{\Gamma \vdash B \wedge \neg A, \Delta}\ \wedge R$$

and the left rule is derived with the help of contraction as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma, B \vdash A, \Delta}{\Gamma, \neg A, B \vdash \Delta}\ \neg L}{\Gamma, B \wedge \neg A, B \vdash \Delta}\ \wedge L_2}{\Gamma, B \wedge \neg A, B \wedge \neg A \vdash \Delta}\ \wedge L_1 \qquad \dfrac{\Gamma, B \wedge \neg A, B \wedge \neg A \vdash \Delta}{\Gamma, B \wedge \neg A \vdash \Delta}\ CL$$

Now that we have a connective for the dual to implication, we can express the duality of proofs in the sequent calculus — for every valid proof of a judgment:

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

there is a valid proof to the dual judgment:

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

The duality relation on propositions, is given in Figure 6. Note that the duality operation A° may be read as taking the negation of A, $\neg A$, and pushing the negation inward all the way using the De Morgan laws, until an unknown proposition variable X is reached.

Theorem 2 (Logical duality)**.** *If $\mathcal{D}$ is a sequent calculus proof of the judgment $\Gamma \vdash \Delta$, then there exists a proof $\mathcal{D}^\circ$ of the dual judgment $\Delta^\circ \vdash \Gamma^\circ$.*

Remark 4. The duality of proofs in the sequent calculus means that if a proposition A is unconditionally true, $\vdash A$, then that means that its dual must be a contradiction, $A^\circ \vdash$, and vice versa. For example, take a general proof of the basic contradiction $A \wedge (\neg A)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \wedge \neg A \vdash A}\ \wedge L_1}{A \wedge \neg A, \neg A \vdash}\ \neg L}{A \wedge \neg A, A \wedge \neg A \vdash}\ \wedge L_2}{A \wedge \neg A \vdash}\ CL$$

For free, duality gives us a general proof of the law of excluded middle, $A \vee (\neg A)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax}{A \vdash A \vee \neg A}\ \vee R_1}{\vdash \neg A, A \vee \neg A}\ \neg R}{\vdash A \vee \neg A, A \vee \neg A}\ \vee R_2}{\vdash A \vee \neg A}\ CR$$

This is not a trivial property — the fact that the sequent calculus can prove the law of excluded middle means that it is a proof system for *classical logic*. Natural deduction, as given in Figure 1 and extended with disjunction and negation, is unable to deduce a general proof of $A \vee (\neg A)$. For this reason, natural deduction is instead a proof system for *intuitionistic logic*, which cannot prove propositions like the law of excluded middle or double negation elimination, $\neg(\neg A)$. Notice that the sequent calculus proof of excluded middle made critical use of multiple conclusions to the right of entailment. The difference between a single conclusion and multiple conclusions is the difference between intuitionistic and classical logic. In fact, extending natural deduc-

tion with multiple conclusions turns it into a system of classical logic [7, 89]. *End remark* 4.

4.3 The sequent calculus as a language

Having explored the sequent calculus from the perspective of logic, what does it mean from the perspective of computation? Is there a programming language that corresponds with the sequent calculus, in the same sense that the λ-calculus corresponds to natural deduction? As it turns out, the computational interpretation of the sequent calculus reveals a lower-level language for describing programs, both corresponding to intuitionistic logic [61] like with the λ-calculus (see Remark 4) as well as classical logic [62, 63].

Similar to the way that the sequent calculus turns half the rules of natural deduction upside down, a programming language corresponding to the sequent calculus turns half the syntax of the λ-calculus inside out. For example, consider the syntax tree for a λ-calculus term that applies a function to three arguments:

$$(((\lambda x.M)\ M_1)\ M_2)\ M_3$$

shown in Figure 7 (a). Notice how the main interaction of the term, the *reducible expression* (redex) $(\lambda x.M)\ M_1$, is buried at the very bottom of the syntax tree. We may want to emphasize the redex by bringing it to the top of the syntax tree, so that it is immediately visible upon first inspection. Imagine grabbing the edge connecting λx and the first application to M_1 and lifting it upward, so that the rest of the tree hangs downward off of this edge. In effect, this action re-associates the syntax tree so that each function application in the *context* (where the hole surrounding the function $\lambda x.M$ is written $\Box$):

$$((\Box\ M_1)\ M_2)\ M_3$$

is inside out, giving us the half-inverted syntax tree in Figure 7 (b). Additionally, the overall context of the whole expression, labeled α, is buried to the bottom of the re-associated syntax tree, so that the path leading from the top of the syntax tree to the top application of M_3 becomes the spine hanging off to the right. The re-association of the syntax tree from natural deduction to sequent style fits within the general procedure for forming of the *zipper* [66, 83, 84] of a tree in functional programming languages, which takes a path into a tree and reifies it into a concrete structure in its own right.

This sequent style of writing programs, by inverting half of the program to bring out the relevant interactions, is captured by Curien and Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus [23]. The $\overline{\lambda}\mu\tilde{\mu}$-calculus emphasizes symmetries in computation that are not so obvious in the λ-calculus. In particular, computation in the $\overline{\lambda}\mu\tilde{\mu}$-calculus is represented as an interaction between a term, M, and its counterpart, a *co-term* K, which together form a *command*, $\langle M \| K \rangle$. In the more traditional setting of the λ-calculus, a co-term K may be understood as the representation of a context that specifies how to continue after it receives input, and the command $\langle M \| K \rangle$ may be understood as plugging the term M into the context represented by K, $K[M]$.

The most basic form of co-term in the $\overline{\lambda}\mu\tilde{\mu}$-calculus corresponds to a *let binding* common in functional programming languages. The term $\mathbf{let}\, x = M' \,\mathbf{in}\, M$ means to bind the variable x to the value returned by M' during evaluation of the sub-term M, and may be encoded in the λ-calculus using functions:

$$(\mathbf{let}\, x = M' \,\mathbf{in}\, M) = (\lambda x.M)\ M'$$

The $\overline{\lambda}\mu\tilde{\mu}$-calculus includes a syntactic representation of a let binding missing its input:

$$\mathbf{let}\, x = \Box \,\mathbf{in}\, K[M]$$

which may be written as the co-term, $\mathbf{in}\, x \Rightarrow \langle M \| K \rangle$, that is waiting for an *input*, a term named x, before running the underlying command $\langle M \| K \rangle$, giving us a form of *input abstraction*. The symmetrical of the sequent calculus also points out a term corresponding to the dual of an ordinary let binding. The dual to input abstraction, an *output abstraction* written $\mathbf{out}\, \alpha \Leftarrow \langle M \| K \rangle$, is waiting for an *output*, a co-term named α, before running the underlying command. In this term, α is a variable standing in for an unknown co-term, and is the dual to ordinary variables of the λ-calculus (a *co-variable*).

The subset of the $\overline{\lambda}\mu\tilde{\mu}$-calculus discussed so far, corresponding to Herbelin's [63] $\mu\tilde{\mu}$-calculus, gives a core language for describing input, output, and interactions. The grammar and typing rules for this core language is given in Figure 8. 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 5, we are currently trying to prove the proposition $A \wedge B$, so it is considered the active proposition of the judgment $\Gamma \vdash A \wedge B$. The concept of an active proposition corresponds to three different situations in a program:

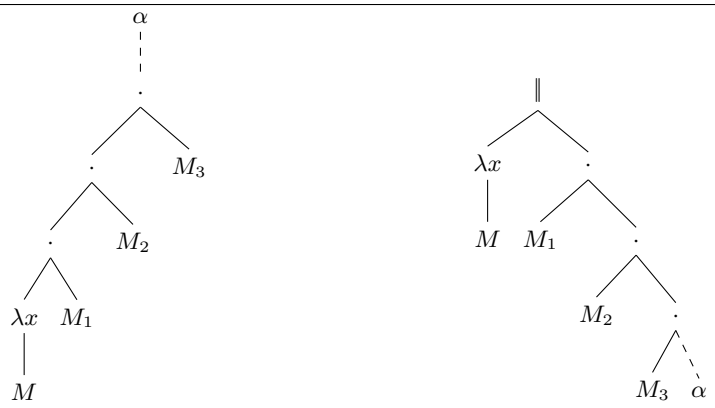
- Active on the right ($\Gamma \vdash M : A | \Delta$): we are building a term M that sends as its output information of type A (that is, M is a *producer* of type A).
- Active on the left ($\Gamma | K : A \vdash \Delta$): we are building a co-term K that receives as its input information of type A (that is, K is a *consumer* of type A).
- Passive ($S : (\Gamma \vdash \Delta)$): we are building a command S that has no distinguished input or output, but may only reference (input) variables Γ and (output) co-variables Δ available from its environment.

Also notice that this language does not include any particular connectives — it is purely a structural system for describing the introduction and use of parameters (variables and co-variables). The Var rule creates a term by just referring to a variable from the available environment, and similarly the $CoVar$ rule creates a co-term by referring to a co-variable. The Cut rule connects a term and co-term that are both waiting to send and receive information of the same type. Finally, the *activation* rules AR and AL pick a distinguished type from the environment to activate by creating an output or input abstraction, respectively. Intuitively, if x is a variable of type A referenced by a command S, then the input abstraction $\mathbf{in}\, x \Rightarrow S$ is a co-term that is waiting for an input of type A that it will internally name x. Dually, if α is a co-variable of type A referenced by S, then the output abstraction $\mathbf{out}\, \alpha \Leftarrow S$ is a term that is waiting for a place send information of type A that it will internally name α.

Remark 5. As it turns out, general output abstractions in the sequent calculus give rise to a manipulation over control flow that is equivalent to the callcc control operator from Scheme [74], or Felleisen's [40] $\mathcal{C}$ operator. Intuitively, the term $\mathbf{out}\, \alpha \Leftarrow \langle M \| K \rangle$ can be read as:

$$\mathbf{out}\, \alpha \Leftarrow \langle M \| K \rangle = \mathsf{callcc}(\lambda \alpha.K[M])$$

This phenomenon is a consequence of Griffin's [56] observation in 1990 that the under the Curry-Howard isomorphism, classical logic corresponds to control flow manipulation, along with the fact that the sequent calculus with multiple conclusions formalizes classical logic (see Remark 4). Under this interpretation, multiple conclusions in the sequent calculus correspond to the availability of multiple co-variables. Indeed, multiple conclusions may be seen as the logical essence for this "classical" form of control effects (so called for the connections to classical logic as well as the traditional control operator callcc), since extending natural deduction with multiple conclusions, as in Parigot's [89] $\lambda\mu$-calculus, also gives a



(a) Natural deduction (b) Sequent calculus

Figure 7. Re-association of the abstract syntax tree for function calls.

$$
\begin{aligned}
A, B, C \in \mathit{Type} &::= X \\
M \in \mathit{Term} &::= x \mid \mathbf{out}\,\alpha \Leftarrow S \\
K \in \mathit{CoTerm} &::= \alpha \mid \mathbf{in}\,x \Rightarrow S \\
S \in \mathit{Command} &::= \langle M \| K \rangle \\
\Gamma \in \mathit{Input} &::= x_1 : A_1, \ldots, x_n : A_n \\
\Delta \in \mathit{Output} &::= \alpha_1 : A_1, \ldots, \alpha_n : A_n \\
\mathit{Judgment} &::= (\Gamma \vdash M : A | \Delta) \mid (\Gamma | K : A \vdash \Delta) \mid S : (\Gamma \vdash \Delta)
\end{aligned}
$$

Axiom and cut:

$$\frac{}{\Gamma, x : A \vdash x : A | \Delta}\ \mathit{Var} \qquad \frac{}{\Gamma | \alpha : A \vdash \alpha : A, \Delta}\ \mathit{CoVar} \qquad \frac{\Gamma \vdash M : A | \Delta \quad \Gamma | K : A \vdash \Delta}{\langle M \| K \rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

Structural rules:

$$\frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mathbf{out}\,\alpha \Leftarrow S : A | \Delta}\ AR \qquad \frac{S : (\Gamma, x : A \vdash \Delta)}{\Gamma | \mathbf{in}\,x \Rightarrow S : A \vdash \Delta}\ AL$$

Figure 8. The types and grammar of the core language of the sequent calculus.

programming language with control effects equivalent to callcc [7]. Additionally, just like restricting the sequent calculus to a single conclusion makes it a system of intuitionistic logic, restricting the $\overline{\lambda}\mu\tilde{\mu}$-calculus to a single co-variable makes it a language for pure functional programming equivalent to the λ-calculus [61]. *End remark* 5.

The remainder of the $\overline{\lambda}\mu\tilde{\mu}$-calculus is a term and a co-term for representing functions of type $A \to B$ in the language, corresponding to the right and left rules for implication in the sequent calculus. The grammar and typing rules are given in Figure 9. Since the right rule for implication is the same as implication introduction in natural deduction, the term for creating a function is the same as in the λ-calculus. On the other hand, the different form of the left implication rule means that we need to use a different syntactic form to represent function application. In the $\overline{\lambda}\mu\tilde{\mu}$-calculus, function application is represented by the co-term, $M \cdot K$. Intuitively, the co-term built by the $\to L$ rule represents a *call-stack*. If M is a term of type A, and K is expecting a result of type B, then the call-stack $M \cdot K$ is waiting to receive a function of type A. As an example, given that M_1, M_2, and M_3 are all terms of type A_1, A_2, and A_3, and K is a co-term waiting for an input of type B, then the call-stack $M_1 \cdot (M_2 \cdot (M_3 \cdot K))$ is a co-term waiting to receive a function of type $A_1 \to (A_2 \to (A_3 \to B))$:

$$\dfrac{\mathcal{D}_1 \quad \dfrac{\mathcal{D}_2 \quad \dfrac{\mathcal{D}_3 \quad \mathcal{E}}{\Gamma \vdash M_3 : A_3 | \Delta \quad \Gamma | K : B \vdash \Delta}\to L}{\Gamma \vdash M_2 : A_1 | \Delta \quad \Gamma | M_3 \cdot K : A_3 \to B \vdash \Delta}\to L}{\Gamma \vdash M_1 : A_1 | \Delta \quad \Gamma | M_2 \cdot M_3 \cdot K : A_2 \to A_3 \to B \vdash \Delta}\to L$$
$$\Gamma | M_1 \cdot M_2 \cdot M_3 \cdot K : A_1 \to A_2 \to A_3 \to B \vdash \Delta$$

Wadler [111, 112] gives an alternate interpretation of the sequent calculus as a programming language, shown in Figure 10. Instead of focusing on functions as in the $\overline{\lambda}\mu\tilde{\mu}$-calculus, Wadler's dual sequent calculus instead considers the other connectives — conjunction, disjunction, and negation — as primary, and functions may be reduced to these basic building blocks. As was the case with functions in $\overline{\lambda}\mu\tilde{\mu}$, the terms of Wadler's sequent calculus are similar to terms in the λ-calculus, whereas elimination forms are expressed as a co-term. On the one hand, terms of pair type, $A \times B$, are introduced as a pair of two terms, (M_1, M_2). On the other hand, the first and second projections are considered two ways of building a co-term of pair type, $\mathsf{fst}[K]$ and $\mathsf{snd}[K]$. For example, if K is a co-term expecting an input of type A, then $\mathsf{fst}[K]$ takes an input of type $A \times B$, extracts the first component, and forwards

$$A, B, C \in Type ::= X \mid A \to B$$
$$M \in Term ::= x \mid \mathbf{out}\,\alpha \Leftarrow S \mid \lambda x.M$$
$$K \in CoTerm ::= \alpha \mid \mathbf{in}\,x \Rightarrow S \mid M \cdot K$$

Logical rules:

$$\frac{\Gamma, x : A \vdash M : B | \Delta}{\Gamma \vdash \lambda x.M : A \to B | \Delta} \to R \qquad \frac{\Gamma \vdash M : A | \Delta \quad \Gamma | K : B \vdash \Delta}{\Gamma | M \cdot K : A \to B \vdash \Delta} \to L$$

Figure 9. The types and grammar for the logical part of the $\overline{\lambda}\mu\tilde{\mu}$-calculus.

$$A, B, C \in Type ::= X \mid A \times B \mid A + B \mid \neg A$$
$$M \in Term ::= x \mid \mathbf{out}\,\alpha \Leftarrow S \mid (M_1, M_2) \mid \mathsf{inl}(M) \mid \mathsf{inr}(M) \mid \mathsf{not}(K)$$
$$K \in CoTerm ::= \alpha \mid \mathbf{in}\,x \Rightarrow S \mid \mathsf{fst}[K] \mid \mathsf{snd}[K] \mid [K_1, K_2] \mid \mathsf{not}[M]$$

Logical rules:

$$\frac{\Gamma \vdash M_1 : A | \Delta \quad \Gamma \vdash M_2 : B | \Delta}{\Gamma \vdash (M_1, M_2) : A \times B | \Delta} \times R \qquad \frac{\Gamma | K : A \vdash \Delta}{\Gamma | \mathsf{fst}[K] : A \times B \vdash \Delta} \times L_1 \qquad \frac{\Gamma | K : B \vdash \Delta}{\Gamma | \mathsf{snd}[K] : A \times B \vdash \Delta} \times L_2$$

$$\frac{\Gamma \vdash M : A | \Delta}{\Gamma \vdash \mathsf{inl}(M) : A + B | \Delta} + R_1 \qquad \frac{\Gamma \vdash M : B | \Delta}{\Gamma \vdash \mathsf{inr}(M) : A + B | \Delta} + R_2 \qquad \frac{\Gamma | K_1 : A \vdash \Delta \quad \Gamma | K_2 : B \vdash \Delta}{\Gamma | [K_1, K_2] : A + B \vdash \Delta} + L$$

$$\frac{\Gamma | K : A \vdash \Delta}{\Gamma \vdash \mathsf{not}(K) : \neg A | \Delta} \neg R \qquad \frac{\Gamma \vdash M : A | \Delta}{\Gamma\, \mathsf{not}[M] : \neg A \vdash \Delta} \neg L$$

Figure 10. The types and grammar for the logical part of Wadler's sequent calculus.

it along to K. The syntax for sum types is dual to the syntax for pairs. The terms of sum type are introduced in a similar manner to the λ-calculus, by injecting a term into the sum by tagging it as $\mathsf{inl}(M)$ or $\mathsf{inr}(M)$, whereas the co-terms of a sum type consist of a pair of co-terms $[K_1, K_2]$. Intuitively, given two co-terms $K_1 : A$ and $K_2 : B$, the co-term $[K_1, K_2]$ accepts an input of type $A + B$ by checking the tag and forwarding the value of type A or B along to either K_1 or K_2 as appropriate. Finally, the less familiar type corresponding to negation captures a form of *continuations*: a term of type $\neg A$ is actually a co-term expecting input and vice versa. The $\neg A$ type is a crucial part of the encoding of functions in the calculus.

5. The duality of evaluation

Having described the *static* properties, the syntax and types, for languages based on the sequent calculus, we still need to show their *dynamic* properties, to explain what it means to run a program. Programs of the λ-calculus are evaluated by repeated application of the β rules for functions (and pairs) which serve as the primary vehicle of computation, and correspond to the proof of local soundness for connectives in natural deduction. However, the sequent calculus does not include elimination rules in the manner of natural deduction, so how can we phrase evaluation? It turns out that the answer to "what is computation in the sequent calculus?" comes from the concept of cut elimination mentioned in Section 4.1.

The proof of cut can be divided into two parts: the *logical* steps and the *structural* steps. The logical steps of cut elimination consider the cases when we have cut together two proofs ending in the left and right rules for the same connective, and show how to rewrite the proof into a new one that does not mention that particular connective. For example, we may have a proof where the $\supset R$ and $\supset L$ rules for the same proposition are cut together:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash B, \Delta\end{array}}{\Gamma \vdash A \supset B, \Delta} \supset R \quad \dfrac{\begin{array}{c}\mathcal{E}_1\\ \vdots\\ \Gamma \vdash A, \Delta\end{array} \quad \begin{array}{c}\mathcal{E}_2\\ \vdots\\ \Gamma, B \vdash \Delta\end{array}}{\Gamma, A \supset B \vdash \Delta} \supset L}{\Gamma \vdash \Delta}\, Cut$$

and this proof can be rewritten without mention of implication by breaking apart the proposition $A \supset B$ as follows:

$$\dfrac{\begin{array}{c}\mathcal{E}_1\\ \vdots\\ \Gamma \vdash A, \Delta\end{array} \quad \dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ \Gamma, A \vdash B, \Delta\end{array} \quad \begin{array}{c}\mathcal{E}_2\\ \vdots\\ \Gamma, A, B \vdash \Delta\end{array}}{\Gamma \vdash A, \Delta}\, Cut}{\Gamma \vdash \Delta}\, Cut$$

This transformation on proofs can be expressed by the reduction of a command in the $\overline{\lambda}\mu\tilde{\mu}$-calculus as follows:

$$(\beta^{\to}) \qquad \langle \lambda x.M \| M' \cdot K \rangle \to \langle M' \| \mathbf{in}\, x \Rightarrow \langle M \| K \rangle \rangle$$

Intuitively, this reduction states the fact that evaluating a function $\lambda x.M$ with respect to the call $M' \cdot K$ is the same thing as binding the argument M' to x and then evaluating the body of the function M with respect to the calling context K.

The structural steps of cut elimination handles all the other cases where we do not have a left and right rule 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. In the $\overline{\lambda}\mu\tilde{\mu}$-calculus, a case where a structural step is needed is reflected by the presence of an input or

output abstraction. We can capture the essence of these structural steps in the $\overline{\lambda}\mu\tilde{\mu}$-calculus by means of *substitution*:

$$(\mu) \qquad \langle \mu\alpha.S \| K \rangle \to S\,\{K/\alpha\}$$
$$(\tilde{\mu}) \qquad \langle M \| \mathbf{in}\, x \Rightarrow S \rangle \to S\,\{M/x\}$$

The $\tilde{\mu}$ reduction substitutes the term M for the variable x introduced by an input abstraction, distributing it in the command S to the points where it is used. The μ reduction is the mirror image, which substitutes a co-term K for a co-variable α introduced by an output abstraction.

Unfortunately, reduction in this calculus is extremely *non-deterministic*, to the point where the execution of a program may take completely divergent paths. The non-determinism of reductions in the $\overline{\lambda}\mu\tilde{\mu}$-calculus corresponds to the fact that classical cut elimination in the sequent calculus is also non-deterministic. The phenomenon is expressed by the fundamental conflict between input and output abstractions, as shown by the two dual μ and $\tilde{\mu}$ reductions for performing substitution:

$$\begin{array}{ccc} & \langle \mathbf{out}\,\alpha \Leftarrow S \| \mathbf{in}\, x \Rightarrow S' \rangle & \\ \mu \swarrow & & \searrow \tilde{\mu} \\ S\,\{(\mathbf{in}\, x \Rightarrow S')/\alpha\} & & S'\,\{(\mathbf{out}\,\alpha \Leftarrow S)/x\} \end{array}$$

Both the term $\mathbf{out}\,\alpha \Leftarrow S$ and co-term $\mathbf{in}\, x \Rightarrow S'$ are fighting for control in the command, and either one may win. The non-deterministic outcome of this conflict is exemplified in the case that neither α nor x referenced in their respective commands:

$$S \leftarrow_{\mu} \langle \mathbf{out}\,\_ \Leftarrow S \| \mathbf{in}\,\_ \Rightarrow S' \rangle \to_{\tilde{\mu}} S'$$

showing that any program may produce arbitrary results, since the same starting point may step to two different completely arbitrary commands. This form of divergent reduction paths is called a *critical pair* and is evidence that the reduction system is not *confluent*, meaning that reductions can be applied in any order and still reach the same result. From the perspective of the semantics of programming languages, this type of non-determinism is undesirable since it makes it impossible to predict the behavior of a program during execution.

5.1 Confluence and evaluation strategy

In order to restore determinism to the calculus, Curien and Herblin observed that all that is needed is to give priority to one reduction over the other [23]:

> Call-by-value consists in giving priority to the (μ)-redexes (which serve to encode the terms, say, of the form $M\ M'$), while call-by-name gives priority to the $(\tilde{\mu})$-redexes.

This observation led them to splinter the $\overline{\lambda}\mu\tilde{\mu}$-calculus into two separate sub-languages — the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus and the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus — which embody call-by-name and call-by-value computation in the sequent calculus. The $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus follows a call-by-value evaluation strategy, and is so named since it corresponds to the LK^q fragment of Danos *et. al.*'s [26] system LK^{tq} for deterministic cut elimination in the sequent calculus. Likewise, the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus follows a call-by-name strategy and corresponds to the LK^t fragment of LK^{tq}.

The design of the $\overline{\lambda}\mu\tilde{\mu}_Q$ and $\overline{\lambda}\mu\tilde{\mu}_T$ sub-languages is guided by stability under call-by-value and call-by-name evaluation, respectively, based on the connection between natural deduction and the sequent calculus. For this reason, the call-by-value sub-language points out a subset of terms called *values*, denoted by the meta-variable V, which consists of variables and λ-abstractions as in Plotkin's original call-by-value λ-calculus [91]. This allows the $\tilde{\mu}$ rule to be restricted according to the call-by-value priority so that only values may be substituted for a variable:

$$(\tilde{\mu}_V) \qquad \langle V \| \mathbf{in}\, x \Rightarrow S \rangle \to S\,\{V/x\}$$

In addition, the co-term for function calls is restricted to the form $V \cdot K$, so that only values may be the argument to a function call. Similarly, the call-by-name sub-language points out a subset of co-terms called *co-values*, denoted by the meta-variable E, which correspond to evaluation contexts in the call-by-name λ-calculus. That way the μ rule may be restricted according to the call-by-name priority so that only co-values are substituted for a co-variable:

$$(\mu_E) \qquad \langle \mathbf{out}\,\alpha \Leftarrow S \| E \rangle \to S\,\{E/\alpha\}$$

The reduction rules for the call-by-value and call-by-name sub-language of the $\overline{\lambda}\mu\tilde{\mu}$-calculus are summarized in Figures 11 and 12.

Remark 6. We may regard the co-term for a function call, $M \cdot K$, as a syntactic representation of a single frame on a *call-stack*. The first component stores the argument to be used by the function, and the second component stores a representation of the return pointer which specifies what should happen once the function call has completed. For example, the co-term $M_1 \cdot M_2 \cdot M_3 \cdot K$ may be viewed as a call-stack for a function accepting three arguments, and which returns to K. This reading gives the $\overline{\lambda}\mu\tilde{\mu}$-calculus, and the sequent calculus in general, its low-level flavor. By representing a context by an explicit syntactic object K, we have a direct representation of a tail-recursive interpreter [12], which can also be seen as a form of abstract machine. In particular, we may view the syntax of the $\overline{\lambda}\mu\tilde{\mu}$-calculus as a higher-level representation of a form of CEK machine [38]: the control (C) is represented by a term M, the continuation (K) is represented by a co-term K, and the environment (E) is implicit and instead implemented as a static substitution operation. Finally, the configuration state of the machine is represented by a command S in the $\overline{\lambda}\mu\tilde{\mu}$-calculus.

Furthermore, the syntactic restrictions of the $\overline{\lambda}\mu\tilde{\mu}_Q$ and $\overline{\lambda}\mu\tilde{\mu}_T$ sub-languages correspond to restrictions that occur in call-by-name and call-by-value abstract machines for the λ-calculus, namely the Krivine machine [76] and a CEK-like machine [38], respectively. On the one hand, in a call-by-name abstract machine for the λ-calculus, only strict call-stacks which immediately perform a function call are formed during execution. On the other hand, only values are pushed on the call-stack in a call-by-value abstract machine, rather than unevaluated terms M. For example, the command:

$$\langle \lambda x.M \| M_1 \cdot M_2 \cdot M_3 \cdot \alpha \rangle$$

where M_1, M_2, M_3 are unevaluated terms, would not be legal in the call-by-value sub-language, since we would only push a computed value onto the call-stack. Additionally, the command $\langle \lambda x.M \| N_1 \cdot \mathbf{in}\,[\,\Rightarrow x]S \rangle$ would not be legal in the call-by-name sub-language, since K corresponds to the non-strict context $\mathbf{let}\, x = \square\, \mathbf{in}\, z$ — this context is not strict because it does not need the evaluation of the term plugged into $\square$ to return a result. *End remark* 6.

In contrast to the solution to non-determinism devised for the $\overline{\lambda}\mu\tilde{\mu}$-calculus, Wadler's dual sequent calculus takes a slightly different approach. In accordance with Curien and Herbelin, Wadler carves out a subset of terms called values and a subset of co-terms called co-values, and forms two deterministic restrictions of the reduction rules for the sequent calculus following a call-by-value and call-by-name evaluation order, respectively. The rules for substitution, μ and $\tilde{\mu}$, follow the same pattern in the call-by-value and call-by-name dual sequent calculus as in $\overline{\lambda}\mu\tilde{\mu}$. Additionally, the reduction rules for the logical constructions of the language take on additional restrictions as appropriate, similar to the syntactic restrictions in $\overline{\lambda}\mu\tilde{\mu}$. For example, the β rule for projecting the com-

$$
\begin{array}{rl}
S \in Command & ::= \langle M \| K \rangle \\
V \in Value & ::= x \mid \lambda x.M \\
M \in Term & ::= V \mid \mathbf{out}\,\alpha \Leftarrow S \\
K \in CoTerm & ::= \alpha \mid V \cdot K
\end{array}
$$

$$
\begin{array}{rl}
(\mu) & \langle \mathbf{out}\,\alpha \Leftarrow S \| K \rangle \to S\,\{K/\alpha\} \\
(\tilde{\mu}_V) & \langle V \| \mathbf{in}\,x \Rightarrow S \rangle \to S\,\{V/x\} \\
(\beta^{\to}) & \langle \lambda x.M \| V \cdot K \rangle \to \langle V \| \mathbf{in}\,x \Rightarrow \langle M \| K \rangle \rangle
\end{array}
$$

Figure 11. The call-by-value $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus.

$$
\begin{array}{rl}
S \in Command & ::= \langle M \| K \rangle \\
M \in Term & ::= V \mid \mathbf{out}\,\alpha \Leftarrow S \\
E \in CoValue & ::= \alpha \mid M \cdot E \\
K \in Coterm & ::= E \mid \mathbf{in}\,x \Rightarrow S
\end{array}
$$

$$
\begin{array}{rl}
(\mu_E) & \langle \mathbf{out}\,\alpha \Leftarrow S \| E \rangle \to S\,\{E/\alpha\} \\
(\tilde{\mu}) & \langle M \| \mathbf{in}\,x \Rightarrow S \rangle \to S\,\{M/x\} \\
(\beta^{\to}) & \langle \lambda x.M \| M' \cdot E \rangle \to \langle M' \| \mathbf{in}\,x \Rightarrow \langle M \| E \rangle \rangle
\end{array}
$$

Figure 12. The call-by-name $\overline{\lambda}\mu\tilde{\mu}_T$-calculus.

$$
\begin{array}{rl}
V \in Value & ::= \alpha \mid (V, V) \mid \mathsf{inl}(V) \mid \mathsf{inr}(V) \mid \mathsf{not}(K) \\
F \in TermContext & ::= (\Box, M) \mid (V, \Box) \mid \mathsf{inl}(\Box) \mid \mathsf{inr}(\Box)
\end{array}
$$

$$
\begin{array}{rll}
(\mu) & \langle \mathbf{out}\,\alpha \Leftarrow S \| K \rangle \to S\,\{K/\alpha\} & \\
(\tilde{\mu}_V) & \langle V \| \mathbf{in}\,x \Rightarrow S \rangle \to S\,\{V/x\} & \\
(\beta_V^{\times}) & \langle (V_1, V_2) \| \mathsf{fst}[K] \rangle \to \langle V_1 \| K \rangle & \langle (V_1, V_2) \| \mathsf{snd}[K] \rangle \to \langle V_2 \| K \rangle \\
(\beta_V^{+}) & \langle \mathsf{inl}(V) \| [K_1, K_2] \rangle \to \langle V \| K_1 \rangle & \langle \mathsf{inr}(V) \| [K_1, K_2] \rangle \to \langle V \| K_2 \rangle \\
(\beta^{\neg}) & \langle \mathsf{not}(K) \| \mathsf{not}[M] \rangle \to \langle M \| K \rangle & \\
(\mu_\eta) & M \to \mathbf{out}\,\alpha \Leftarrow \langle M \| \alpha \rangle & \\
(\tilde{\mu}_\eta) & K \to \mathbf{in}\,x \Rightarrow \langle x \| K \rangle & \\
(\varsigma) & F[M] \to \mathbf{out}\,\alpha \Leftarrow \langle M \| \mathbf{in}\,x \Rightarrow \langle F[x] \| \alpha \rangle \rangle &
\end{array}
$$

Figure 13. Wadler's call-by-value dual sequent calculus.

ponents of a pair must be restricted so that the pair is first a value:

$$\langle (V_1, V_2) \| \mathsf{fst}\ K \rangle \to_{\beta\times} \langle V_1 \| K \rangle$$
$$\langle (V_1, V_2) \| \mathsf{snd}\ K \rangle \to_{\beta\times} \langle V_2 \| K \rangle$$

Likewise, the call-by-value β reductions for the sum type can only match on the tag of a sum term when the underlying component is a value:

$$\langle \mathsf{inl}(V) \| [K_1, K_2] \rangle \to_{\beta+} \langle V \| K_1 \rangle$$
$$\langle \mathsf{inr}(V) \| [K_1, K_2] \rangle \to_{\beta+} \langle V \| K_2 \rangle$$

However, the syntax for these two deterministic reduction systems is not restricted — the same programs are acceptable in both the call-by-value and call-by-name systems. To compensate for this additional generality, additional rules must be added to push computation forward when we are facing a term that lies outside the syntactic restriction. For example, when we are trying to match on the term $\mathsf{inl}(\mathbf{out}\,\alpha \Leftarrow S)$, where $\mathbf{out}\,\alpha \Leftarrow S$ is not a value, then the β rule for sums is not strong enough to choose which branch to take:

$$\langle \mathsf{inl}(\mathbf{out}\,\alpha \Leftarrow S) \| [K_1, K_2] \rangle \not\to_{\beta+}$$

Similar issues arise when we attempt to project out of a non-value pair (M_1, M_2). Instead, Wadler's dual sequent calculus adds a new family of reductions ς whose purpose is to *lift* computations out of structures so that reduction may continue. In the above program,

$$E \in CoValue ::= \alpha \mid \mathsf{fst}[E] \mid \mathsf{snd}[E] \mid [E, E] \mid \mathsf{not}[M]$$
$$F \in CoTermContext ::= \mathsf{fst}[\Box] \mid \mathsf{snd}[\Box] \mid [\Box, K] \mid [E, \Box]$$

$$(\mu_E) \quad \langle \mathbf{out}\,\alpha \Leftarrow S \| E\rangle \to S\,\{E/\alpha\}$$
$$(\tilde{\mu}) \quad \langle M \| \mathbf{in}\,x \Rightarrow S\rangle \to S\,\{M/x\}$$
$$(\beta_E^{\times}) \quad \langle (M_1, M_2) \| \mathsf{fst}[E]\rangle \to \langle M_1 \| E\rangle \qquad \langle (M_1, M_2) \| \mathsf{snd}[E]\rangle \to \langle M_2 \| E\rangle$$
$$(\beta_E^{+}) \quad \langle \mathsf{inl}(M) \| [E_1, E_2]\rangle \to \langle V \| E_1\rangle \qquad \langle \mathsf{inr}(M) \| [E_1, E_2]\rangle \to \langle V \| E_2\rangle$$
$$(\beta^{\neg}) \quad \langle \mathsf{not}(K) \| \mathsf{not}[M]\rangle \to \langle M \| K\rangle$$
$$(\mu_\eta) \quad M \to \mathbf{out}\,\alpha \Leftarrow \langle M \| \alpha\rangle$$
$$(\tilde{\mu}_\eta) \quad K \to \mathbf{in}\,x \Rightarrow \langle x \| K\rangle$$
$$(\varsigma) \quad F[K] \to \mathbf{in}\,x \Rightarrow \langle \mathbf{out}\,\alpha \Leftarrow \langle x \| F[\alpha]\rangle \| K\rangle$$

Figure 14. Wadler's call-by-name dual sequent calculus.

we may instead take the step:

$$\langle \mathsf{inl}(\mathbf{out}\,\alpha \Leftarrow S) \| [K_1, K_2]\rangle \to_\varsigma \langle \mathbf{out}\,\alpha \Leftarrow S \| \mathbf{in}\,x \Rightarrow \langle \mathsf{inl}(x) \| [K_1, K_2]\rangle\rangle$$

where we have pulled M out of the sum tag inl by giving it a name. Now reduction may continue by letting the term take control over computation through the μ rule. The full set of rules for Wadler's call-by-value and call-by-name systems of reduction for the dual sequent calculus are given in Figures 13 and 14.

The rules for functions in Wadler's dual calculus are derived from an encoding of functions in terms of the other basic types for pairs, sums, and negation. There are two encoding of implications in terms of the other connectives:

$$A \to B = (\neg A) \vee B$$
$$A \to B = \neg(A \wedge (\neg B))$$

which are both logically equivalent in the setting of classical logic. These two logical encodings reveal two encodings of functions, one in terms of disjunction:

$$\lambda x.M = \mathbf{out}\,\gamma \Leftarrow \langle \mathsf{inl}(\mathsf{not}(\mathbf{in}\,x \Rightarrow \langle \mathsf{inr}(M) \| \gamma\rangle)) \| \gamma\rangle$$
$$M \cdot K = [\mathsf{not}[M], K]$$

and one in terms of conjunction:

$$\lambda x.M = \mathsf{not}(\mathbf{out}\,z \Leftarrow \langle z \| \mathsf{fst}[\mathbf{in}\,x \Rightarrow \langle z \| \mathsf{snd}[\mathsf{not}[M]]\rangle]\rangle)$$
$$M \cdot K = \mathsf{not}[(M, \mathsf{not}(K))]$$

The second encoding, in terms of conjunction, is used to define functions in the call-by-value calculus, since the encoding of a λ-abstraction is a value as intended. The first encoding, in terms of disjunction, is used to define functions in the call-by-name calculus.

5.2 Classical computation and the devil's choice

Example 6. Now that we have some reduction systems that tell us how to compute with classical logic, examine how a proof like the law of excluded middle from Remark 4 may be interpreted from the perspective of a program. To first gain some intuition, consider the following tale (with acknowledgments to Wadler [111], Selinger, and countless others who have made similar allegories):

> One day, the devil approached a man and presented him with a tempting offer: "I will give you one trillion dollars, or I will grant you any wish your heart desires upon payment of one trillion dollars. However, if accept the offer, I may choose which prize you receive."
> The man, as one may expect, was hesitant of making such a deal — what were the hidden costs? The devil assured him that the terms are as he said, all the man need to do is accept and he will be presented with a gift of the devil's choosing. Deciding that he has nothing to lose from either option, the man agrees, "I will take your offer, now which will you give me?"
> After some hesitation, the devil replies "I will grant you your wish, but only once you have paid me my sum."
> The man is disappointed by the response, how is he to afford such a high price? For a time, he went about his life as before, but over time, the open offer began to hang heavy over his head. How could he just ignore the unbounded potential of a free wish? From then on, he made it his life's goal to accrue the hefty sum required to receive his wish. The task took him many years, demanded all his time and energy, and pushed him to perform some questionable deeds along the way. However, as he became an old man, he finally saved up enough money to pay for his prize.
> At this time, the devil appeared before the now old man again, and the man beamed triumphantly, "I have your payment, devil, now give me my wish!" as he handed over a case full of money for his end of the bargain.
> The devil takes the money and looks it over in contemplation for a moment. Facing the man again, he says, "Did I say I would give you a wish? I meant that I would give you one trillion dollars," and with a sly smirk hands the very same case back to the old man.

We may phrase the devil's choice of gift as the type $riches + (riches \to wish)$, either the devil will choose to give riches (produce a value of type $riches$) or accept riches in order to produce an arbitrary wish (present a function for transforming values of type $riches$ to a result of type $wish$). Note that since the devil is the *giver*, so he is modeled as a term of the type, whereas the man is the *receiver*, so he is a co-term. A term that follows the behavior of the devil from the parable may be defined as:

$$Devil's\ Choice = \mathbf{out}\,\alpha \Leftarrow \langle \mathsf{inr}(\lambda x.\mathbf{out}\,\_ \Leftarrow \langle \mathsf{inl}(x) \| \alpha\rangle) \| \alpha\rangle$$

whose typing derivation is given in Figure 15 where the type $riches$ is denoted by A and $wish$ by B. Note that this term makes critical use of the fact that it may (1) name the co-term it is interacting with by introducing a co-variable, and (2) duplicating that co-variable.

We can exhibit the behavior of the devil's choice by considering different paths that the man may take, and show that the devil is never stumped by an obligation he cannot fulfill. For our purposes,

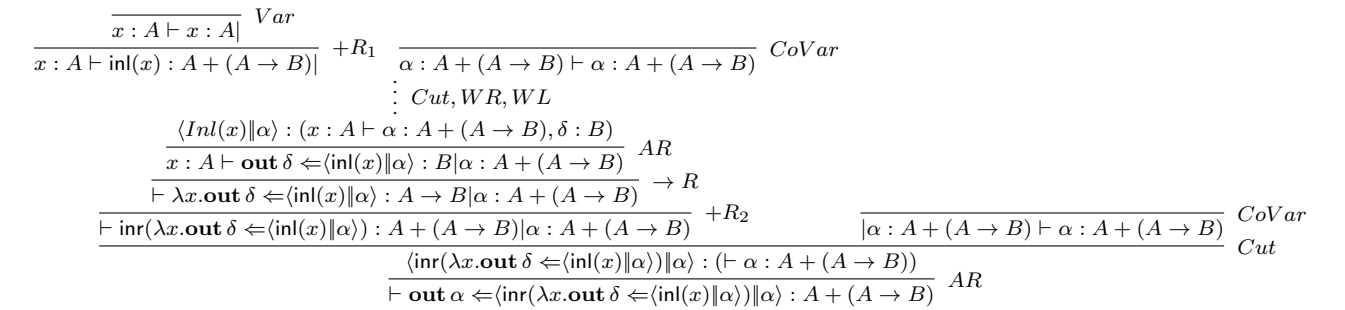


Figure 15. Typing derivation of the devil's choice.

we will focus on reduction in the call-by-value setting, although similar behavior holds in call-by-name as well. Since we know that the devil's first choice is to offer the wish for payment (present a value of the form $\mathsf{inr}(V)$), we focus on the mans reaction to this move. On the one hand, we can model what happens if the man were to ignore this offer with the co-term $[K_1, \mathbf{in}\_ \Rightarrow S_2]$, so that the second branch discards the given value:

$$\begin{aligned}
&\langle \mathbf{out}\,\alpha \Leftarrow \langle \mathsf{inr}(\lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| \alpha \rangle) \| \alpha \rangle \| [K_1, \mathbf{in}\,_ \Rightarrow S_2] \rangle \\
&\quad \to_{\mu} \langle \mathsf{inr}(\lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| [K_1, \mathbf{in}\,_ \Rightarrow S_2] \rangle) \| [K_1, \mathbf{in}\,_ \Rightarrow S_2] \rangle \\
&\quad \to_{\beta^+} \langle \lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| [K_1, \mathbf{in}\,_ \Rightarrow S_2] \rangle \| \mathbf{in}\,_ \Rightarrow S_2 \rangle \\
&\quad \to_{\tilde{\mu}_V} S_2
\end{aligned}$$

In this case, the man never made use of the devil's offer, so the devil can safely promise anything without being called out on it. On the other hand, we also have the course taken in the tale where the man manages to save up enough money (modeled by a value V_2 of type $riches$) and demands his wish (modeled by a co-value K_2 of type $wish$), which is modeled by the co-term $[K_1, V_2 \cdot K_2]$. The reduction of this command illustrates the devil's trick of changing his mind in order to use the man's own money to fulfill his obligation, which is shown by feeding the argument V_2 given in the second branch of the co-term to the first branch K_1:

$$\begin{aligned}
&\langle \mathbf{out}\,\alpha \Leftarrow \langle \mathsf{inr}(\lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| \alpha \rangle) \| \alpha \rangle \| [K_1, V_2 \cdot K_2] \rangle \\
&\quad \to_{\mu} \langle \mathsf{inr}(\lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| [K_1, V_2 \cdot K_2] \rangle) \| [K_1, V_2 \cdot K_2] \rangle \\
&\quad \to_{\beta^+} \langle \lambda x.\mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| [K_1, V_1 \cdot K_1] \rangle \| V_2 \cdot K_2 \rangle \\
&\quad \to_{\beta^\to} \langle V_2 \| \mathbf{in}\, x \Rightarrow \langle \mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(x) \| [K_1, V_2 \cdot K_2] \rangle \| K_2 \rangle \rangle \\
&\quad \to_{\tilde{\mu}_V} \langle \mathbf{out}\,_ \Leftarrow \langle \mathsf{inl}(V_2) \| [K_1, V_2 \cdot K_2] \rangle \| K_2 \rangle \\
&\quad \to_{\mu} \langle \mathsf{inl}(V_2) \| [K_1, V_2 \cdot K_2] \rangle \\
&\quad \to_{\mu} \langle V_2 \| K_1 \rangle
\end{aligned}$$

We may also illustrate this phenomenon in the syntax of natural deduction by using the callcc operator of Scheme. In a λ-calculus based language, we may rephrase the devil's choice by the term:

$$Devil's\ Choice = \mathsf{callcc}(\lambda k.\, \mathsf{inr}(\lambda x.\, \mathsf{throw}\, k\ \mathsf{inl}(x)))$$

Then the man's reaction to the devil is given by a case statement that analyzes the devil's choice. On the one hand, the course where the man ignores the offer is given by a case statement that ignores the input:

$$\begin{array}{l}
\mathbf{case}\ Devil's\ Choice\ \mathbf{of} \\
\quad \left| \begin{array}{ll}
\mathsf{inl}(x) \Rightarrow M_1 & \twoheadrightarrow M_2 \\
\mathsf{inr}(_) \Rightarrow M_2 &
\end{array} \right.
\end{array}$$

Since the function given in the second case is never used, we just take the second branch. On the other hand, the course where the man demands his wish is given by a case statement that calls the function in the second case with an argument:

$$\begin{array}{l}
\mathbf{case}\ Devil's\ Choice\ \mathbf{of} \\
\quad \left| \begin{array}{ll}
\mathsf{inl}(x) \Rightarrow M_1 & \twoheadrightarrow M_1\,\{V_2/x\} \\
\mathsf{inr}(f) \Rightarrow K_2[f\ V_2] &
\end{array} \right.
\end{array}$$

The devil pulls out his trick and unexpectedly re-routes the program, feeding the argument to f in the second case in order to fulfill the value expected by the first case. *End example* 6.

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

We saw how the symmetries of the sequent calculus present a logical duality that captures De Morgan duals in Section 4.2. This duality carries over the Curry-Howard isomorphism and presents itself as dualities in programming languages: (1) a duality between the *static* semantics (types) of the language, and (2) a duality between the *dynamic* semantics (reductions) of the language. These dualities of programming languages were first observed by Filinski [42] in 1989 from the correspondence with duality in category theory, which was later expanded upon by Selinger [105, 106] in the style of natural deduction. Curien and Herbelin [23] and Wadler [111, 112] bring this duality to the 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 corresponds directly from the logical duality of the sequent calculus. Since duality in the sequent calculus flips a judgment, so that assumptions are exchanged with conclusions, we also have a corresponding flip in the programming language. The dual of a term M of type A is a co-term K of the dual type and vice versa, and 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 ($+$) corresponds to conjunction ($\wedge$) and disjunction ($\vee$), we have

$$\begin{aligned}
(A \times B)^\circ &\triangleq (A^\circ) + (B^\circ) \\
(A + B)^\circ &\triangleq (A^\circ) \times (B^\circ)
\end{aligned}$$

The full duality relationship of types and programs in Wadler's dual sequent calculus is defined in Figure 16, giving us a syntactic duality between terms and types:

Theorem 3 (Type duality)**.** *In Wadler's dual sequent calculus:*

- $S : (\Gamma \vdash \Delta)$ *is a well-typed command if and only if* $S^\circ : (\Delta^\circ \vdash \Gamma^\circ)$ *is a well-typed command.*
- $\Gamma \vdash M : A | \Delta$ *is a well-typed term if and only if* $\Delta^\circ | K^\circ : A^\circ \vdash \Gamma^\circ$ *is a well-typed co-term.*
- $\Gamma | K : A \vdash \Delta$ *is a well-typed co-term if and only if* $\Delta^\circ \vdash K^\circ : A^\circ | \Gamma^\circ$ *is a well-typed term.*

Duality of judgments:

$$(S : (\Gamma \vdash \Delta))^\circ \triangleq S^\circ : (\Delta^\circ \vdash \Gamma^\circ)$$

$$(\Gamma \vdash M : A|\Delta)^\circ \triangleq \Delta^\circ | M^\circ : A^\circ \vdash \Gamma^\circ \qquad (\Gamma | K : A \vdash \Delta)^\circ \triangleq \Delta^\circ \vdash K^\circ : A^\circ | \Gamma^\circ$$

Duality of types:

$$(A \times B)^\circ \triangleq (A^\circ) + (B^\circ) \qquad (A + B)^\circ \triangleq (A^\circ) \times (B^\circ) \qquad (\neg A)^\circ \triangleq \neg(A^\circ)$$

Duality of programs:

$$\langle M \| K \rangle^\circ \triangleq \langle K^\circ \| M^\circ \rangle$$

$$(\mathbf{out}\,\alpha \Leftarrow S)^\circ \triangleq \mathbf{in}\,\alpha^\circ \Rightarrow (S^\circ) \qquad (\mathbf{in}\,x \Rightarrow S)^\circ \triangleq \mathbf{out}\,x^\circ \Leftarrow (S^\circ)$$

$$(M_1, M_2)^\circ \triangleq [M_1{}^\circ, M_2{}^\circ] \qquad [K_1, K_2]^\circ \triangleq (K_1{}^\circ, K_2{}^\circ)$$

$$(\mathsf{inl}(M))^\circ \triangleq \mathsf{fst}[M^\circ] \qquad (\mathsf{fst}[K])^\circ \triangleq \mathsf{inl}(K^\circ)$$

$$(\mathsf{inr}(M))^\circ \triangleq \mathsf{snd}[M^\circ] \qquad (\mathsf{snd}[K])^\circ \triangleq \mathsf{inr}(K^\circ)$$

$$(\mathsf{not}(K))^\circ \triangleq \mathsf{not}[K^\circ] \qquad (\mathsf{not}[M])^\circ \triangleq \mathsf{not}(M^\circ)$$

Figure 16. The duality relation of Wadler's sequent calculi.

Also of note is the fact that the duality operation is involutive: the dual of the dual is exactly the same as the original.

Theorem 4 (Involution)**.** *For all commands S, terms M, and co-terms K in Wadler's dual sequent calculus, $S^{\circ\circ} \triangleq S$, $M^{\circ\circ} \triangleq M$, and $K^{\circ\circ} \triangleq K$.*

The dynamic aspect of duality takes form as a relationship between the two reduction systems for evaluating programs in the sequent calculus: call-by-value reduction is dual to call-by-name reduction. That is, if we have a command S that behaves a certain way according to the call-by-value calculus, then the dual command S° behaves the same in a correspondingly dual way according to the call-by-name calculus. The two systems for reduction mirror each other exactly rule for rule.

Theorem 5 (Operational duality)**.**
- *If $S \to S'$ by Wadler's call-by-value sequent calculus, then $S^\circ \to S'^\circ$ by Wadler's call-by-name sequent calculus (and analogously for terms and co-terms).*
- *If $S \to S'$ by Wadler's call-by-name sequent calculus, then $S^\circ \to S'^\circ$ by Wadler's call-by-value sequent calculus (and analogously for terms and co-terms).*

A similar situation holds in the $\overline{\lambda}\mu\tilde{\mu}$-calculus. Just like how we needed to add subtraction as the logical counterpart to implication in Section 4.2, here we need to add the dual form of functions to complete the duality of the $\overline{\lambda}\mu\tilde{\mu}$-calculus. By analogy, the dual of functions is also referred to as subtraction, and it represents a transformation on co-terms, as the counterpart to a transformation on terms. The type rules for subtraction are the same as the rules for subtraction, and the syntax is reversed from functions:

$$\frac{\Gamma | K : B \vdash \alpha : A, \Delta}{\Gamma | \tilde{\lambda}\alpha.K : B - A \vdash \Delta} -R$$

$$\frac{\Gamma \vdash M : B | \Delta \quad \Gamma | K : A \vdash \Delta}{\Gamma \vdash M - K : B - A | \Delta} -L$$

Similarly, the β rule for subtraction is a mirror image of the rule for functions:

$$(\beta^-) \qquad \langle M - K' \| \tilde{\lambda}\alpha.K \rangle \to \langle \mathbf{out}\,\alpha \Leftarrow \langle M \| K \rangle \| K' \rangle$$

Finally, we extend the set of values in the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus so that $V - K$ is a value, and extend the set of co-values in the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus so that $\tilde{\lambda}\alpha.K$ is a co-value. With the addition of subtraction, duality of the $\overline{\lambda}\mu\tilde{\mu}$-calculus is defined in Figure 17. Furthermore, we get similar static and dynamic properties of this duality relation: duality preserves typing, duality is involutive, and the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus (call-by-value) is dual to the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus (call-by-name).

Theorem 6 (Type duality)**.** *In the $\overline{\lambda}\mu\tilde{\mu}$-calculus:*
- *$S : (\Gamma \vdash \Delta)$ is a well-typed command if and only if $S^\circ : (\Delta^\circ \vdash \Gamma^\circ)$ is a well-typed command.*
- *$\Gamma \vdash M : A | \Delta$ is a well-typed term if and only if $\Delta^\circ | K^\circ : A^\circ \vdash \Gamma^\circ$ is a well-typed co-term.*
- *$\Gamma | K : A \vdash \Delta$ is a well-typed co-term if and only if $\Delta^\circ \vdash K^\circ : A^\circ | \Gamma^\circ$ is a well-typed term.*

Furthermore, if a command, term, or co-term lies in the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus, its dual lies in the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus and vice versa.

Theorem 7 (Involution)**.** *For all commands S, terms M, and co-terms K in the $\overline{\lambda}\mu\tilde{\mu}$-calculus, $S^{\circ\circ} \triangleq S$, $M^{\circ\circ} \triangleq M$, and $K^{\circ\circ} \triangleq K$.*

Theorem 8 (Operational duality)**.**
- *If $S \to S'$ by the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus then $S^\circ \to S'^\circ$ by the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus (and analogously for terms and co-terms).*
- *If $S \to S'$ by the $\overline{\lambda}\mu\tilde{\mu}_T$-calculus then $S^\circ \to S'^\circ$ by the $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus (and analogously for terms and co-terms).*

6. The unity of duality

More recently, insights developed in the realm of logic have revealed a deeper connection in the dual roles between the two sides of the sequent calculus. The inspiration for this connection can be tracked back to Dummett's [33] 1976 lectures on the *justification* of logical principles. In essence, Dummett suggested that there are effectively two ways for determining whether logical laws are meaningful, which reveals a certain bias in the logician: the *verificationist* and the *pragmatist*.

In the eyes of a verificationist, it is the introduction rules (corresponding to the right rules in the sequent calculus), that give meaning to a logical connective. These are the primitive rules that define its character. All other rules (the elimination or left rules) must then be justified by means of the forms of introduction. In other words, the meaning of a proposition can be devised from its *canonical*

Duality of judgments:

$$(S : (\Gamma \vdash \Delta))^\circ \triangleq S^\circ : (\Delta^\circ \vdash \Gamma^\circ)$$

$$(\Gamma \vdash M : A|\Delta)^\circ \triangleq \Delta^\circ|M^\circ : A^\circ \vdash \Gamma^\circ \qquad (\Gamma|K : A \vdash \Delta)^\circ \triangleq \Delta^\circ \vdash K^\circ : A^\circ|\Gamma^\circ$$

Duality of types:

$$(A \to B)^\circ \triangleq (B^\circ) - (A^\circ) \qquad (B - A)^\circ \triangleq (A^\circ) \to (B^\circ)$$

Duality of programs:

$$\langle M \| K \rangle^\circ \triangleq \langle K^\circ \| M^\circ \rangle$$

$$(\mathbf{out}\,\alpha \Leftarrow S)^\circ \triangleq \mathbf{in}\,\alpha^\circ \Rightarrow (S^\circ) \qquad (\mathbf{in}\,x \Rightarrow S)^\circ \triangleq \mathbf{out}\,x^\circ \Leftarrow (S^\circ)$$

$$(\lambda x.M)^\circ \triangleq \tilde{\lambda} x^\circ.(M^\circ) \qquad (\tilde{\lambda}\alpha.K)^\circ \triangleq \lambda \alpha^\circ.(M^\circ)$$

$$(M \cdot K)^\circ \triangleq K^\circ - M^\circ \qquad (M - K)^\circ \triangleq K^\circ \cdot M^\circ$$

Figure 17. The duality relation of the $\overline{\lambda}\mu\tilde{\mu}$-calculus (extended with subtraction).

proofs [93], composed of introduction rules, and the other rules are sound with respect to them. This is a very similar property to local soundness for natural deduction described in Section 2.1, albeit a deeper one. In this case, given any canonical proof of $\Gamma \vdash A \wedge B$, we know that it must end with the $\wedge I$ rule since it is composed of introduction rules. Therefore, either elimination rule applied to a canonical proof of $\Gamma \vdash A \wedge B$ is subject to the appropriate local soundness reduction, which results in a canonical proof of either A or B.

In the eyes of a pragmatist, it is the elimination rules (corresponding to the left rules in the sequent calculus), that give meaning to a logical connective. That is to say, the primitive concept is what can possibly be done with such an assumption. This stance is the polar opposite of the verificationist. For a pragmatist, canonical proofs are composed of eliminations and the other rules are shown sound with respect to the uses of assumptions rather than the verification of facts. Dummett originally called for "harmony," that the two perspectives aught to result in the same meaning. However, we will see that in the sequent calculus, these two different approaches to viewing connectives gives a symmetric framework for viewing important concepts that arise in both proof search and in programming languages. In particular, we end up with two different styles of organizing programs (data construction vs. message passing), leading to two different forms of programs for product and sum types.

6.1 Polarization and focalization

Modern proof search employs several techniques to improve the efficiency of searching algorithms. Of particular interest are the searching techniques that come from the proof theoretic concepts of *focalization* and *polarization*, developed by Andreoli [4], Girard [53, 54], and Laurent [78]. The nature of these concepts are most apparent in the way they arise in Girard's linear logic [51]. Therefore, let us take a brief look at linear logic and its basic concepts before considering the impact on proof search.

Girard's linear logic [51] can be seen as a logic of resources. Recall in Remarks 1 and 3 that we admitted additional structural rules that gave us some flexibility over the use and tracking of assumptions (and conclusions). In particular, we assumed *weakening* principles that allowed for hypotheses and consequences to go unused in a proof, as well as *contraction* that allowed for their duplication. Linear logic disregards these allowances, instead treating assumptions as resources that *must* all be used and conclusions as obligations that *must* all be met.[5] In terms of our implicit treatment of the structural rules in the sequent calculus from Figure 5, this restriction corresponds to a restriction of the basic Ax axiom:

$$\frac{}{A \vdash A}\ Ax$$

so that we are only allowed to conclude a proof when we have *exactly* the right assumption to draw our desired conclusion, with no waste on either side.

Due to the careful use of resources in a proof, linear logic reveals different presentations of the usual connectives that are no longer equivalent. For example, the rules for conjunction given in Figure 5 are used to define the & connective of linear logic (pronounced "with"):

$$\frac{\Gamma \vdash A_1, \Delta \quad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A \,\&\, A_1, \Delta}\ \&R$$

$$\frac{\Gamma, A_1 \vdash \Delta}{\Gamma, A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma, A_2 \vdash \Delta}{\Gamma, A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1$$

However, linear logic also presents a different set of rules for describing conjunction, giving rise to the $\otimes$ connective (pronounced "times" or "tensor"):

$$\frac{\Gamma_1 \vdash A_1, \Delta_1 \quad \Gamma_2 \vdash A_2, \Delta_2}{\Gamma_1, \Gamma_2 \vdash A_1 \otimes A_2, \Delta_1, \Delta_2}\ \otimes R$$

$$\frac{\Gamma, A_1, A_2 \vdash \Delta}{\Gamma, A_1 \otimes A_2 \vdash \Delta}\ \otimes L$$

which makes essential use of our interpretation that *all* of the hypotheses must be true, effectively forcing a comma to the left of entailment to mean "and."

Dually, we also have two presentations of disjunction in linear logic. The traditional presentation of disjunction in the sequent calculus is distinguished by the $\oplus$ connective (pronounced "plus"):

$$\frac{\Gamma \vdash A_1, \Delta}{\Gamma \vdash A_1 \oplus A_2, \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \oplus A_2, \Delta}\ \oplus R_2$$

$$\frac{\Gamma, A_1 \vdash \Delta \quad \Gamma, A_2 \vdash \Delta}{\Gamma, A_1 \oplus A_2 \vdash \Delta}\ \oplus L$$

[5] It is typical for systems of linear logic to also include *exponential* connectives, usually called "of course!" ($!A$) and "why not?" ($?A$), that explicitly permit and track the erasure or duplication of propositions in a proof. However, for our purposes, we will not be considering such connectives.

$$A, B, C \in Proposition ::= X \mid A \oplus B \mid A \otimes B \mid 0 \mid 1 \mid A \,\&\, B \mid A \parr B \mid \bot \mid \top$$
$$\Gamma \in Hypothesis ::= A_1, \ldots, A_n$$
$$\Delta \in Consequence ::= A_1, \ldots, A_n$$
$$Judgment ::= \Gamma \vdash \Delta$$

Axiom and cut:

$$\frac{}{A \vdash A}\ Ax \qquad \frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, A \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ Cut$$

Logical rules:

Positive:

$$\frac{\Gamma \vdash A_1, \Delta}{\Gamma \vdash A_1 \oplus A_2, \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \oplus A_2, \Delta}\ \oplus R_2$$
$$\frac{\Gamma, A_1 \vdash \Delta \quad \Gamma, A_2 \vdash \Delta}{\Gamma, A_1 \oplus A_2 \vdash \Delta}\ \oplus L$$

no $0R$ rule $\qquad \frac{}{\Gamma, 0 \vdash \Delta}\ 0L$

$$\frac{\Gamma_1 \vdash A_1, \Delta_1 \quad \Gamma_2 \vdash A_2, \Delta_2}{\Gamma_1, \Gamma_2 \vdash A_1 \otimes A_2, \Delta_1, \Delta_2}\ \otimes R$$
$$\frac{\Gamma, A_1, A_2 \vdash \Delta}{\Gamma, A_1 \otimes A_2 \vdash \Delta}\ \otimes L$$

$$\frac{}{\vdash 1}\ 1R \qquad \frac{\Gamma \vdash \Delta}{\Gamma, 1 \vdash \Delta}\ 1L$$

Negative:

$$\frac{\Gamma, A_1 \vdash \Delta}{\Gamma, A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma, A_1 \vdash \Delta}{\Gamma, A_1 \,\&\, A_2 \vdash \Delta}\ \&L_2$$
$$\frac{\Gamma \vdash A_1, \Delta \quad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \,\&\, A_2, \Delta}\ \&R$$

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

$$\frac{\Gamma_1, A_1 \vdash \Delta_1 \quad \Gamma_2, A_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A_1 \parr A_2 \vdash \Delta_1, \Delta_2}\ \parr L$$
$$\frac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash A_1 \parr A_2, \Delta}\ \parr R$$

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

Figure 18. Girard's linear logic with two forms of conjunction ($\otimes$, &), disjunction ($\oplus$, $\parr$), truth (1, $\top$), and falsehood (0, $\bot$).

Whereas we have an alternate definition of disjunction called $\parr$ (pronounced "par"):

$$\frac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash A_1 \parr A_2, \Delta}\ \parr R$$
$$\frac{\Gamma_1, A_1 \vdash \Delta_1 \quad \Gamma_2, A_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, A_1 \parr A_2 \vdash \Delta_1, \Delta_2}\ \parr L$$

which makes essential use of our interpretation that only *one* of the consequences must be true, effectively meaning that a comma to the right of entailment means "or." Furthermore, we have a similar split in the connectives that represent truth or falsehood in the sequent calculus, giving us two representations of "true" — $\top$ and 1 — and two representations of "false" — 0 and $\bot$ — which are the units (the degenerate nullary versions) of the connectives &, $\otimes$, $\oplus$, and $\parr$ respectively. This system of linear logic is summarized in Figure 18

Example 7. It may be illustrative to show how the different formulations of conjunction and disjunction differ from one another. As an example, we cannot prove one conjunction from the other, and vice versa. On the one hand, consider the following attempt to prove $A \,\&\, B$ from $A \otimes B$:

$$\frac{\dfrac{A, B \vdash A \quad A, B \vdash B}{A, B \vdash A \,\&\, B}\ \&R}{A \otimes B \vdash A \,\&\, B}\ \otimes L$$

We cannot conclude this proof because the resources we received from $A \otimes B$ were both duplicated by the $\&R$ rule, leaving us with "left over" assumptions and preventing us from ending the proof. The fact that we get both A and B as resources is why the left rule for $\otimes$ splits assumptions and conclusions, so that we may prove $A \otimes B$ from itself:

$$\frac{\dfrac{\dfrac{}{A \vdash A}\ Ax \quad \dfrac{}{B \vdash B}\ Ax}{A, B \vdash A \otimes B}\ \otimes R}{A \otimes B \vdash A \otimes B}\ \otimes L$$

Note that the order of the left and right rules for $\otimes$ matter. If we had instead applied the right rule first (so that it appears as the bottom rule of the proof):

$$\frac{\dfrac{A, B \vdash A}{A \otimes B \vdash A}\ \otimes L \quad \vdash B}{A \otimes B \vdash A \otimes B}\ \otimes R$$

then we would be stuck in an unprovable state, since we would have had to move the entire assumption $A \otimes B$ to one branch of the proof or the other.

On the other hand, consider the following attempts to prove $A \otimes B$ from $A \,\&\, B$:

$$\dfrac{\dfrac{\overline{A \vdash A}\ Ax}{A \,\&\, B \vdash A}\ \&L_1 \qquad \vdash B}{A \,\&\, B \vdash A \otimes B}\ \otimes R$$

$$\dfrac{\dfrac{\vdash A \quad \overline{B \vdash B}\ Ax}{B \vdash A \otimes B}\ \otimes R}{A \,\&\, B \vdash A \otimes B}\ \&L_2$$

Even though the assumption $A \,\&\, B$ grants us allowance to assume A and to assume B, we must choose one of the two to use, leaving the other end dangling. The fact that choosing which assumption we would like to use eliminates the other option is why the right rule for $\&$ duplicates assumptions and conclusions, so that we may prove $A \,\&\, B$ from itself:

$$\dfrac{\dfrac{\overline{A \vdash A}\ Ax}{A \,\&\, B \vdash A}\ \&L_1 \qquad \dfrac{\overline{B \vdash B}\ Ax}{A \,\&\, B \vdash B}\ \&L_2}{A \,\&\, B \vdash A \,\&\, B}\ \&R$$

Note again that the order of the left and right rules for $\&$ matter. If we had instead applied one of the left rules first (so that we choose $\&L_1$ or $\&L_2$ at the bottom of the proof):

$$\dfrac{\dfrac{\overline{A \vdash A}\ Ax \qquad A \vdash B}{A \vdash A \,\&\, B}\ \&R}{A \,\&\, B \vdash A \,\&\, B}\ \&L_1$$

then we would be stuck in an unprovable state, since we would have already committed to use of either assumption A or assumption B throughout the remainder of the proof. *End example* 7.

Performing proof search is like an exercise where we are given a judgment, $\Gamma \vdash \Delta$, as a *goal*, and our task is to either: (1) show that the judgment is provable by building a completed proof tree starting with $\Gamma \vdash \Delta$ at the bottom and working our way up to the leaves by successive application of inference rules, or (2) show that the judgment is unprovable by demonstrating that no such proof tree can be built. When we search for a proof in this way, it is very easy to waste a lot of effort by pushing the proof tree into an unprovable state and then spending a lot of time working on it before realizing that we are stuck. It is in our best interest to minimize work done on a futile partial proof during the search procedure, and the choices we make can have a huge impact on this. For example, suppose we are trying to prove the judgment $\Gamma \vdash A \otimes B, \Delta$, where Γ and Δ are very large collections of very complicated propositions. As a first move, we might begin by trying the rules:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash A, \Delta & \vdash B\end{array}}{\Gamma \vdash A \otimes B, \Delta}\ \otimes R$$

and spend a very long time working out a proof of $\Gamma \vdash A, \Delta$ before we look at the second branch and realize that we don't have enough assumptions to finish a proof of $\vdash B$, meaning that all of our effort was for naught. These kinds of situations are what the concept of *focalization* aims to minimize in the practice of proof search.

The main idea behind focalization is a recognition that some inference rules are "safe" from the perspective of proof search, in that they will never turn a partially completed proof from provable to unprovable, whereas other rules are "dangerous", in that they have the potential to create a dead end that creates wasted effort. For example, we already saw that the $\otimes R$ rule is "dangerous," it began an unprovable proof out of a potentially provable judgment. On the other hand, we can consider the $\otimes L$ "safe" because this rule is *reversible*, meaning that the premise follows from the deduction:

$$\dfrac{\Gamma, A \otimes B \vdash \Delta}{\Gamma, A, B \vdash \Delta}$$

Therefore, if the deduction of the rule, $\Gamma, A \otimes B \vdash \Delta$, is a provable goal, then the premise, $\Gamma, A, B \vdash \Delta$, is also a provable goal. The fact that a rule is reversible means that both sides of the rule are equally provable, and so it can never cause us any harm to use the rule during proof search when its available. Any rule that is not reversible, like $\otimes R$, is called *irreversible*. For example, the $\oplus R_1$ rule is irreversible, since provability of $\Gamma \vdash A_1 \oplus A_2, \Delta$ certainly does not imply provability of $\Gamma \vdash A_1, \Delta$, as we may have needed to deduce A_2 instead.

As it turns out, the connectives $\oplus$, $\otimes$, 0, and 1 all have reversible left rules and irreversible right rules. On the other hand, their symmetric counterparts, $\&$, $\parr$, $\top$, and $\bot$ all have reversible right rules and irreversible left rules. We can concisely summarize the reversibility properties of these sets of rules by diving them into two camps and assigning each one a *polarity*: positive connectives (with reversible left rules) and negative connectives (with reversible right rules). The polarity of the connectives in linear logic is shown in Figure 18.

Returning to focalization, we can use the polarity of a connective to guide the building of a proof so that we cut down on the potential search space and avoid less wasted work on a dead end conjecture. The general procedure of focalization is to separate the development of a proof into two distinct, alternating phases:

1. *asynchronous*: where we apply as many reversible rules as possible, and their order doesn't matter (hence the name). In this phase, we use any available left rules for positive connectives and right rules for negative ones.
2. *synchronous*: where we pick a single proposition (the *focus*) and break it down as much as possible with irreversible rules. In this phase, we use right rules for positive connectives and left rules for negative ones.

Then, the general procedure for proof search is begin with the asynchronous phase and carry it out for as long as possible before switching to the synchronous phase, so that this work cannot be undone due to backtracking caused by a wrong move in the following synchronous phase. The point of sticking with a single focus during the synchronous phase is so that once a dangerous move has made, we should carry this train of thought through to the end to see if we've made a mistake early as possible.

These principles of focalized proof search can be reflected in the logic itself, as shown in Figure 19. The basic idea is to syntactically point out that a particular proposition is in focus by placing it in a so-called *stoup* [52], either on the right ($\Gamma \vdash A; \Delta$) or on the left ($\Gamma; A \vdash \Delta$), in contrast to the usual sequent judgment where no particular proposition stands out. The new form of judgments give a syntactic account of the phases: the synchronous phase (a sequent with a focus on the left or right) and the asynchronous phase (an unfocused sequent). With a syntactic notion of focus, we can restrict the dangerous, irreversible rules so that they can only be applied to a proposition that is already in focus, whereas the reversible rules apply to unfocused judgments. Additionally, we need to add structural rules, FR and FL, that assert that a focused proof is as good as an unfocused proof (read top down), and which correspond to picking a proposition to focus on (read bottom up). We also have the opposite structural rules, BR and BL, that "blur" a sequent by removing a proposition from focus (read bottom up) if it has the wrong polarity.

$$
\begin{aligned}
A^+, B^+, C^+ \in \mathit{Proposition}^+ &::= X^+ \mid A \oplus B \mid A \otimes B \mid 0 \mid 1 \\
A^-, B^-, C^- \in \mathit{Proposition}^- &::= X^- \mid A \,\&\, B \mid A \parr B \mid \bot \mid \top \\
A, B, C \in \mathit{Proposition} &::= A^+ \mid A^- \\
\Gamma \in \mathit{Hypothesis} &::= A_1, \ldots, A_n \\
\Delta \in \mathit{Consequence} &::= A_1, \ldots, A_n \\
\mathit{Judgment} &::= \Gamma \vdash \Delta \mid \Gamma \vdash A; \Delta \mid \Gamma; A \vdash \Delta
\end{aligned}
$$

Axiom and cut:

$$\frac{}{A^+ \vdash A^+;}\ Ax^+ \qquad \frac{}{;A^- \vdash A^-}\ Ax^- \qquad \frac{\Gamma_1 \vdash A, \Delta_1 \quad \Gamma_2, A \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ Cut$$

Logical rules:

Positive:

$$\frac{\Gamma \vdash A_1; \Delta}{\Gamma \vdash A_1 \oplus A_2; \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash A_2; \Delta}{\Gamma \vdash A_1 \oplus A_2; \Delta}\ \oplus R_2$$

$$\frac{\Gamma, A_1 \vdash \Delta \quad \Gamma, A_2 \vdash \Delta}{\Gamma, A_1 \oplus A_2 \vdash \Delta}\ \oplus L$$

no $0R$ rule $\qquad \frac{}{\Gamma, 0 \vdash \Delta}\ 0L$

$$\frac{\Gamma_1 \vdash A_1; \Delta_1 \quad \Gamma_2 \vdash A_2; \Delta_2}{\Gamma_1, \Gamma_2 \vdash A_1 \otimes A_2; \Delta_1, \Delta_2}\ \otimes R$$

$$\frac{\Gamma, A_1, A_2 \vdash \Delta}{\Gamma, A_1 \otimes A_2 \vdash \Delta}\ \otimes L$$

$$\frac{}{\vdash 1;}\ 1R \qquad \frac{\Gamma \vdash \Delta}{\Gamma, 1 \vdash \Delta}\ 1L$$

Negative:

$$\frac{\Gamma; A_1 \vdash \Delta}{\Gamma; A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma; A_1 \vdash \Delta}{\Gamma; A_1 \,\&\, A_2 \vdash \Delta}\ \&L_2$$

$$\frac{\Gamma \vdash A_1, \Delta \quad \Gamma \vdash A_2, \Delta}{\Gamma \vdash A_1 \,\&\, A_2, \Delta}\ \&R$$

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

$$\frac{\Gamma_1; A_1 \vdash \Delta_1 \quad \Gamma_2; A_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2; A_1 \parr A_2 \vdash \Delta_1, \Delta_2}\ \parr L$$

$$\frac{\Gamma \vdash A_1, A_2, \Delta}{\Gamma \vdash A_1 \parr A_2, \Delta}\ \parr R$$

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

Structural rules:

$$\frac{\Gamma \vdash A^+; \Delta}{\Gamma \vdash A^+, \Delta}\ FR \qquad \frac{\Gamma; A^- \vdash \Delta}{\Gamma, A^- \vdash \Delta}\ FL$$

$$\frac{\Gamma \vdash A^-, \Delta}{\Gamma \vdash A^-; \Delta}\ BR \qquad \frac{\Gamma, A^+ \vdash \Delta}{\Gamma; A^+ \vdash \Delta}\ BL$$

Figure 19. Focalized sequent calculus with two forms of conjunction ($\otimes$, $\&$), disjunction ($\oplus$, $\parr$), truth (1, $\top$), and falsehood (0, $\bot$).

As it turns out, the concept of polarity in the sequent calculus corresponds with the choice of Dummett's two approaches: positive is to verificationist as negative is to pragmatist. Moreover, the verificationist and pragmatist notions of canonical proofs correspond to focusing on the right and left of the sequent, respectively. Focusing on the right forces us to continue to build a positive conclusion with right rules beginning from basic atoms of truth: axioms, standing in for constants or unknown proofs, or from a negative conclusion that lives in the pragmatist world and cannot be broken down further into atomic pieces. Similarly, focusing on the left forces us to continue to build negative assumptions from basic atoms of falsehood.

6.2 Positive connectives

Both Zeilberger [116, 117] as well as Curien and Munch-Maccagnoni [24, 87] observed the similarities between polarity and focalization in proof search and structures in programming languages. For now, let us restrict our attention to the positive connectives, which correspond to a verificationist style of proof. The verificationist style structures rules of formation so that the deductions of a proposition fall within a fixed set of well-known canonical forms, whereas the uses of a proposition are arbitrary. Therefore, in a program that corresponds with a verificationist proof, the terms that produce output also must fall within a fixed set of forms. The co-terms of that consume input, however, are allowed to be arbitrary. In order to gain a foot-hold on the unrestricted nature of positive co-terms, we may describe them by *inversion* on the possible forms of their input. That is to say, positive co-terms may be defined by *cases* on all possible structures that it may receive as input.

For example, let's consider how to interpret the rules for $\oplus$ as verificationist programs of sum types by extending the basic structural core from Figure 8.[6] On the one hand, we have two rules for building a proof concluding $A_1 \oplus A_2$, so we can distinguish which of the two rules was chosen for building a term by using the injection constructors inl and inr, like we did for Wadler's dual sequent calculus:

$$\frac{\Gamma \vdash M : A_1 | \Delta}{\Gamma \vdash \mathsf{inl}(M) : A_1 \oplus A_2 | \Delta} \oplus R_1 \qquad \frac{\Gamma \vdash M : A_2 | \Delta}{\Gamma \vdash \mathsf{inr}(M) : A_1 \oplus A_2 | \Delta} \oplus R_2$$

On the other hand, the use of $A_1 \oplus A_2$ is permitted to be open ended; the only requirement is that it be well-defined with respect to the constructed forms of the right rules. Therefore, we can build a co-term by using case analysis on the two constructors for sums, allowing for them to give arbitrary behavior so long as they give behavior for both cases:

$$\frac{S_1 : (\Gamma, x_1 : A_1 \vdash \Delta) \quad S_2 : (\Gamma, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\, [\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inl}(x_2) \Rightarrow S_2] : A_1 \oplus A_2 \vdash \Delta} \oplus L$$

denoted **case** by analogy to case statements in functional programming languages. This terminology is not an accident; the above use of constructors and case analysis for programming with sum types is analogous to the use of algebraic data types in functional languages.

Following the verificationist development of sums, we can also give programs for the other positive connectives. The general pattern for the verificationist approach to the positive connectives is that the terms are formed by construction, similar to Wadler's dual sequent calculus, whereas the co-terms are formed by case analysis on term constructors. The $\otimes$ connective is interpreted as a pair data type, corresponding to a special case of tuples from functional languages. The one rule for concluding $A_1 \otimes A_2$ gives us a single constructor for forming pairs:

$$\frac{\Gamma \vdash M_1 : A_1 | \Delta \quad \Gamma \vdash M_2 : A_2 | \Delta}{\Gamma \vdash (M_1, M_2) : A_1 \otimes A_2 | \Delta} \otimes R$$

along with a co-term for performing case analysis on pairs:

$$\frac{S(\Gamma, x_1 : A_1, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\, [(x_1, x_2) \Rightarrow S] : A_1 \otimes A_2 \vdash \Delta} \otimes L$$

Finally, we can incorporate continuations into the verificationist paradigm by introducing a constructor for building a negated term out of a co-term, corresponding to the rule $\neg R$ from Figure 5:

$$\frac{\Gamma | K : A \vdash \Delta}{\Gamma \vdash \mathsf{not}^+(K) : \neg^+ A | \Delta} \neg^+ R$$

and the corresponding co-term that matches on the constructor:

$$\frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma | \mathbf{case}\, [\mathsf{not}^+(\alpha) \Rightarrow S] : \neg^+ A \vdash \Delta} \neg^+ L$$

This system for positive types is shown in Figure 20, which extends Figure 8 and is based on the positive fragment of system L [87].

In order to incorporate focalization into the system, we can restrict the typing rules for terms in the same way as the focalized logic in Figure 19. As it turns out, the restricted forms of focalized proofs correspond exactly to the definitions of values in the call-by-value setting, as in Figure 13. The focalized variant of positive system L [24] is shown in Figure 21. Note that the syntax of programs has been restricted; for example, we can no longer write the (non-value) term $\mathsf{inl}(\mathbf{out}\, \alpha \Leftarrow S)$. However, we can write such a term in expanded form by first giving a name to the non-value component:

$$\mathsf{inl}(\mathbf{out}\, \alpha \Leftarrow S) = \mathbf{out}\, \beta \Leftarrow \langle \mathbf{out}\, \alpha \Leftarrow S \| \mathbf{in}\, x \Rightarrow \langle \mathsf{inl}(x) \| \beta \rangle \rangle$$

So the focalized syntax can be seen as a more verbose, but more explicitly ordered, alternative that introduces more naming for non-value terms.

The concept of reversibility also gives us an answer to the fundamental non-determinism of the classical sequent calculus. When faced with a command like $\langle \mathbf{out}\, \alpha \Leftarrow S_1 \| \mathbf{in}\, x \Rightarrow S_2 \rangle$, we can use the type of the command to tell us what the term and the co-term "really look like." For example, if the command is between a term and co-term of type $A \otimes B$, then the typing derivation has the following form:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ S_1 : (\Gamma \vdash \alpha : A \otimes B, \Delta)\end{array}}{\Gamma \vdash \mathbf{out}\, \alpha \Leftarrow S_1 : A \otimes B | \Delta}\, AR \qquad \dfrac{\begin{array}{c}\mathcal{E}\\ \vdots\\ S_2 : (\Gamma, x : A \otimes B \vdash \Delta)\end{array}}{\Gamma | \mathbf{in}\, x \Rightarrow S_2 : A \otimes B \vdash \Delta}\, AL}{\langle \mathbf{out}\, \alpha \Leftarrow S_1 \| \mathbf{in}\, x \Rightarrow S_1 \rangle : \Gamma \vdash \Delta}\, Cut$$

However, since we know that the left rule for $\otimes$ is reversible, we can achieve an equivalent co-term that ends with $\otimes L$:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \vdots\\ S_1 : (\Gamma \vdash \alpha : A \otimes B, \Delta)\end{array}}{\Gamma \vdash \mathbf{out}\, \alpha \Leftarrow S_1 : A \otimes B | \Delta}\, AR \qquad \dfrac{\begin{array}{c}\mathcal{E}'\\ \vdots\\ S_2' : (\Gamma, x : A, y : B \vdash \Delta)\end{array}}{\Gamma | \mathbf{case}\, [(x, y) \Rightarrow S_2'] : A \otimes B \vdash \Delta}\, \otimes L}{\langle \mathbf{out}\, \alpha \Leftarrow S_1 \| \mathbf{case}\, [(x, y) \Rightarrow S_2'] \rangle : \Gamma \vdash \Delta}\, Cut$$

Therefore, by using the reversibility of the typing rules, we discovered that there wasn't an issue after all, revealing the fact that the co-term was in a sense "lying" by misrepresenting its intent. On the other hand, if we have the command $\langle V \| \mathbf{in}\, x \Rightarrow S \rangle$, where V is a value (*i.e.,* a focused term) then it is safe to substitute V for x since it must be a pair (V_1, V_2) (or a variable standing in for a pair).

[6] Note that we are now leaving the linear world and returning to classical logic with the unrestrained management of resources.

$$A, B, C \in Type ::= X \mid A \oplus B \mid A \otimes B \mid \neg^+ A$$
$$M \in Term ::= x \mid \mathbf{out}\,\alpha \Leftarrow S \mid \mathsf{inl}(M) \mid \mathsf{inr}(M) \mid (M, M) \mid \mathsf{not}^+(K)$$
$$K \in CoTerm ::= \alpha \mid \mathbf{in}\,x \Rightarrow S \mid \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow S | \mathsf{inr}(y) \Rightarrow S] \mid \mathbf{case}\,[(x, y) \Rightarrow S] \mid \mathbf{case}\,[\mathsf{not}^+(\alpha) \Rightarrow S]$$

Logical rules:

$$\frac{\Gamma \vdash M : A_1 | \Delta}{\Gamma \vdash \mathsf{inl}(M) : A_1 \oplus A_2 | \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash M : A_2 | \Delta}{\Gamma \vdash \mathsf{inr}(M) : A_1 \oplus A_2 | \Delta}\ \oplus R_2$$

$$\frac{S_1 : (\Gamma, x_1 : A_1 \vdash \Delta) \quad S_2 : (\Gamma, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\,[\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inl}(x_2) \Rightarrow S_2] : A_1 \oplus A_2 \vdash \Delta}\ \oplus L$$

$$\frac{\Gamma \vdash M_1 : A_1 | \Delta \quad \Gamma \vdash M_2 : A_2 | \Delta}{\Gamma \vdash (M_1, M_2) : A_1 \otimes A_2 | \Delta}\ \otimes R \qquad \frac{S(\Gamma, x_1 : A_1, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\,[(x_1, x_2) \Rightarrow S] : A_1 \otimes A_2 \vdash \Delta}\ \otimes L$$

$$\frac{\Gamma | K : A \vdash \Delta}{\Gamma \vdash \mathsf{not}^+(K) : \neg^+ A | \Delta}\ \neg^+ R \qquad \frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma | \mathbf{case}\,[\mathsf{not}^+(\alpha) \Rightarrow S] : \neg^+ A \vdash \Delta}\ \neg^+ L$$

Figure 20. Syntax and types for system L (the positive fragment).

$$A, B, C \in Type ::= X \mid A \oplus B \mid A \otimes B \mid \neg^+ A$$
$$V \in Value ::= x \mid \mathsf{inl}(V) \mid \mathsf{inr}(V) \mid (V, V) \mid \mathsf{not}^+(V)$$
$$M \in Term ::= V \mid \mathbf{out}\,\alpha \Leftarrow S$$
$$K \in CoTerm ::= \alpha \mid \mathbf{in}\,x \Rightarrow S \mid \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow S | \mathsf{inr}(y) \Rightarrow S] \mid \mathbf{case}\,[(x, y) \Rightarrow S] \mid \mathbf{case}\,[\mathsf{not}^+(\alpha) \Rightarrow S]$$
$$S \in Command ::= \langle M \| K \rangle$$
$$\Gamma \in Input ::= x_1 : A_1, \ldots, x_n : A_n$$
$$\Delta \in Output ::= \alpha_1 : A_1, \ldots, \alpha_n : A_n$$
$$Judgment ::= S : (\Gamma \vdash \Delta) \mid \Gamma \vdash V : A; \Delta \mid \Gamma \vdash M : A | \Delta \mid \Gamma | K : A \vdash \Delta$$

Axiom and cut:

$$\frac{}{\Gamma, x : A \vdash x : A; \Delta}\ Var \qquad \frac{}{\Gamma | \alpha : A \vdash \alpha : A, \Delta}\ CoVar \qquad \frac{\Gamma \vdash M : A | \Delta \quad \Gamma | K : A \vdash \Delta}{\langle M \| K \rangle : (\Gamma \vdash \Delta)}\ Cut$$

Logical rules:

$$\frac{\Gamma \vdash V : A_1; \Delta}{\Gamma \vdash \mathsf{inl}(V) : A_1 \oplus A_2; \Delta}\ \oplus R_1 \qquad \frac{\Gamma \vdash V : A_2; \Delta}{\Gamma \vdash \mathsf{inr}(V) : A_1 \oplus A_2; \Delta}\ \oplus R_2$$

$$\frac{S_1 : (\Gamma, x_1 : A_1 \vdash \Delta) \quad S_2 : (\Gamma, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\,[\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inl}(x_2) \Rightarrow S_2] : A_1 \oplus A_2 \vdash \Delta}\ \oplus L$$

$$\frac{\Gamma \vdash V_1 : A_1; \Delta \quad \Gamma \vdash V_2 : A_2; \Delta}{\Gamma \vdash (V_1, V_2) : A_1 \otimes A_2; \Delta}\ \otimes R \qquad \frac{S(\Gamma, x_1 : A_1, x_2 : A_2 \vdash \Delta)}{\Gamma | \mathbf{case}\,[(x_1, x_2) \Rightarrow S] : A_1 \otimes A_2 \vdash \Delta}\ \otimes L$$

$$\frac{\Gamma | K : A \vdash \Delta}{\Gamma \vdash \mathsf{not}^+(K) : \neg^+ A; \Delta}\ \neg^+ R \qquad \frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma | \mathbf{case}\,[\mathsf{not}^+(\alpha) \Rightarrow S] : \neg^+ A \vdash \Delta}\ \neg^+ L$$

Structural rules:

$$\frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mathbf{out}\,\alpha \Leftarrow S : A | \Delta}\ AR \qquad \frac{S : (\Gamma, x : A \vdash \Delta)}{\Gamma | \mathbf{in}\,x \Rightarrow S : A \vdash \Delta}\ AL \qquad \frac{\Gamma \vdash V : A; \Delta}{\Gamma \vdash V : A | \Delta}\ FR$$

Figure 21. Syntax and types for focalized system L (the positive fragment).

$$
\begin{array}{rr}
(\mu) & \langle \mathbf{out}\,\alpha \Leftarrow S \| K\rangle \to S\,\{K/\alpha\} \\
(\tilde{\mu}_V) & \langle V \| \mathbf{in}\,x \Rightarrow S\rangle \to S\,\{V/x\} \\
(\beta^{\oplus}) & \langle \mathsf{inl}(V) \| \mathbf{case}\,[\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inr}(x_2) \Rightarrow S_2]\rangle \to S_1\,\{V/x_1\} \\
(\beta^{\oplus}) & \langle \mathsf{inr}(V) \| \mathbf{case}\,[\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inr}(x_2) \Rightarrow S_2]\rangle \to S_2\,\{V/x_2\} \\
(\beta^{\otimes}) & \langle (V_1, V_2) \| \mathbf{case}\,[(x_1, x_2) \Rightarrow S]\rangle \to S\,\{V_1, V_2/x_1, x_2\} \\
(\beta^{\neg^+}) & \langle \mathsf{not}^+(K) \| \mathbf{case}\,[\mathsf{not}^+(\alpha) \Rightarrow S]\rangle \to S\,\{K/\alpha\}
\end{array}
$$

Figure 22. Reduction in focalized system L (the positive fragment).

Therefore polarity of the type of a cut can tell us "who is the liar," and restore determinism in an analogous manner as in Section 5.

The β rules for system L follows similar rules as case analysis in functional languages. For example, we have sum types which select a branch of the case based on the constructor tag:

$$\langle \mathsf{inl}(V) \| \mathbf{case}\,[\mathsf{inl}(x_1) \Rightarrow S_1 | \mathsf{inr}(x_2) \Rightarrow S_2]\rangle \to S_1\,\{V/x_1\}$$

and pair types which decompose the pair into elements:

$$\langle (V_1, V_2) \| \mathbf{case}\,[(x_1, x_2) \Rightarrow S]\rangle \to S\,\{V_1, V_2/x_1, x_2\}$$

In Figure 22, we give the rules for reduction of the positive fragment of focalized system L [24]. If we want to evaluate programs in the unfocalized syntax, we run into similar problems as Wadler's dual sequent calculus in Section 5. The solution is the same [87]: add additional reduction rules, ς, for lifting terms out of constructors, analogous to the ς rules of Wadler's sequent calculus.

6.3 Negative connectives

In comparison to the positive connectives, the negative connectives and pragmatist approach to proofs may seem a bit unusual. Instead of focusing on how we may conclude true facts we instead take the dual approach and focus our attention on how we may make use of those facts. In this way, we limit the ways that we may use an assumed proposition to a fixed set of known canonical forms, whereas the conclusions of a proposition are arbitrary. The programs that correspond with pragmatist proofs are likewise dual to verificationist proofs, so that the relative roles of producers and consumers are reversed. In a pragmatist program, the terms that produce output are allowed to be completely arbitrary. Instead, it is the co-terms that consume input that must fall within a fixed set of known forms — the legal observations of a type. We may then define terms by inversion on the possible forms of their consumer, so that they are given by cases on the observation of their output.

To begin our exploration of negative connectives, consider how to interpret the rules for & as types for a pragmatist programming construct. According to the pragmatist programmer, the most fundamental thing a type tells us is how we may use information of that type. Building on the structural core from Figure 8, the two rule for building a proof assuming $A_1 \,\&\, A_2$ gives us two ways to construct a consumer for processing information of type $A_1 \,\&\, A_2$ in terms of the two fixed projections fst and snd like we did for Wadler's dual sequent calculus:

$$\frac{\Gamma | K : A_1 \vdash \Delta}{\Gamma | \,\mathsf{fst}[K] : A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma | K : A_2 \vdash \Delta}{\Gamma | \,\mathsf{snd}[K] : A_1 \,\&\, A_2 \vdash \Delta}\ \&L_2$$

For example, if we have a co-term K_1 that accepts something of type A_1, then $\mathsf{fst}[K_1]$ accepts something of type $A_1 \,\&\, A_2$ by requesting the first component and forwarding it along to K_1. Likewise, if K_2 accepts type A_2, then $\mathsf{snd}[K_2]$ accepts $A_1 \,\&\, A_2$ by requesting the second component and forwarding it to K_2.

By the pragmatist approach, the production of $A_1 \,\&\, A_2$ is allowed to be open ended; the only requirement is that it be well-defined with respect to the constructed forms of the left rules. How then are we to define what a term of type $A_1 \,\&\, A_2$ looks like? Rather than try to be clever, we will utilize the symmetry of the sequent calculus and make terms of type $A_1 \,\&\, A_2$ analogous to co-terms of type $A_1 \oplus A_2$. More specifically, dual to the fact that co-terms of a positive type are defined by case analysis on the terms of that type, terms of a negative type are defined by case analysis on the co-terms of that type. This means that terms of the type $A_1 \& A_2$ give rise to a *dual form of case analysis* than is found in functional programming languages:

$$\frac{S_1 : (\Gamma \vdash \alpha_1 : A_1, \Delta) \quad S_2 : (\Gamma \vdash \alpha_2 : A_2, \Delta)}{\Gamma \vdash \mathbf{case}\,(\mathsf{fst}[\alpha_1] \Leftarrow S_1 | \mathsf{snd}[\alpha_2] \Leftarrow S_2) : A_1 \,\&\, A_2 | \Delta}\ \&R$$

Intuitively, a term of type $A_1 \,\&\, A_2$ is an abstract form of product that says what to do when it is asked for the first component and when it is asked for the second component, running some arbitrary behavior that (may) respond to the underlying co-term expecting a result of type A_1 or A_2. In other words, a term of type $A_1 \,\&\, A_2$ considers all the cases for the possible observations that may be made upon it, and defines some specific behavior for each observation. The pragmatist style of programming resembles a form of message passing, where the observations fit into a fixed set of molds (the possible messages), and the producers decide on what to do based on the observation.

Let us now extend the pragmatist approach to products ($A_1 \,\&\, A_2$) to also give programs that correspond to the other negative connectives. The general pattern for the pragmatist approach to the negative connectives is that the co-terms are formed by construction, similar to Wadler's dual sequent calculus, whereas the terms are formed by case analysis on co-term constructors. The $\parr$ connective is interpreted as an abstract form of disjunction. The one rule for assuming $A_1 \parr A_2$, $\parr L$, gives us a single constructor for forming a message containing two branches, one for accepting A_1 and one for A_2:

$$\frac{\Gamma | K_1 : A_1 \vdash \Delta \quad \Gamma | K_2 : A_2 \vdash \Delta}{\Gamma | [K_1, K_2] : A_1 \parr A_2 \vdash \Delta}\ \parr L$$

along with a term for responding to the message:

$$\frac{S : (\Gamma \vdash \alpha_1 : A_1, \alpha_2 : A_2, \Delta)}{\Gamma \vdash \mathbf{case}\,([\alpha_1, \alpha_2] \Leftarrow S) : A_1 \parr A_2 | \Delta}\ \parr R$$

Intuitively, we may think of $[K_1, K_2]$ as a concrete counterpart to the abstract form of case analysis given for $A_1 \oplus A_2$, both sub-co-terms are two messages given to the term of type $A_1 \parr A_2$ that serve as two branches that the term may take. For example, a term of type $\mathsf{Int} \parr \mathsf{Int}$ may send an integer to the left branch, like $\mathbf{case}\,([\alpha, \beta] \Leftarrow \langle 1 \| \alpha\rangle)$, or to the right branch, like $\mathbf{case}\,([\alpha, \beta] \Leftarrow \langle 2 \| \beta\rangle)$, which serve similar roles as the constructed terms $\mathsf{inl}(1)$ and $\mathsf{inr}(2)$. Finally, the pragmatist paradigm gives a different formulation of continuations with a single co-term

$$
\begin{array}{rcl}
A, B, C \in \mathit{Type} & ::= & X \mid A \,\&\, B \mid A \parr B \mid \bot \mid \top \mid \neg^- B \\
M \in \mathit{Term} & ::= & x \mid \mathbf{out}\,\alpha \Leftarrow S \mid \mathbf{case}\,(\mathsf{fst}[\alpha_1] \Leftarrow S_1 | \mathsf{snd}[\alpha_2] \Leftarrow S_2) \mid \mathbf{case}\,([\alpha_1, \alpha_2] \Leftarrow S) \mid \mathbf{case}\,(\mathsf{not}^-[x] \Leftarrow S) \\
E \in \mathit{CoValue} & ::= & \alpha \mid \mathsf{fst}[E] \mid \mathsf{snd}[E] \mid [E, E] \mid \mathsf{not}^-[M] \\
K \in \mathit{CoTerm} & ::= & E \mid \mathbf{in}\,x \Rightarrow S \\
S \in \mathit{Command} & ::= & \langle M \| K \rangle \\
\Gamma \in \mathit{Input} & ::= & x_1 : A_1, \ldots, x_n : A_n \\
\Delta \in \mathit{Output} & ::= & \alpha_1 : A_1, \ldots, \alpha_n : A_n \\
\mathit{Judgment} & ::= & S : (\Gamma \vdash \Delta) \mid \Gamma \vdash M : A | \Delta \mid \Gamma; E : A \vdash \Delta \mid \Gamma | K : A \vdash \Delta
\end{array}
$$

Axiom and cut:

$$\frac{}{\Gamma, x : A \vdash x : A | \Delta}\ \mathit{Var} \qquad \frac{}{\Gamma; \alpha : A \vdash \alpha : A, \Delta}\ \mathit{CoVar} \qquad \frac{\Gamma \vdash M : A | \Delta \quad \Gamma | K : A \vdash \Delta}{\langle M \| K \rangle : (\Gamma \vdash \Delta)}\ \mathit{Cut}$$

Logical rules:

$$\frac{\Gamma; E : A_1 \vdash \Delta}{\Gamma; \mathsf{fst}[E] : A_1 \,\&\, A_2 \vdash \Delta}\ \&L_1 \qquad \frac{\Gamma; E : A_2 \vdash \Delta}{\Gamma; \mathsf{snd}[E] : A_1 \,\&\, A_2 \vdash \Delta}\ \&L_2$$

$$\frac{S_1 : (\Gamma \vdash \alpha_1 : A_1, \Delta) \quad S_2 : (\Gamma \vdash \alpha_2 : A_2, \Delta)}{\Gamma \vdash \mathbf{case}\,(\mathsf{fst}[\alpha_1] \Leftarrow S_1 | \mathsf{snd}[\alpha_2] \Leftarrow S_2) : A_1 \,\&\, A_2 | \Delta}\ \&R$$

$$\frac{\Gamma; E_1 : A_1 \vdash \Delta \quad \Gamma; E_2 : A_2 \vdash \Delta}{\Gamma; [E_1, E_2] : A_1 \parr A_2 \vdash \Delta}\ \parr L \qquad \frac{S : (\Gamma \vdash \alpha_1 : A_1, \alpha_2 : A_2, \Delta)}{\Gamma \vdash \mathbf{case}\,([\alpha_1, \alpha_2] \Leftarrow S) : A_1 \parr A_2 | \Delta}\ \parr R$$

$$\frac{\Gamma \vdash M : A | \Delta}{\Gamma; \mathsf{not}^-[M] : \neg^- A \vdash \Delta}\ \neg^- L \qquad \frac{S : (\Gamma, x : A \vdash \Delta)}{\Gamma \vdash \mathbf{case}\,(\mathsf{not}^-[x] \Leftarrow S) : \neg^- A | \Delta}\ \neg^- R$$

Structural rules:

$$\frac{S : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mathbf{out}\,\alpha \Leftarrow S : A | \Delta}\ AR \qquad \frac{S : (\Gamma, x : A \vdash \Delta)}{\Gamma | \mathbf{in}\,x \Rightarrow S : A \vdash \Delta}\ AL \qquad \frac{\Gamma; E : A \vdash \Delta}{\Gamma | E : A \vdash \Delta}\ FL$$

Figure 23. Syntax and types for focalized system L (the negative fragment).

constructor corresponding to the rule $\neg L$ from Figure 5:

$$\frac{\Gamma \vdash M : A | \Delta}{\Gamma | \,\mathsf{not}^-[M] : \neg^- A \vdash \Delta}\ \neg^- L$$

along with a term that matches on the constructor of the message:

$$\frac{S : (\Gamma, x : A \vdash \Delta)}{\Gamma \vdash \mathbf{case}\,(\mathsf{not}^-[x] \Leftarrow S) : \neg^- A | \Delta}\ \neg^- R$$

Intuitively, the verificationist approach defines continuations by how they are produced (a continuation is *built* from a co-term, corresponding to the reification of an evaluation context), whereas the pragmatist approach defines continuations by how they operate (a continuation is *used* by throwing a value to it, corresponding to the throw operation of SML/NJ).

Remark 7. It is worthwhile to pause and ask why the pragmatist representation of logical connectives may appear to be backwards. For example, $\parr$ is a logical "or" whose interpretation appears to be an "and" combination of two things, whereas $\&$ is a logical "and" that whose interpretation appears to be an "or" choice of two alternatives. The reason is that the pragmatist approach requires us to completely reverse the way we think about proving. With the verificationist approach, the focus is on the way that we may establish truth: to show that "A and B" is true, we need to show that *both* A and B are true; to show that "A or B" is true, we it suffices to show that *either* A is true or B is true. Instead, the pragmatist approach asks us to focus on the ways that we may establish falsehood: to show that "A and B" is false, it suffices to show that *either* A is false or B is false; to show that "A or B" is false we need to show that *both* A and B are false. Whereas the verificationist is primarily concerned with building a *proof*, the pragmatist is instead concerned with building a *refutation*. Therefore, the pragmatist interpretation of negative connectives intuitively has a negation baked in: "and" is represented by a choice and "or" is represented by a pair because they are about refutations rather than proofs. *End remark* 7.

As with the verificationist approach to positive connectives, we can incorporate focalization into the pragmatist approach to negative connectives as well. The focalized variant of negative system L [24] is shown in Figure 23. Dually to the fact that focalized proofs of the positive fragment give rise to a definition of values, the focalized proofs of the negative fragment give rise to a definition of co-values in the call-by-name setting, exactly the same as in Figure 14. Again, focalization restricts the syntax of programs, so that we can no longer write the (non-co-value) co-term $\mathsf{fst}[\mathbf{in}\,x \Rightarrow S]$. Just like the terms in the positive fragment can be expanded into focalized form, the co-terms in the negative fragment can be expanded as well by giving a name to the non-co-value component:

$$\mathsf{fst}[\mathbf{in}\,x \Rightarrow S] = \mathbf{in}\,y \Rightarrow \langle \mathbf{in}\,\alpha \Rightarrow \langle y \| \mathsf{fst}[\alpha] \rangle \| \mathbf{in}\,x \Rightarrow S \rangle$$

$$(\mu_E) \qquad \langle \mathbf{out}\, \alpha \Leftarrow S \| E \rangle \to S\,\{E/\alpha\}$$

$$(\tilde{\mu}) \qquad \langle M \| \mathbf{in}\, x \Rightarrow S \rangle \to \{M/x\}$$

$$(\beta^{\&}) \qquad \langle \mathbf{case}\,(\mathsf{fst}[\alpha_1] \Leftarrow S_1 | \mathsf{snd}[\alpha_2] \Leftarrow S_2) \| \mathsf{fst}[E] \rangle \to S_1\,\{E/\alpha_1\}$$

$$(\beta^{\&}) \qquad \langle \mathbf{case}\,(\mathsf{fst}[\alpha_1] \Leftarrow S_1 | \mathsf{snd}[\alpha_2] \Leftarrow S_2) \| \mathsf{snd}[E] \rangle \to S_2\,\{E/\alpha_2\}$$

$$(\beta^{\parr}) \qquad \langle \mathbf{case}\,([\alpha_1, \alpha_2] \Leftarrow S) \| [E_1, E_2] \rangle \to S\,\{E_1, E_2/\alpha_1, \alpha_2\}$$

$$(\beta^{\neg^-}) \qquad \langle \mathbf{case}\,(\mathsf{not}^-[x] \Leftarrow S) \| \mathsf{not}^-[M] \rangle \to S\,\{M/x\}$$

Figure 24. Reduction in system L (the negative fragment).

So the focalized syntax for pragmatist programs is a more verbose way of explicitly ordering computation by introducing more naming for non-co-value co-terms.

The type-based answer to the fundamental non-determinism of the classical sequent calculus also extends to the negative connectives. This approach still relies on the reversibility of inference rules, except that because negative connectives are reversible in opposite ways to positive connectives, we get the opposite resolution to the dilemma. Suppose again that we are faced with the command $\langle \mathbf{out}\, \alpha \Leftarrow S_1 \| \mathbf{in}\, x \Rightarrow S_2 \rangle$ with a similar typing derivation as before, except that now x and α have the type $A \parr B$. We know that the right rule for $\parr$ is reversible, so we can turn the typing derivation into the more explicit:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}' \\ \vdots \\ S_1' : (\Gamma \vdash \alpha : A, \beta : B, \Delta)\end{array}}{\Gamma \vdash \mathbf{case}\,([\alpha, \beta] \Leftarrow S_1') : A \parr B | \Delta}\ \parr R \qquad \dfrac{\begin{array}{c}\mathcal{E} \\ \vdots \\ S_2 : (\Gamma, x : A \parr B \vdash \Delta)\end{array}}{\Gamma | \mathbf{in}\, x \Rightarrow S_2 : A \parr B \vdash \Delta}\ AL}{\langle \mathbf{case}\,([\alpha, \beta] \Leftarrow S_1') \| \mathbf{in}\, x \Rightarrow S_2 \rangle : (\Gamma \vdash \Delta)}\ Cut$$

giving us the more explicit command $\langle \mathbf{case}\,([\alpha, \beta] \Leftarrow S_1') \| \mathbf{in}\, x \Rightarrow S_2 \rangle$ that now spells out exactly which side should be prioritized. Therefore, for negative types, asking "who is the liar" gives the opposite answer, restoring determinism to the system by favoring the co-term over the term.

The β rules for the negative fragment of system L follow the same notion of case analysis as the positive fragment, except in the reverse direction. For example, terms of product types select a response based on the constructor tag of their observation:

$$\langle \mathbf{case}\,(\mathsf{fst}[\alpha] \Leftarrow S_1 | \mathsf{snd}[\beta] \Leftarrow S_2) \| \mathsf{fst}[E] \rangle \to S_1\,\{E/\alpha\}$$

and terms of sum types decompose their observation into the two independent branches:

$$\langle \mathbf{case}\,([\alpha, \beta] \Leftarrow S) \| [E_1, E_2] \rangle \to S\,\{E_1, E_2/\alpha, \beta\}$$

In Figure 24, we give the rules for reduction of the negative fragment of focalized system L [24]. The role between Wadler's ς rules in the call-by-name sequent calculus and evaluation of the unfocalized syntax is the same for the negative fragment of system L [87].

6.4 Synthetic connectives and deep patterns

Recall that the focalization procedure for proof search described in Section 6.1 was made up of two phases: asynchronous and synchronous. The type focalized type systems given in Figures 21 and 23 are concerned with enforcing the synchronous phase: when we apply an irreversible rule to a type, the stoup prevents us from switching to work on a completely different type (by activating some free variable with the AR or AL rules). However, these type systems to not attempt to enforce the completion of the synchronous and asynchronous phases. We may end a focused derivation in the synchronous phase using the Var and $CoVar$ rules whenever we like, even if we could continue to break apart the type in focus. We may switch from the asynchronous to the to the synchronous phase and begin using irreversible rules whenever we like, even if some reversible rules could still apply. Instead, Zeilberger's [116, 117] polarized logic enforces completion of the synchronous and asynchronous phases, leading to a notion of synthetic connectives which correspond to a mandatory form of deep pattern matching, similar to pattern matching found in functional programming languages.

We will begin to examine the consequences of forcing completion of the two phases to completion by considering only the positive fragment of the polarized logic. The first observation to make is that the judgments that mark the end of an asynchronous phase always follow a certain form. On the one hand, if we have the judgment $\Gamma, A \otimes B \vdash \Delta$, then it cannot be at the end of the asynchronous phase because we can apply the reversible left $\otimes$ rule to break the $A \otimes B$ on the left down further into $\Gamma, A, B \vdash \Delta$. On the other hand, if we have the judgment $\Gamma \vdash A \otimes B, \Delta$, then we cannot do anything with the $A \otimes B$ in the right since the right $\otimes$ rule is irreversible, thus forcing us to focus on it and enter the synchronous phase. Therefore, a positive judgment at the end of an asynchronous phase will always have the form $X_1, \ldots, X_n \vdash A_1, \ldots, A_m$: the $A_1, \ldots, A_m$ to the right of the sequent are positive propositions that do not have reversible rules to apply, and the $X_1, \ldots, X_n$ are positive proposition variables that we cannot do anything with since they are an unknown. This form of judgment is *stable* since no reversible rules can apply to it, and thus it marks the end of an asynchronous phase. A stable judgment corresponds to only allowing variables of unknown positive types, while letting co-variables be of any positive type.

The second observation to make is that a proposition composed out of positive connectives can be fused together into a single *synthetic* connective by describing the *patterns* of its proofs. For example, suppose we would like to prove the positive proposition $X \otimes (Y \oplus \neg^+ A)$ using the rules from Figure 19. There are two prototypical, focused derivations of this proposition. The first derivation attempts to prove both X and Y using the irreversible $\otimes R$ and $\oplus R_1$ rules:

$$\dfrac{\dfrac{}{X \vdash X;}\ Ax^+ \qquad \dfrac{\dfrac{}{Y \vdash Y;}\ Ax^+}{Y \vdash Y \oplus \neg^+ A;}\ \oplus R_1}{X, Y \vdash X \otimes (Y \oplus \neg^+ A);}\ \otimes R$$

We may complete this prove by taking X and Y as hypotheses. The second derivation attempts to prove both X and $\neg^+ A$ using

the irreversible $\otimes R$, $\oplus R_2$, and $\neg^+ A$ rules:

$$\dfrac{\dfrac{}{X \vdash X;}\ Ax^+ \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \vdots\\ \Gamma, A \vdash \Delta\end{array}}{\Gamma \vdash \neg^+ A; \Delta}\ \neg^+ R}{\Gamma \vdash Y \oplus \neg^+ A; \Delta}\ \oplus R_2}{X, \Gamma \vdash X \otimes (Y \oplus \neg^+ A); \Delta}\ \otimes R$$

Besides assuming X, we need the additional hypotheses Γ and consequences Δ. Since the premise of the $\neg^+ A$ rule puts the positive sub-proposition A into our assumptions, and we are not able to focus on an assumed positive proposition, the synchronous phase ends and we have some arbitrary proof $\mathcal{E}$ of the unfocused judgment $\Gamma, A \vdash \Delta$. Every single focalized proof of $X \otimes (Y \oplus \neg^+ A)$ must follow the shape of one of these two forms. In effect, we can treat this particular combination of the $\otimes$, $\oplus$ and $\neg^+$ connectives as a single synthesized connective defined by the following two right rules:

$$\dfrac{}{X, Y \vdash X \otimes (Y \oplus \neg^+ A);}\ \_ \otimes (\_ \oplus \neg^+ \_) R_1$$

$$\dfrac{\Gamma, A \vdash \Delta}{\Gamma, X \vdash X \otimes (Y \oplus \neg^+ A); \Delta}\ \_ \otimes (\_ \oplus \neg^+ \_) R_2$$

This observation may be formalized in terms of a language as *patterns* that expose named placeholders (*i.e.,* variables) deep within some structure, similar to patterns for data structures in functional programming languages. The first derivation of the synthetic rule $\_ \otimes (\_ \oplus \neg^+ \_) R_1$ is summarized by the pattern $(x, \mathsf{inl}(y))$ which contains the placeholders $x : X$ and $y : Y$ that stand in for the unknown assumptions:[7]

$$\dfrac{\dfrac{}{x : X \Rightarrow (x : X \vdash)} \quad \dfrac{\dfrac{}{y : Y \Rightarrow (y : Y \vdash)}}{\mathsf{inl}(y) : Y \oplus \neg^+ A \Rightarrow (y : Y \vdash)}}{(x, \mathsf{inl}(y)) : X \otimes (Y \oplus \neg^+ A) \Rightarrow (x : X, y : Y \vdash)}$$

The second derivation of the synthetic rule $\_ \otimes (\_ \oplus \neg^+ \_) R_2$ is summarized by the pattern $(x, \mathsf{inr}(\mathsf{not}^+(\alpha)))$ which contains the placeholders $x : X$ and $\alpha : A$, where α stands in for the unknown, unfocalized proof $\mathcal{E}$ for using A:

$$\dfrac{\dfrac{}{x : X \Rightarrow (x : X \vdash)} \quad \dfrac{\dfrac{}{\mathsf{not}^+(\alpha) : \neg^+ A \Rightarrow (\vdash \alpha : A)}}{\mathsf{inr}(\mathsf{not}^+(\alpha)) : Y \oplus \neg^+ A \Rightarrow (\vdash \alpha : A)}}{(x, \mathsf{inr}(\mathsf{not}^+(\alpha))) : X \otimes (Y \oplus \neg^+ A) \Rightarrow (x : X \vdash \alpha : A)}$$

By forcing completion the complete elaboration of the structure of the connectives, we effectively turn $X \otimes (Y \oplus \neg^+ A)$ into a single structure in its own right with the two irreversible right rules following the above proofs — at the end of the focalized derivation of the $X \otimes (Y \oplus \neg^+ A)$ patterns, we have completely atomized the structure into its constituent parts. The rules for forming patterns show us how to build these synthesized rules out of more basic principles for each combination of connectives.

The third observation to make is that a complete, synchronous phase of a focalized proof may be summarized by filling the placeholders left in a proof pattern. In the case of positive connectives, the synchronous phase for A has the form $\Gamma \vdash A; \Delta$ which gives us a notion of the *direct proofs* of A (recall that the judgment $\vdash A$ is interpreted as "with no assumptions A is true"). For example, suppose we have the value $(9, \mathsf{inl}(9))$ of type $\mathsf{Int} \otimes (\mathsf{Int} \oplus \neg^+ \mathsf{Int})$, which is a pair of two integers.[8] Then this value may be seen as filling the pattern $(x, \mathsf{inl}(y))$ so that both x and y are given the value 9. In other words the value $(9, \mathsf{inl}(9))$ lies in the image of a substitution over the proof pattern described by the synthetic $\_ \otimes (\_ \oplus \neg^+ \_) R_1$ rule, $(x, \mathsf{inl}(y))\,\{9/x\}\,\{9/y\}$. Other values of this same shape can be expressed as different substitutions over the same pattern. For example, the value $(9, \mathsf{inl}(5))$ is the same as $(x, \mathsf{inl}(y))\,\{9/x\}\,\{5/y\}$. The type $\mathsf{Int} \otimes (\mathsf{Int} \oplus \neg^+ \mathsf{Int})$ also allows for values of a different shape that made up of an integer and a co-term that accepts an integer like $(3, \mathsf{inr}(E))$. This value is equivalent to a substitution for the other pattern $(x, \mathsf{inr}(\alpha))$ equivalent to the $\_ \otimes (\_ \oplus \neg^+ \_) R_2$ rule: the value $(3, \mathsf{inr}(E))$ is the same as $(x, \mathsf{inr}(\alpha))\,\{9/x\}\,\{E/\alpha\}$. Observe that completion of the synchronous phase is enforced by the placeholders in a pattern must stand in for either a value of an unknown positive type X that has an unknown structure, or a co-value of a positive type that would not be in focus. Therefore, an entire synchronous phase of a verificationist proof of a positive proposition corresponds to a notion of value that fills in the details missing in a pattern of the type.

The fourth observation to make is that completely breaking down a single proposition in an asynchronous phase may be summarized by inversion on the possible proof patterns. In the case of the positive connectives, the asynchronous phase for A has the form $\Gamma, A \vdash \Delta$ which gives us a notion of *indirect refutations* of A (recall that the judgment $A \vdash$ is interpreted as "if A is true then false is true," or in other words "assuming A leads to a contradiction"). The general way to refute a positive proposition A is to suppose all the ways it could have been true (that is, all the proof patterns for direct proofs), and show that each possibility leads to a contradiction. In terms of programming in the sequent calculus, this method of refutation corresponds to matching on the patterns of A values, and giving a command for each pattern. Since we do not care about the order in which we match on the structure of values, the pattern match is deep, drilling down to the leaves of a structure in a single step. For example, refuting the proposition $X \otimes (Y \oplus \neg^+ A)$ corresponds to giving a co-value that pattern matches on values of type $X \otimes (Y \oplus \neg^+ A)$:

$$\dfrac{S_1 : (\Gamma, x : X, y : Y \vdash \Delta) \quad S_2 : (\Gamma, x : X \vdash \alpha : A, \Delta)}{\Gamma | \mathbf{case}\,[(x, \mathsf{inl}(y)) \Rightarrow S_1 | (x, \mathsf{inr}(\mathsf{not}^+(\alpha))) \Rightarrow S_2] : X \otimes (Y \oplus \neg^+ A) \vdash \Delta}$$

In this rule, we consider all the possible derivable patterns for the type $X \otimes (Y \oplus \neg^+ A)$, which turns out to be only the previously discussed patterns $(x, \mathsf{inl}(y))$ and $(x, \mathsf{inr}(\mathsf{not}^+(\alpha)))$. In order to derive a contradiction in each case, the underlying commands may reference placeholders in the patterns as well anything else in Γ and Δ. Observe that by inverting on the deep patterns, we have in effect applied all the possible reversible rules for $X \otimes (Y \oplus \neg^+ A)$. Therefore, the entire asynchronous phase for a verificationst refutation of a positive proposition corresponds to a notion of co-value that matches on the patterns of the type.

All together, the above observations form the basis of Zeilberger's $\mathbf{CU}$ calculus [116] (also known as $\mathcal{L}$ [117]). The syntax and types of the positive fragment of $\mathbf{CU}$ are given in Figure 26. Patterns take on a primary role in defining this language: values are written as an explicit syntax for substitution applied to a pattern, and co-values are written by encapsulating a map from patterns to commands. On the one hand, the rules for patterns describe the particular structural details for each individual positive connective.[9] On the other hand, the rules for values and co-values are de-

[7] The notation for pattern judgments is borrowed from case analysis on positive types from Section 6.2.

[8] For this example, the type Int is treated as neither a positive nor negative proposition, but instead some unknown atomic proposition of the same status as a proposition variable X.

[9] Notice that the typing rules for patterns use input and output environments as linear resources, similar to linear logic. This is because the (co-)variables

$$
\begin{array}{rcll}
A, B, C \in \mathit{Type} & ::= & X \mid A \oplus B \mid A \otimes B \mid \neg^{+} A \mid \ldots & \\
p \in \mathit{Pattern} & ::= & x \mid \mathsf{inl}(p) \mid \mathsf{inr}(p) \mid (p, p) \mid \mathsf{not}^{+}(\alpha) \mid \ldots & \\
\sigma \in \mathit{Substitution} & ::= & \{E_1/\alpha_1\} \ldots \{y_1/x_1\} \ldots & \\
V \in \mathit{Value} & ::= & p\sigma & \\
\phi \in \mathit{Map} & ::= & [p \Rightarrow S | \ldots] & \\
E \in \mathit{CoValue} & ::= & \mathbf{case}\,[\phi] & \\
S \in \mathit{Command} & ::= & \langle V \| \alpha \rangle \mid \langle V \| E \rangle & \\
\Gamma \in \mathit{Input} & ::= & x_1 : X_1, \ldots, x_n : X_n & \\
\Delta \in \mathit{Output} & ::= & \alpha_1 : A_1, \ldots, \alpha_n : A_n & \\
\mathit{Judgment} & ::= & p : A \Rightarrow (\Gamma \vdash \Delta) & \text{patterns} \\
& \mid & (\Gamma' \vdash \Delta')\sigma : (\Gamma \vdash \Delta) & \text{substitutions} \\
& \mid & \Gamma \vdash V : A; \Delta & \text{direct values} \\
& \mid & \Gamma | E : A \vdash \Delta & \text{indirect co-values} \\
& \mid & S : (\Gamma \vdash \Delta) & \text{commands}
\end{array}
$$

Pattern rules:

$$\frac{}{x : X \Rightarrow (x : X \vdash)}$$

$$\frac{p : A_1 \Rightarrow (\Gamma \vdash \Delta)}{\mathsf{inl}(p) : A_1 \oplus A_2 \Rightarrow (\Gamma \vdash \Delta)} \qquad \frac{p : A_2 \Rightarrow (\Gamma \vdash \Delta)}{\mathsf{inr}(p) : A_1 \oplus A_2 \Rightarrow (\Gamma \vdash \Delta)}$$

$$\frac{p_1 : A_1 \Rightarrow (\Gamma_1 \vdash \Delta_1) \quad p_2 : A_2 \Rightarrow (\Gamma_2 \vdash \Delta_2)}{(p_1, p_2) : A_1 \otimes A_2 \Rightarrow (\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2)}$$

$$\frac{}{\mathsf{not}^{+}(\alpha) : \neg^{+} A \Rightarrow (\vdash \alpha : A)}$$

$$\ldots$$

Substitution rules:

$$\frac{}{(\vdash)\epsilon : (\Gamma \vdash \Delta)} \qquad \frac{(\Gamma' \vdash \Delta')\sigma : (\Gamma, x : X \vdash \Delta)}{(\Gamma', x' : X \vdash \Delta')\sigma\,\{x/x'\} : (\Gamma, x : X \vdash \Delta)} \qquad \frac{\Gamma | E : A \vdash \Delta \quad (\Gamma' \vdash \Delta')\sigma : (\Gamma \vdash \Delta)}{(\Gamma' \vdash \alpha : A, \Delta')\sigma\,\{E/\alpha\} : (\Gamma \vdash \Delta)}$$

Value and co-value rules:

$$\frac{p : A \Rightarrow (\Gamma' \vdash \Delta') \quad (\Gamma' \Rightarrow \Delta')\sigma : (\Gamma \vdash \Delta)}{\Gamma \vdash p\sigma : A; \Delta} \qquad \frac{\forall (p : A \Rightarrow (\Gamma' \vdash \Delta')).\phi(p) : (\Gamma, \Gamma' \vdash \Delta, \Delta')}{\Gamma | \mathbf{case}\,[\phi] : A \vdash \Delta}$$

Cut rules:

$$\frac{\Gamma \vdash V : A; \alpha : A, \Delta}{\langle V \| \alpha \rangle : (\Gamma \vdash \alpha : A, \Delta)} \qquad \frac{\Gamma \vdash V : A; \Delta \quad \Gamma | E : A \vdash \Delta}{\langle V \| E \rangle : (\Gamma \vdash \Delta)}$$

Figure 25. Syntax and types of $\mathbf{CU}$ (the positive fragment).

fined once and for all, relying on patterns to describe the peculiarities of the particular type in question. A value is formed by filling in the holes of a pattern, and a co-value is formed by quantifying over all possible patterns of a type. The positive fragment of $\mathbf{CU}$ has judgments for forming patterns, substitutions, direct values, indirect co-values, and commands. The negative fragment of $\mathbf{CU}$, given in Figure 25, is the mirror image of the positive fragment, similar to the symmetry between the positive and negative fragments of system L, and forms direct co-values out of co-patterns and indirect values by matching over co-patterns.

Remark 8. The mappings from patterns to commands can be thought of as a finite set of pattern-command pairs, generalizing the form of case analysis from Figure 20. However, in the original presentation, the syntax is generalized even further so that maps so maps are considered as functions from the meta-theory, giving a *higher-order syntax* [115]. This technique allows us to use a broader form of pattern matching than is typically used in programming languages. For instance, we may define the natural numbers by an infinite set of patterns:

$$\frac{}{0 : \mathsf{Int} \Rightarrow (\vdash)} \qquad \frac{}{1 : \mathsf{Int} \Rightarrow (\vdash)} \qquad \frac{}{2 : \mathsf{Int} \Rightarrow (\vdash)} \qquad \ldots$$

Then, a co-value of type Int would be given by an infinite set of cases $\mathbf{case}\,[0 \Rightarrow S_0 | 1 \Rightarrow S_1 | 2 \Rightarrow S_2 | \ldots]$ given by a mapping between the natural numbers and commands. This technique effectively pushes the burden of defining well-founded relations over the natural numbers (perhaps by induction) to the meta-theory, instead of defining the meaning inside of the language itself. *End remark* 8.

We can also give reduction rules for pattern matching in $\mathbf{CU}$, as shown in Figure 27. Because the positive and negative forms of val-

in patterns represent a *unique* place in a structure, and should not be referenced more than once in the same pattern.

$$
\begin{array}{rcll}
A, B, C \in Type & ::= & X \mid A \,\&\, B \mid A \parr B \mid \neg^{-} A \mid \ldots & \\
q \in CoPattern & ::= & \alpha \mid \mathsf{fst}[q] \mid \mathsf{snd}[q] \mid [q, q] \mid \mathsf{not}^{-}[x] \mid \ldots & \\
\sigma \in Substitution & ::= & \left\{V_1'/x_1\right\} \ldots \{\beta_1/\alpha_1\} \ldots & \\
E \in CoValue & ::= & q\sigma & \\
\psi \in CoMap & ::= & (q \Leftarrow S \mid \ldots) & \\
V \in Value & ::= & \mathbf{case}\,(\psi) & \\
S \in Command & ::= & \langle x \| E \rangle \mid \langle V \| E \rangle & \\
\Gamma \in Input & ::= & x_1 : A_1, \ldots, x_n : A_n & \\
\Delta \in Output & ::= & \alpha_1 : X_1, \ldots, \alpha_n : X_n & \\
Judgment & ::= & q : A \Leftarrow (\Gamma \vdash \Delta) & \text{co-patterns} \\
& \mid & (\Gamma' \vdash \Delta')\sigma : (\Gamma \vdash \Delta) & \text{substitutions} \\
& \mid & \Gamma; E : A \vdash \Delta & \text{direct co-values} \\
& \mid & \Gamma \vdash V : A \mid \Delta & \text{indirect values} \\
& \mid & S : (\Gamma \vdash \Delta) & \text{commands}
\end{array}
$$

Co-pattern rules:

$$\frac{}{\alpha : X \Leftarrow (\vdash \alpha : X)}$$

$$\frac{q : A_1 \Leftarrow (\Gamma \vdash \Delta)}{\mathsf{fst}[q] : A_1 \,\&\, A_2 \Leftarrow (\Gamma \vdash \Delta)} \qquad \frac{q : A_2 \Leftarrow (\Gamma \vdash \Delta)}{\mathsf{snd}[q] : A_1 \,\&\, A_2 \Leftarrow (\Gamma \vdash \Delta)}$$

$$\frac{q_1 : A_1 \Leftarrow (\Gamma_1 \vdash \Delta_1) \quad q_2 : A_2 \Leftarrow (\Gamma_2 \vdash \Delta_2)}{[q_1, q_2] : A_1 \parr A_2 \Leftarrow (\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2)}$$

$$\frac{}{\mathsf{not}^{-}[x] : \neg^{-} A \Leftarrow (x : A \vdash)}$$

$$\ldots$$

Substitution rules:

$$\frac{}{(\vdash)\epsilon : (\Gamma \vdash \Delta)} \qquad \frac{(\Gamma' \Rightarrow \Delta')\sigma : (\Gamma \vdash \alpha : X, \Delta)}{(\Gamma' \Rightarrow \alpha' : X, \Delta')\sigma\,\{\alpha/\alpha'\} : (\Gamma \vdash \alpha : X, \Delta)} \qquad \frac{\Gamma \vdash V : A \mid \Delta \quad (\Gamma' \Rightarrow \Delta')\sigma : (\Gamma \vdash \Delta)}{(\Gamma', x : A \Rightarrow \Delta')\sigma\,\{V/x\} : (\Gamma \vdash \Delta)}$$

Value and co-value rules:

$$\frac{q : A \Leftarrow (\Gamma' \vdash \Delta') \quad (\Gamma' \vdash \Delta')\sigma : (\Gamma \vdash \Delta)}{\Gamma; q\sigma : A \vdash \Delta} \qquad \frac{\forall q : (A \Leftarrow (\Gamma' \vdash \Delta')).\psi(q) : (\Gamma, \Gamma' \vdash \Delta, \Delta')}{\Gamma | \mathbf{case}\,(\psi) : A \vdash \Delta}$$

Cut rules:

$$\frac{\Gamma, x : A; E : A \vdash \Delta}{\langle x \| E \rangle : (\Gamma, x : A \vdash \Delta)} \qquad \frac{\Gamma \vdash V : A \mid \Delta \quad \Gamma; E : A \vdash \Delta}{\langle V \| E \rangle : (\Gamma \vdash \Delta)}$$

Figure 26. Syntax and types of **CU** (the negative fragment).

$$
\begin{array}{ll}
(\beta^{+}) & \langle p\sigma \| \mathbf{case}\,[\phi] \rangle \to (\phi(p))\sigma \\
(\beta^{-}) & \langle \mathbf{case}\,(\psi) \| q\sigma \rangle \to (\psi(q))\sigma
\end{array}
$$

Figure 27. Reduction in **CU**.

ues and co-values are given once and for all, we only have two rules to consider: the positive rule for pattern matching on direct values and the negative rule for pattern matching on direct co-values. These reductions may be understood by analogy to the reductions of system L. For instance, suppose we have the reduction:

$$\langle \mathsf{inl}(9) \| \mathbf{case}\, [\mathsf{inl}(x) \Rightarrow S_1 | \mathsf{inr}(y) \Rightarrow S_2] \rangle \to S_1\, \{9/x\}$$

Recall that the value $\mathsf{inl}(9)$ may be seen as substituting 9 for x in the pattern $\mathsf{inl}(x)$. Therefore, the above reduction may also be written as:

$$\langle \mathsf{inl}(x)\, \{9/x\} \| \mathbf{case}\, [\mathsf{inl}(x) \Rightarrow S_1 | \mathsf{inr}(y) \Rightarrow S_2] \rangle \to S_1\, \{9/x\}$$

Unpacking the steps of this rule, we see that we match the pattern $\mathsf{inl}(x)$ in order to look up the correct command described by the case abstraction. Next, we select the correct command, and then perform the same substitution over the command as we used to obtain our original value $\mathsf{inl}(9)$. This process is generalized for deep patterns in $\mathbf{CU}$. The rule

$$\langle p\sigma \| \mathbf{case}\, [\phi] \rangle \to (\phi(p))\sigma$$

is implemented by (1) looking up the correct command described by the case analysis (written $\phi(p)$), and (2) actually performing the substitution to fill in the specifics of the value in the resulting command (written $S\sigma$).

Remark 9. Note that, oddly enough, stable judgments are not stable under substitution of positive types for type variables X. For instance, if we have the stable judgment $\Gamma, X \vdash \Delta$, and we find out that X actually stands in for $X_1 \otimes X_2$. By substitution, we get the judgment $\Gamma, X_1 \otimes X_2 \vdash \Delta$, which is no longer stable since we can apply an additional reversible $\otimes L$ rule before ending the asynchronous phase. In order to regain stability under substitution, we would need to relax our mandate and allow for more *complex hypothesis* [117] like $X_1 \otimes X_2$, therefore permitting the asynchronous phase to be ended early. This relaxing of the restriction corresponds to *complex variables* that choose to give a name to a value of positive type without decomposing its structure, ending pattern matching early. Allowing for complex variables gives us a calculus that is more like system L extended with a notion of deep (but not mandatory) pattern matching, except without general output abstractions for positive types and without input abstractions for negative types. *End remark* 9.

6.5 Combining polarities

Now that we have both the positive and negative fragments of logic and programming, how do we combine them into a single, unified languages? One of the simplest things we could do is to just take the disjoint union of the two. That way, some type A is either a type from the positive fragment, A^+, or from the negative fragment, A^-, and similarly for terms and co-terms. However, such a system would be unusually restrictive compared to functional programming languages. In a functional language, it is typical to store a function (which is a negative term) inside of a data structure (which is a positive term). However, this would not be allowed by a simple disjoint union between the two fragments: a positive term like a data structure is positive all the way down. Instead, we would like to allow for some mingling between positive and negative types and positive and negative (co-)terms.

In order to be explicit about the presence of negative terms in positive structures, and vice versa, by using Girard's [54] "shift" connectives to identify a switch between the positive and negative polarities. Therefore, in addition to putting together the positive and negative fragments of system L side-by-side, we also extend them so they may reference each other. On the one hand, in the positive fragment, we add the additional type $\downarrow A^-$, so that for every negative type A^- we have the positive type $\downarrow A^-$. Going along with our story that positive values are structures, we likewise add the structured term $\downarrow(M^-)$ of type $\downarrow A^-$ that contains a negative term along with a co-term, $\mathbf{case}\, [\downarrow(x) \Rightarrow S]$, for unpacking the structure and pulling out the underlying term. On the other hand, in the negative fragment, we add the additional type $\uparrow A^+$, so that for every positive type A^+ we have the negative type $\uparrow A^+$. In order to remain symmetric, we add the abstract term $\mathbf{case}\, (\uparrow[\alpha] \Leftarrow S)$ which is waiting for a structured message $\uparrow[K^+]$ of type $\uparrow A^+$ containing a positive co-term. The syntax and types for shifts in focalized system L is given in Figure 28.

Additionally, we can explain how to evaluate a program containing shifts by extending the reduction rules of system L to include β rules for the new types. The β rules for the shift connectives follow the same general pattern as all the others: use case analysis to examine the structure of (co-)values and substitute the underlying values for variables in the pattern. The rule for case analysis on a shift from negative to positive is

$$\langle \downarrow(M^-) \| \mathbf{case}\, [\downarrow(x) \Rightarrow S] \rangle \to S\left\{M^-/x\right\}$$

and the rule for case analysis on a shift from positive to negative is

$$\langle \mathbf{case}\, (\uparrow[\alpha] \Leftarrow S) \| \uparrow[K^+] \rangle \to S\left\{K^+/\alpha\right\}$$

Notice two things about these rules: (1) they resemble the call-by-name $\tilde{\mu}$ rule and the call-by-value μ rule, except wrapped up in a case analysis that unpacks a singleton structure, and (2) they imply that a negative term is a value in the sense of call-by-value, and a positive co-term is a co-value in the sense of call-by-name. Therefore, another way of interpreting the shift connectives in the polarized setting is that they allow for embedding a call-by-name term in a call-by-value structure and embedding a call-by-value co-term in a call-by-name co-structure.

We can also extend the pattern calculus $\mathbf{CU}$ with shift connectives in a similar fashion. Since neither the values, co-values, nor the reduction rules mention any connective in particular, all we need to do is extend the patterns and co-patterns to include the appropriate structures, as shown in Figure 29. On the one hand, the pattern for the positive shift outlines a structure containing a negative component called x:

$$\overline{\downarrow(x) : \downarrow A^- \Rightarrow (x : A^- \vdash)}$$

On the other hand, the pattern for the negative shift outlines a co-structure containing a positive component called α:

$$\overline{\uparrow[\alpha] : \uparrow A^+ \Leftarrow (\vdash \alpha : A^+)}$$

By adding these two patterns for shifts, the rest of the rules in $\mathbf{CU}$ for forming values, co-values, and commands are automatically extended to accommodate the new types.

Remark 10. Now that we can talk about both polarities in the same setting, we are able to consider other connectives, besides the shifts, that mix polarities. For example, let's consider functions in a polarized and focalized language. There are different interpretations of functions according to different evaluation strategies — call-by-name, call-by-value, and call-by-need. However, because we are using types to determine evaluation order, each of these different interpretations gives rise to a different function type. In order to cut to the heart of the meaning of functions in polarized logic, we should look for the "ideal" function type, the most "primordial" [117] one from which all the others can be derived.

Recall that the function type is negative, since its right rule is reversible, and so programs of the function type are defined by the pragmatist in terms of the structure of a function call (the co-term of function type). Additionally, Section 6.4 showed that trying to carry out a proof search phase for as long as possible encouraged us to pattern match as deeply as we could into a structure. Therefore, it is

$$A^+ \in Type^+ ::= \ldots \mid \downarrow A^- \qquad A^- \in Type^- ::= \ldots \mid \uparrow A^+$$
$$V^+ \in Value^+ ::= \ldots \mid \downarrow(M^-) \qquad V^- \in Value^- ::= \ldots \mid \mathbf{case}\,(\uparrow[\alpha] \Leftarrow S)$$
$$E^+ \in CoValue^+ ::= \ldots \mid \mathbf{case}\,[\downarrow(x) \Rightarrow S] \qquad E^- \in CoValue^- ::= \ldots \mid \uparrow[K^+]$$

$$\frac{\Gamma \vdash M : A^- | \Delta}{\Gamma \vdash \downarrow(M^-) : \downarrow A^-; \Delta}\ \downarrow R \qquad \frac{\Gamma | K^+ : A^+ \vdash \Delta}{\Gamma; \uparrow[K^+] : \uparrow A^+ \vdash \Delta}\ \uparrow L$$

$$\frac{S : (\Gamma, x : A^- \vdash \Delta)}{\Gamma | \mathbf{case}\,[\downarrow(x) \Rightarrow S] : \downarrow A^- \vdash \Delta}\ \downarrow L \qquad \frac{S : (\Gamma \vdash \alpha : A^+, \Delta)}{\Gamma \vdash \mathbf{case}\,(\uparrow[\alpha] \Leftarrow S) : \uparrow A^+ | \Delta}\ \uparrow R$$

$$(\beta^{\downarrow}) \qquad \langle \downarrow(M^-) \| \mathbf{case}\,[\downarrow(x) \Rightarrow S] \rangle \to S\left\{M^-/x\right\}$$
$$(\beta^{\uparrow}) \qquad \langle \mathbf{case}\,(\uparrow[\alpha] \Leftarrow S) \| \uparrow[K^+] \rangle \to S\left\{K^+/\alpha\right\}$$

Figure 28. Explicit polarity shifts in focalized system L.

$$\overline{\downarrow(x) : \downarrow A^- \Rightarrow (x : A^- \vdash)} \qquad \overline{\uparrow[\alpha] : \uparrow A^+ \Leftarrow (\vdash \alpha : A^+)}$$

Figure 29. Polarity shifts in patterns and co-patterns.

in our best interest to define the function type so that it allows us to peer deeply into the structure of a function call. This corresponds to keeping the sub-components of a function call in focus. That leads us to the following left rule for functions:

$$\frac{\Gamma \vdash V : A^+; \Delta \quad \Gamma; E : B^- \vdash \Delta}{\Gamma; V \cdot E : A^+ \to B^- \vdash \Delta} \to L$$

Given a function call, $V \cdot E$, to maintain focus on the argument V it must have a positive type and to maintain focus on E it must have a negative type, based on our judgments for focusing in logic. The terms of function type may be given in exactly the same manner as all the other negative connectives

$$\frac{S : (\Gamma, x : A^+ \vdash \beta : B^-, \Delta)}{\Gamma \vdash \mathbf{case}\,(x \cdot \beta \Leftarrow S) : A^+ \to B^- | \Delta} \to R$$

or in terms of the more familiar λ-abstraction notation from the λ-calculus

$$\frac{\Gamma, x : A^+ \vdash M : B^- | \Delta}{\Gamma \vdash \lambda x.M : A^+ \to B^- | \Delta} \to R$$

The two notations for functions are equivalent to one another. Additionally, we may give a rule for co-patterns of function type

$$\frac{p : A^+ \Rightarrow (\Gamma \vdash \Delta) \quad q : B^- \Leftarrow (\Gamma' \vdash \Delta')}{p \cdot q : A^+ \to B^- \Leftarrow (\Gamma, \Gamma' \vdash \Delta, \Delta')}$$

which reveals the impact of the choice of polarities for the argument and result of a function. If we were to make the argument to a function negative, then we would no longer be able to form a pattern for it, thus forcing the an end to the first premise, and similarly if the result was positive.

However, it is easy to halt a pattern by inserting a shift which signals the end of a phase. Therefore, we may recover other choices of polarities for function types by inserting shifts where appropriate. For example, the wholly negative function type is given by shifting the argument, $\downarrow(A^-) \to B^-$, which signals in the type that we cannot directly examine the structure of the argument. Additionally, the whole positive function type is given by shifting both the result of the function call as well as the function itself, $\downarrow(A^+ \to \uparrow(B^+))$, which signals that we cannot directly examine the use of the result or on a function value (that is, both the continuation that accepts the result and the function itself are "black boxes").

Following this exploration of function types in focalized, polarized logic, it is worthwhile to revisit our definitions of negation. Notice that in the positive and negative fragments of **CU**, from Figures 25 and 26, the (co-)patterns for the two forms of negation end abruptly. This is because they are defined all within the same polarity: the positive negation $\mathsf{not}(\alpha)$ contains a positive co-term, which is a black box that we cannot directly examine, whereas the negative negation (pardon the apparent redundancy of terminology) contains a negative term, which is also a black box. If we want to continue forming a pattern through a negation, we are forced to switch polarity as well. This observation suggests the existence of *yet another* two dual forms of negation that are more "ideal" and more "primordial" than the ones we previously considered. In essence, by swapping both the polarity and the sides of a command, these notions of negation internalize the duality of the sequent calculus and correspond to the notion of involutive negation from linear logic.

The negative form of the dual negation, which we'll write as $(A^+)^{\perp}$ in the spirit of linear logic, is similar to $\neg^- A^-$ except for the polarity of the underlying type. Instead of turning a negative term into another negative co-value, $(A^+)^{\perp}$ turns a positive value into a negative co-value, allowing us to keep the sub-term in focus:

$$\frac{\Gamma \vdash V : A^+; \Delta}{\Gamma; [V]^{\perp} : (A^+)^{\perp} \vdash \Delta}\ \_^{\perp} L$$

And the term for $(A^+)^{\perp}$ is similar to the one for $\neg^- A^-$:

$$\frac{S : (\Gamma, x : A^+ \vdash \Delta)}{\Gamma \vdash \mathbf{case}\,([x]^{\perp} \Rightarrow S) : (A^+)^{\perp} | \Delta}\ \_^{\perp} R$$

Likewise, the positive form of dual negation, which we'll write as $(A^-)_{\top}$, is similar to $\neg^+ A^+$ except for the polarity of the underlying type. Instead of turning a positive co-term into a positive value, $(A^-)_{\top}$ turns a negative co-value into a positive value, maintaining

focus on the sub-co-term:

$$\frac{\Gamma; E : A^- \vdash \Delta}{\Gamma \vdash (E)_\top : (A^-)_\top; \Delta}\ {}_-{}_\top R$$

And by this point, there are no surprises in the co-term for $(A^+)^\perp$:

$$\frac{S : (\Gamma \vdash \alpha : A^-, \Delta)}{\Gamma | \mathbf{case}\,[(\alpha)_\top \Rightarrow S] : (A^-)_\top \vdash \Delta}\ {}_-{}_\top L$$

Additionally, the patterns and co-patterns for the positive and negative forms of dual negation are given as

$$\frac{p : A^+ \Rightarrow (\Gamma \vdash \Delta)}{[p]^\perp : (A^+)^\perp \Leftarrow (\Gamma \vdash \Delta)} \qquad \frac{q : A^- \Leftarrow (\Gamma \vdash \Delta)}{(q)_\top : (A^-)_\top \Rightarrow (\Gamma \vdash \Delta)}$$

where unlike $\neg^+ A^+$ and $\neg^- A^-$, we are able to continue forming a pattern for the underlying component. To recover the previous forms of negation, we can manually signal an end to the (co-)patterns by inserting an explicit shift, so that $\neg^-(A^-)$ is isomorphic to $(\downarrow A^-)^\perp$ and $\neg^+(A^+)$ is isomorphic to $(\uparrow A^+)_\top$.

Furthermore, the fact that involutive negation in linear logic is actually involutive, so that $(A^\perp)^\perp$ is exactly the same as A, is reflected in the forms of dual negation, although in a slightly weaker sense. In essence, the type $((A^+)^\perp)_\top$ is isomorphic to A^+, so that an arbitrary value of type $((A^+)^\perp)_\top$ has the form $([V]^\perp)_\top$ which is isomorphic to V. Intuitively, any direct examination of the structure of V can also be done to $([V]^\perp)_\top$ by just peeling off the outer $([\_]^\perp)_\top$ (and similarly for $[(E)_\top]^\perp$). In other words, the extended pattern $([p]^\perp)_\top$ neither adds nor hides any information about the underlying pattern p. *End remark* 10.

Remark 11. In order to relate polarized languages to functional programming languages, we need to account for the evaluation strategy of the particular functional language by using polarities in the polarized types. This means that since ML is a strict language whereas Haskell is a lazy language, a program that looks the same in both ML and Haskell will actually correspond to completely different programs in a polarized language with completely different types. In essence, a translation into polarized logic reveals the evaluation strategy of a language in both the type and the syntax of the program, similar to a *continuation-passing style* (CPS) transform. Additionally, polarized logic may be viewed as a sort of type and effect system [80], in which evaluation is considered an effect that is static tracked along with types.

For example, the correspondence between polarities, in the form of the $\mathbf{CU}$ calculus, and parts of ML are illustrated in Figure 30. In particular, the data types of ML, like the pair type $'a *' b$ and sum type $'a +' b$, correspond to positive types. Additionally, ML functions correspond to a wholly positive function type, which can be encoded in terms of the primordial function type mentioned in Remark 10, and are isomorphic to a similar encoding in terms of conjunction and negation as Wadler's call-by-value encoding of functions from Section 5.1.

One very important detail to note is that in $\mathbf{CU}$, the type of ML terms is *different* than the type of ML values. In particular, all the terms in ML are shifted. The extra shift arises because in $\mathbf{CU}$, the terms of positive type are values that contain no (direct) computation — they just *are*. Therefore, in order to introduce any ability to represent computation in a term of positive type in $\mathbf{CU}$, it needs to be shifted to the negative polarity. This extra shift opens the door to modeling behavior by making the term into a case abstraction — a positive pair value $(1, 2) : \mathsf{Int} \otimes \mathsf{Int}$ cannot *do* anything, but the negative value $\mathbf{case}\,(\uparrow[\alpha] \Leftarrow S) : \uparrow(\mathsf{Int} \otimes \mathsf{Int})$ can. A pair of two non-value terms in ML, $(M_1, M_2) :' a *' b$, would have to be translated to the polarized calculus in CPS style $\mathbf{case}\,(\uparrow[\alpha] \Leftarrow \langle M_1 \| \uparrow[\mathbf{case}\,[x \Rightarrow \langle M_2 \| \uparrow[\mathbf{case}\,[y \Rightarrow \langle (x, y) \| \alpha \rangle]]\rangle]]\rangle)$ where we use complex variables (see Remark 9) to avoid writing down how to break down and reconstruct the values returned by M_1 and M_2.[10] Additionally, notice that to model an ML computation that produces a function we need to insert even more shifts, making $\uparrow(\downarrow(A^+ \to \uparrow B^+))$ the type of function computations. The extra shift surrounding the type is necessary for modeling computations that cause a side effect before returning a function, like $(\mathsf{print}\ \texttt{"hi"}; (\lambda x.x))$, or computations that never return. Notice that the shifted value $\mathbf{case}\,(\uparrow[\alpha] \Leftarrow S)$ bears resemblance to an output abstraction $\mathbf{out}\,\alpha \Leftarrow S$ of type $\mathsf{Int} \otimes \mathsf{Int}$. In effect, the shift makes up for the fact that, unlike system L, $\mathbf{CU}$ does not contain output abstractions for positive types. Therefore, the extra input and output abstractions in system L (as well as the non-polarized languages in Section 5), allow for a more direct style representation of programs.

Additionally, Figure 30 shows the relationship of Haskell with $\mathbf{CU}$. Of note, the correspondence between $\mathbf{CU}$ and a lazy functional language is much less direct than it was for a strict functional language. This is because the data types of Haskell to not map so nicely onto the positive connectives in polarized logic due to extra laziness. For instance, when performing case analysis on a sum type $\mathsf{Either}\ a\ b$ in Haskell, we stop evaluation once we reach the distinguishing tag, instead of continuing on to evaluate the subterm. The fact that we stop evaluation once reaching a constructor is revealed as the inner shifts in the table. On the other hand, an unevaluated term of the Haskell $\mathsf{Either}\ a\ b$ is also passed around like a value (ignoring the sharing and memoization performed by many implementations), which introduces the outer shifts. Dually to ML, Haskell functions correspond to a wholly negative function type, which can be encoded in terms of the primordial function type mentioned in Remark 10, and are isomorphic to a similar encoding in terms of disjunction and negation as Wadler's call-by-name encoding of functions from Section 5.1.[11] On the nicer side, there is no split in the distinction between terms and values as there was for ML, every term in Haskell is given the same type without introducing an extra type-level notion of "valueness."

It is worthwhile to note that some accounts [116] of polarized logic suggest that pairs in Haskell correspond to the $\&$ connective, the negative form of conjunction. However, this characterization of Haskell pairs is not correct — as with Haskell sums and ML pairs, Haskell pairs are defined by their constructors and observed by analyzing the structure of a pair. For example, if we let Ω be a nonterminating program, then if Haskell pairs were modeled by $\&$ the terms Ω and $(\mathsf{fst}(\Omega), \mathsf{snd}(\Omega))$ should be indistinguishable. However, the context $\mathbf{case}\,\Box\,\mathbf{of}\,(x, y) \Rightarrow \mathsf{True}$ observes a difference between the two: plugging in $(\mathsf{fst}(\Omega), \mathsf{snd}(\Omega))$ causes the program to return True and plugging in Ω causes the program to loop forever. Therefore, pairs are better modeled in terms of the positive form of conjunction, in the same vein as the fact that the data type for Haskell sums are better modeled with $\oplus$ than with $⅋$. And in general, all of Haskell data types are modeled in terms of positive connectives (and shifts) in polarized logic. *End remark* 11.

[10] Note that the fact that without adding complex variables to $\mathbf{CU}$, we would be forced to pattern match on the structures of x and y means that we would have to write a different program for sequencing the elements for every pair type. This means that without the help of complex variables, the $\mathbf{CU}$ calculus cannot be polymorphic over structures.

[11] This model of Haskell functions does not account for memoization, as performed by every major implementation of Haskell, and does not hold up in the presence of the primitive *seq* operation which can distinguish a λ-abstraction from any other term of function type.

ML	Polarities
$V :' a$	$V : A^+$
$M :' a$	$M : \uparrow(A^+)$
$'a *' b$	$A^+ \otimes B^+$
$'a +' b$	$A^+ \oplus B^+$
$'a \to' b$	$\downarrow(A^+ \to \uparrow(B^+))$ $\approx \neg^+(A^+ \otimes \neg^+ B^+)$

Haskell	Polarities
$M : a$	$M : A^-$
(a, b)	$\uparrow(\downarrow(A^-) \otimes \downarrow(B^-))$
Either a b	$\uparrow(\downarrow(A^-) \oplus \downarrow(B^-))$
$a \to b$	$\downarrow(A^-) \to B^-$ $\approx \neg^-(A^-) \parr B^-$

Figure 30. Rosetta stone for relating strict (ML) and lazy (Haskell) functional languages to the polarized $\mathbf{CU}$.

7. A retrospective

By now, we have covered many different systems that give various approaches to the computational interpretation of the sequent calculus and the computational perspective of duality. The concept of duality comes up when considering the relationship between producers and consumers of information, the call-by-value and call-by-name evaluations strategies, and structures and destructive case analysis. We also saw how concepts that arose in the field of proof search, namely focalization and polarization, are connected to the concept of pattern matching on structures in functional programming languages.

However, there are still some questions that have not been addressed. For instance, how do other evaluation strategies, like call-by-need [6, 9, 81][12] fit into the picture? If we follow the story of polarized logic, that the polarity determines evaluation order, then there is no room — by definition there are only two polarities so we can only directly account for two evaluation strategies with this approach.

Instead, let's take a step back and look at the bigger picture, letting go of the particular details of each of these systems in order to bring out the general themes hiding under the surface of each of them. Since our primary goal is to understand the behavior of programs, we will play down the emphasis on types from perspective of the proof theoretic background in favor of considering run-time behavior first, in terms of an *equational theory* that determines when two programs (commands, terms, or co-terms) behave the same as one another. This is a shift from an *intrinsic* [98] view of types (types define the meaning of programs) to an *extrinsic* one — a divide that goes back to Church and Curry. In particular, we will consider the sequent calculus with an approach more similar to ones taken for the λ-calculus, and see what it can tell us about how we write programs.[13]

7.1 A parametric theory of the sequent calculus

Notice that, with the exception of the $\mathbf{CU}$ calculus from Section 6.4, every language for the sequent calculus in Sections 4, 5, and 6 are based on the same structural core. This core, given in Figure 8, forms the basis of naming in the sequent calculus in terms of variables and co-variables as well as input and output abstractions. Further still, the fundamental dilemma of computation in classical sequent calculus lies wholly within this core. The root cause of non-determinism is a conflict between the input and output abstractions, where each one tries to take control over the future path of evaluation. Therefore, we will first focus on how to resolve the fundamental dilemma in the structural core of the sequent calculus before tackling any other issues.

[12] Call-by-need can be thought of as a memoizing version of call-by-name where the arguments to function calls are performed on demand, like in call-by-name, but where the value of an argument is remembered so that it is computed only once, like in call-by-value.

[13] This following discussion is based on work to appear in ESOP 2014 [32].

Recall that the source of the conflict in the structural core of the sequent calculus are the two rules for implementing substitution:

$$(\mu) \qquad \langle \mathbf{out}\, \alpha \Leftarrow S \| K \rangle = S\,\{K/\alpha\}$$
$$(\tilde{\mu}) \qquad \langle M \| \mathbf{in}\, x \Rightarrow S \rangle = S\,\{M/x\}$$

As stated, a command like $\langle \mathbf{out}\, \_ \Leftarrow S_1 \| \mathbf{in}\, \_ \Rightarrow S_2 \rangle$, where the (co-)variables are never used, is equal to both S_1 and S_2, so any two arbitrary commands may be considered equal. All of the various languages solve this issue by restricting one of the two rules to remove the conflict — languages that restrict the μ rule implement a form of call-by-name evaluation and languages that restrict the $\tilde{\mu}$ rule implement a form of call-by-value evaluation. However, in lieu of inventing various different languages with different evaluation strategies, let's instead allow for restriction to what *both* rules are capable of substituting

$$(\mu_E) \qquad \langle \mathbf{out}\, \alpha \Leftarrow S \| E \rangle = S\,\{E/\alpha\}$$
$$(\tilde{\mu}_V) \qquad \langle V \| \mathbf{in}\, x \Rightarrow S \rangle = S\,\{V/x\}$$

and leaving specifics of what stands in for V and E open for interpretation. That is to say, we make the sets of values (V) and co-values (E) a *parameter* of the theory, in the same sense as the parametric λ-calculus [99], that may be filled in at a later time. The full parametric equational theory for the structural core is given in Figure 31. Since the rules for extensionality of input and output abstractions did not cause any issue, we leave them alone.

By leaving the choice of restrictions open as parameters, the *same* parametric theory may describe different evaluation strategies by instantiating the parameters in different ways. For this reason, we say that a *strategy* for the parametric theory is defined by a choice of values and co-values. The previous characterizations of call-by-value and call-by-name come out as particular instances of the parametric theory. For example, we can define the call-by-value strategy $\mathcal{V}$, shown in Figure 32, by restricting the set of values to not include output abstractions, leaving variables as the only value, and letting every co-term be a co-value. Notice that in effect, this decision restricts the $\tilde{\mu}$ rule in the usual way for call-by-value while letting the μ rule be unrestricted. The call-by-name strategy $\mathcal{N}$ is defined in the dual way by letting every term be a value and restricting the set of co-values to not include input abstractions, leaving co-variables as the only co-value.

We can also explore other choices of the parameters that describe strategies other than just call-by-value and call-by-name. For instance, we can characterize a notion of call-by-need in terms of a "lazy value" strategy $\mathcal{LV}$ shown in Figure 33. This strategy is similar to a previous call-by-need theory for the sequent calculus [13]. The intuition is similar to the call-by-need λ-calculus [9]: a non-value term bound to a variable represents a delayed computation that will only be evaluated when it is needed. Then, once the term has been reduced to a value (in the sense of call-by-value), it may be substituted for the variable. Therefore, in the command

$$\langle M_1 \| \mathbf{in}\, x \Rightarrow \langle M_2 \| \mathbf{in}\, y \Rightarrow S \rangle \rangle$$

$$(\mu_E) \quad \langle \mathbf{out}\,\alpha \Leftarrow S \| E\rangle = S\,\{E/\alpha\} \qquad (\eta_\mu) \quad \mathbf{out}\,\alpha \Leftarrow \langle M \| \alpha\rangle = M$$
$$(\tilde{\mu}_V) \quad \langle V \| \mathbf{in}\,x \Rightarrow S\rangle = S\,\{V/x\} \qquad (\eta_{\tilde{\mu}}) \quad \mathbf{in}\,x \Rightarrow \langle x \| K\rangle = K$$

Figure 31. A parametric theory for the structural core of the sequent calculus.

$$V \in Value_{\mathcal{V}} ::= x \qquad V \in Value_{\mathcal{N}} ::= M$$
$$E \in CoValue_{\mathcal{V}} ::= K \qquad E \in CoValue_{\mathcal{V}} ::= \alpha$$

Figure 32. Call-by-value ($\mathcal{V}$) and call-by-name ($\mathcal{N}$) strategies for the structural core.

$$V \in Value_{\mathcal{LV}} ::= x \qquad V \in Value_{\mathcal{LN}} ::= x \mid \mathbf{out}\,\alpha \Leftarrow C_{\mathcal{LN}}[\langle V \| \alpha\rangle]$$
$$E \in CoValue_{\mathcal{LV}} ::= \alpha \mid \mathbf{in}\,x \Rightarrow C_{\mathcal{LV}}[\langle x \| E\rangle] \qquad E \in CoValue_{\mathcal{LN}} ::= \alpha$$
$$C_{\mathcal{LV}} \in DelayedCxt ::= \Box \mid \langle M \| \mathbf{in}\,y \Rightarrow C_{\mathcal{LV}}\rangle \qquad C_{\mathcal{LN}} \in DelayedCxt_{\mathcal{LN}} ::= \Box \mid \langle \mathbf{out}\,\alpha \Leftarrow C_{\mathcal{LN}} \| K\rangle$$

Figure 33. "Lazy-call-by-value" ($\mathcal{LV}$) and "lazy-call-by-name" ($\mathcal{LN}$) strategies for the structural core.

we temporarily ignore M_1 and M_2 and work on the inner command S. If it turns out that S evaluates to $\langle x \| E\rangle$, we are left in the state

$$\langle M_1 \| \mathbf{in}\,x \Rightarrow \langle M_2 \| \mathbf{in}\,y \Rightarrow \langle x \| E\rangle\rangle\rangle$$

where E is a co-value that wants to know something about x, making $\mathbf{in}\,x \Rightarrow \langle M_2 \| \mathbf{in}\,y \Rightarrow \langle x \| E\rangle\rangle$ into a co-value as well. Therefore, if M_1 is a non-value output abstraction, it may take over by the μ_E rule.

Remark 12. Another way to think about strategies, and the parameterized notions of values and co-values, is to consider the parts of an equational theory. Typically, equational theories are expressed by a set of axioms (primitive equalities assumed to hold) along with some basic properties or rules for forming larger equations. By definition, equations are closed under reflexivity, symmetry, and transitivity. In other words, we know the following principles, expressed as inference rules, hold for equality in the sequent calculus

$$\frac{}{S = S}\ Refl \qquad \frac{S_2 = S_1}{S_1 = S_2}\ Sym \qquad \frac{S_1 = S_2 \quad S_2 = S_3}{S_1 = S_3}\ Trans$$

and similarly for terms and co-terms. We also consider equality in an expression language to apply anywhere in an expression — if two things are equal, then they must be equal in any context. This principle is called *congruence*, and may be expressed as

$$\frac{S = S'}{C[S] = C[S']}\ Congruence$$

and again similarly for (co-)terms.

Additionally, in a language with an internal notion of variables, like the λ-calculus or the structural core of the sequent calculus, we generally expect the equational theory to be closed under substitution. That is to say, if two things are equal, then they should still be equal when I substitute the same term for the same variable in both of them. However, this principle does not always hold in full generality. For example, the ML terms $\mathbf{let}\,y = x\,\mathbf{in}\,5$ and 5 are equal. However, if we substitute the term $(\texttt{print "hi"}; 1)$ for x, we end up with $\mathbf{let}\,y = (\texttt{print "hi"}; 1)\,\mathbf{in}\,5$ and 5, which are no longer equal because one produces a side effect and the other does not. Instead, ML has a restricted notion of the substitution principle: if two programs are equal, then they are still equal when we substitute the same *value* (an integer, a pair of values, a function abstraction, ...) for the same variable in both of them. This restriction prevents the previous counter-example. The exact same issue arises in the classical sequent calculus, since it also includes a notion of effects that allows for manipulation over the flow of control. Therefore, we need to restrict the substitution principle in the sequent calculus to only allow for substituting values for variables. Additionally, since we have a second notion of substitution, we also have a restriction to only allow for substituting co-values for co-variables. This leads us to substitution principles that say if two commands (or terms or co-terms) are equal, they must still be equal after substituting (co-)values for (co-)variables:

$$\frac{S = S'}{S\,\{V/x\} = S'\,\{V/x\}}\ Subst_V \qquad \frac{S = S'}{S\,\{E/\alpha\} = S'\,\{E/\alpha\}}\ Subst_E$$

We may also consider the axioms for the structural core of the sequent calculus to be trivial statements about variable binding. The η_μ and $\eta_{\tilde{\mu}}$ rules state that giving a name to something, and then using it immediately (without repetition) in the same place is the same thing as doing nothing. Additionally, we may say that binding a variable to itself is the same thing as doing nothing:

$$(\mu_\alpha) \qquad \langle \mathbf{out}\,\alpha \Leftarrow S \| \alpha\rangle = S$$
$$(\tilde{\mu}_x) \qquad \langle x \| \mathbf{in}\,x \Rightarrow S\rangle = S$$

If we take the substitution principles as reasoning steps in our equational theory, we can derive the axioms in Figure 31. The trick is to realize that a command like command $\langle V \| \mathbf{in}\,x \Rightarrow S\rangle$ is the image of $\langle x \| \mathbf{in}\,x \Rightarrow S\rangle$ under substitution of V for x. That is to say that $\langle V \| \mathbf{in}\,x \Rightarrow S\rangle$ is syntactically the same as $\langle x \| \mathbf{in}\,x \Rightarrow S\rangle\,\{V/x\}$. Therefore, we can derive the $\tilde{\mu}_V$ axiom from $\tilde{\mu}_x$ and $Subst_V$ as follows:

$$\frac{\dfrac{}{\langle x \| \mathbf{in}\,x \Rightarrow S\rangle = S}\ \tilde{\mu}_x}{\langle V \| \mathbf{in}\,x \Rightarrow S\rangle = S\,\{V/x\}}\ Subst_V$$

The derivation of μ_E from μ_α and $Subst_E$ is similar.

Additionally, the substitution principles are derivable from the more powerful $\tilde{\mu}_V$ and μ_E axioms. For example, we can derive $Subst_V$ from by recognizing that both sides of the equation can be deduced from a command like $\langle V \| \mathbf{in}\,x \Rightarrow S\rangle$ with the $\tilde{\mu}_V$ axiom, so that congruence allows us to lift the equality $S = S'$ under the bindings. The full derivation of $Subst_V$ is shown in Figure 34, and $Subst_E$ may be derived in a similar way. Therefore, the $\tilde{\mu}_V$ and μ_E rules may be seen as a realization of the substitution principles of an equational theory in the form of axioms. *End remark* 12.

$$\dfrac{\dfrac{\dfrac{}{\langle V \| \mathbf{in}\, x \Rightarrow S\rangle = S\,\{V/x\}}\,\tilde{\mu}_V}{S\,\{V/x\} = \langle V \| \mathbf{in}\, x \Rightarrow S\rangle}\,Sym \qquad \dfrac{\dfrac{S = S'}{\langle V \| \mathbf{in}\, x \Rightarrow S\rangle = \langle V \| \mathbf{in}\, x \Rightarrow S'\rangle}\,Congr. \qquad \dfrac{}{\langle V \| \mathbf{in}\, x \Rightarrow S'\rangle = S'\,\{V/x\}}\,\tilde{\mu}_V}{\langle V \| \mathbf{in}\, x \Rightarrow S\rangle = S'\,\{V/x\}}\,Trans}{S\,\{V/x\} = S'\,\{V/x\}}\,Trans$$

Figure 34. Deriving the substitution principle for values from the $\tilde{\mu}_V$ axiom.

7.2 User-defined data and co-data types

Having considered the structural core of the sequent calculus, and resolving the fundamental dilemma in a general way by making the strategy a parameter to the equational theory, we now move on to add connectives to the calculus. Instead of looking at each connective individually, and determining its properties case-by-case, we will again try to remain general: we will summarize all of the previously discussed connectives under the umbrella of user-defined types. In deciding how we may add new types to the language, we will take the verificationist and pragmatist approaches from Section 6 as part of the type declaration. On the one hand, the verificationist approach gives us a notion of user-defined data types similar to algebraic data types in functional programming languages. On the other hand, the pragmatist approach gives us the dual notion of user-defined *co-data types*, which are instead similar to interfaces in object-oriented languages. In lieu of spelling out the full generality of user-defined types, we will work by examples.

As a starting point, we will base the syntax of user-defined (co-)data type declarations in the sequent calculus on data type declaration in functional languages. However, in order to provide a syntax for (co-)data type declaration that covers every single connective we have considered so far, we need a syntax that is more general than the usual form of algebraic data type (ADT) declaration from ML-based languages. Therefore, we will consider how the generalized syntax for GADTs in Haskell [71, 104] may be used for ordinary data type declarations. For example, the typical sum type Either and pair type Both may be declared as:

$$\begin{aligned}&\mathbf{data}\,\mathsf{Either}\;a\;b\,\mathbf{where}\\ &\qquad \mathsf{Left} : a \to \mathsf{Either}\;a\;b\\ &\qquad \mathsf{Right} : b \to \mathsf{Either}\;a\;b\end{aligned}$$

$$\begin{aligned}&\mathbf{data}\,\mathsf{Both}\;a\;b\,\mathbf{where}\\ &\qquad \mathsf{Pair} : a \to b \to \mathsf{Both}\;a\;b\end{aligned}$$

In the declaration for Either, we specify that there are two constructors, Left and Right, that take a value of type a and b, respectively, and build a value of type Either $a\;b$. In the declaration for Both, we specify that there is one constructor, Pair, that takes a value of type a, a value of type b, and builds a value of type Both $a\;b$.

When declaring a new type in the sequent calculus, we will take the general form of a GADT, but instead describe the constructors by a sequent-style judgment. For connectives following the verificationist approach, we have data type declarations that introduce new concrete terms and abstract co-terms. For instance, we can give a declaration of $A \oplus B$ as

$$\begin{aligned}&\mathbf{data}\;A \oplus B\,\mathbf{where}\\ &\qquad \mathsf{inl} : A \vdash A \oplus B|\\ &\qquad \mathsf{inr} : B \vdash A \oplus B|\end{aligned}$$

where we mark the distinguished output of the constructor as $A \oplus B|$ and replace the function arrow ($\to$) with logical entailment ($\vdash$), to emphasize that the function type is just another user-defined type like all the others. This declaration extends the syntax of the language with two new concrete terms for the constructors, $\mathsf{inl}(M)$ and $\mathsf{inr}(M)$, and with one new abstract co-term for case analysis, $\mathbf{case}\,[\mathsf{inl}(x) \Rightarrow S_1 | \mathsf{inr}(y) \Rightarrow S_2]$. Additionally, we can declare $A \otimes B$ as

$$\begin{aligned}&\mathbf{data}\;A \otimes B\,\mathbf{where}\\ &\qquad \mathsf{pair} : A, B \vdash A \otimes B|\end{aligned}$$

where the multiple inputs to the constructor are given as a list of inputs to the left of the sequent, as opposed to the "curried" style used in the declaration of Both. This declaration extends the syntax of the language with one new concrete term for the constructor, $\mathsf{pair}(M, M')$, and one new abstract co-term for case analysis, $\mathbf{case}\,[\mathsf{pair}(x, y) \Rightarrow S]$.

However, note that the notion of user-defined types in the sequent calculus is more general than in functional programming languages. We also have co-data declarations that introduce abstract terms and concrete co-terms. A co-data declaration may be thought of as an interface that describes the allowable messages we may send to an abstract value. By analogy to object-oriented programming, an interface (co-data type declaration) describes the fixed set of methods (co-structures) that an object (co-case abstraction) has to support (provide cases for), and the object (co-case abstraction) defines the behavior that results from a method call (command). For example, we can declare $A \,\&\, B$ as

$$\begin{aligned}&\mathbf{codata}\;A \,\&\, B\,\mathbf{where}\\ &\qquad \mathsf{fst} : |A \,\&\, B \vdash A\\ &\qquad \mathsf{snd} : |A \,\&\, B \vdash B\end{aligned}$$

which extends the language with a new abstract term for case analysis, $\mathbf{case}\,(\mathsf{fst}[\alpha] \Leftarrow S_1 | \mathsf{snd}[\beta] \Leftarrow S_2)$, and two concrete co-terms, $\mathsf{fst}[K]$ and $\mathsf{snd}[K]$. We can also declare the function type as an instance

$$\begin{aligned}&\mathbf{codata}\;A \to B\,\mathbf{where}\\ &\qquad \mathsf{call} : A | A \to B \vdash B\end{aligned}$$

which, following the pattern by rote, extends the language with a new abstract term, $\mathbf{case}\,(\mathsf{call}[x, \alpha] \Leftarrow S)$, and a new concrete co-term, $\mathsf{call}[M, K]$. The rest of the basic connectives are declared as user-defined (co-)data types in Figure 35.

Having allowed for extending the language with an ample variety of new syntactic forms, we now need to explain how they behave. In addition, since the evaluation strategy is given as a parameter to the equational theory of the structural core, we should express the behavior in some way that is valid in *any* choice of strategy. To accomplish our goal, we will use β and η axioms for defining the dynamic meaning of user-defined (co-)data types. For example, we may extend the equational theory with the following β axiom for functions

$$(\beta^{\to}) \qquad \begin{aligned}&\langle \mathbf{case}\,(\mathsf{call}[x, \alpha] \Leftarrow S) \| \mathsf{call}[M, K]\rangle\\ &\quad = \langle M \| \mathbf{in}\, x \Rightarrow \langle \mathbf{out}\;\alpha \Leftarrow S \| K\rangle\rangle\end{aligned}$$

which matches on the structure of function application and binds the sub-components to the appropriate (co-)variables. The thing to notice is that this rule applies for *any* function call $\mathsf{call}[M, K]$, whether or not M or K are (co-)values. This works because M and K are put in interaction with input and output abstractions, and we

$\mathbf{data}\, A \oplus B\, \mathbf{where}$	$\mathbf{codata}\, A \,\&\, B\, \mathbf{where}$
$\quad \mathsf{inl} : A \vdash A \oplus B \mid$	$\quad \mathsf{fst} : \mid A \,\&\, B \vdash A$
$\quad \mathsf{inr} : B \vdash A \oplus B \mid$	$\quad \mathsf{snd} : \mid A \,\&\, B \vdash B$
$\mathbf{data}\, A \otimes B\, \mathbf{where}$	$\mathbf{codata}\, A \parr B\, \mathbf{where}$
$\quad \mathsf{pair} : A, B \vdash A \otimes B \mid$	$\quad \mathsf{split} : \mid A \parr B \vdash A, B$
$\mathbf{data}\, 1\, \mathbf{where}$	$\mathbf{codata}\, \bot\, \mathbf{where}$
$\quad \mathsf{unit} : \vdash 1 \mid$	$\quad \mathsf{tp} : \mid \bot \vdash$
$\mathbf{data}\, 0\, \mathbf{where}$	$\mathbf{codata}\, \top\, \mathbf{where}$
$\mathbf{data}\, A - B\, \mathbf{where}$	$\mathbf{codata}\, A \to B\, \mathbf{where}$
$\quad \mathsf{uncall} : A \vdash A - B \mid B$	$\quad \mathsf{call} : A \mid A \to B \vdash B$
$\mathbf{data}\, \neg^{+} A\, \mathbf{where}$	$\mathbf{codata}\, \neg^{-} A\, \mathbf{where}$
$\quad \mathsf{not}^{+} : \vdash \neg^{+} A \mid A$	$\quad \mathsf{not}^{-} : A \mid \neg^{-} A \vdash$

Figure 35. Declarations of the basic data and co-data types.

have already informed the core structural theory how to correctly implement our chosen strategy. Therefore, if we are evaluating our program according to call-by-value, would have to evaluate M first before substituting it for x. Likewise, in call-by-name, we would have to evaluate K into a co-value before it may be substituted for α. Next, we have the following η axiom for functions

$$(\eta^{\to}) \qquad z = \mathbf{case}\,(\mathsf{call}[x, \alpha] \Leftarrow \langle z \| \mathsf{call}[x, \alpha] \rangle)$$

which says that an unknown function z is equivalent to a trivial case abstraction that matches a function call, and forwards it along unchanged to z. Here, it is important to note that the η rule would not work if we replaced z with a general term M. The problem is exactly the same as in the call-by-value λ-calculus: if we are allowed to η expand any term, then we have that

$$5 =_{\beta} (\lambda x.5)\ (\lambda y.\Omega\ y) =_{\eta} (\lambda x.5)\ \Omega \approx \Omega$$

where Ω stands in for a term that loops forever. So if we allow for η expanding arbitrary terms in the call-by-value λ-calculus, then a value like 5 is the same thing as a program that loops forever. The solution in the call-by-value λ-calculus is to limit the η rule to only apply to values. Here, we use the variable z to stand in for an unknown value, since we are only allowed to substitute values for variables. This has the nice side effect that neither the β or η rules explicitly mention values or co-values in any way — they are strategy independent. To make the comparison with previous characterizations of functions in the sequent calculus, we can also express these rules a more traditional syntax:

$$(\beta^{\to}) \qquad \langle \lambda x.M \| M' \cdot K \rangle = \langle M' \| \mathbf{in}\, x \Rightarrow \langle M \| K \rangle \rangle$$
$$(\eta^{\to}) \qquad z = \lambda x.\mathbf{out}\, \alpha \Leftarrow \langle z \| x \cdot \alpha \rangle$$

The β and η rules for user-defined data types follow a similar, but mirrored, pattern. For example, the β rule for $\oplus$ performs case analysis on the tag of the term without requiring that the sub-term be a value:

$$(\beta^{\oplus}) \qquad \langle \mathsf{inl}(M) \| \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow S_1 \mid \mathsf{inr}(y) \Rightarrow S_2] \rangle = \langle M \| \mathbf{in}\, x \Rightarrow S_1 \rangle$$
$$(\beta^{\oplus}) \qquad \langle \mathsf{inr}(M) \| \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow S_1 \mid \mathsf{inr}(y) \Rightarrow S_2] \rangle = \langle M \| \mathbf{in}\, y \Rightarrow S_2 \rangle$$

Again, this rule works because we end up putting the sub-term in interaction with an input abstraction, allowing the equational theory of the structural core take care of managing evaluation order. For example, in the call-by-value setting, while this rule is stronger than the one given for Wadler's dual sequent calculus, it is still allowable according to the given call-by-value CPS transformation [111]. The η rule for $\oplus$ is also similarly mirrored from functions, where we expand an unknown co-value γ into a case abstraction:

$$(\eta^{\oplus}) \quad \gamma = \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow \langle \mathsf{inl}(x) \| \gamma \rangle \mid \mathsf{inr}(y) \Rightarrow \langle \mathsf{inl}(x) \| \gamma \rangle]$$

Recall that both Wadler's dual sequent calculus and system L included ς rules that lift out sub-(co-)terms from inside a structure. However, we did not include them as axioms in our equational theory. As it turns out, these rules are derivable in general by using the β and η axioms described above. For example, we have the ς rules for $\oplus$:

$$(\varsigma^{\oplus}) \qquad \mathsf{inl}(M) = \mathbf{out}\, \alpha \Leftarrow \langle M \| \mathbf{in}\, x \Rightarrow \langle \mathsf{inl}(x) \| \alpha \rangle \rangle$$
$$(\varsigma^{\oplus}) \qquad \mathsf{inr}(M) = \mathbf{out}\, \alpha \Leftarrow \langle M \| \mathbf{in}\, x \Rightarrow \langle \mathsf{inr}(x) \| \alpha \rangle \rangle$$

These rules can be derived by η expansion followed by β reduction:

$$\begin{aligned} \mathsf{inl}(M) &=_{\eta_\mu} \mathbf{out}\, \alpha \Leftarrow \langle \mathsf{inl}(M) \| \alpha \rangle \\ &=_{\eta^{\oplus}} \mathbf{out}\, \alpha \Leftarrow \langle \mathsf{inl}(M) \| \mathbf{case}\,[\mathsf{inl}(x) \Rightarrow \langle \mathsf{inl}(x) \| \alpha \rangle \mid \ldots] \rangle \\ &=_{\beta^{\oplus}} \mathbf{out}\, \alpha \Leftarrow \langle M \| \mathbf{in}\, x \Rightarrow \langle \mathsf{inl}(x) \| \alpha \rangle \rangle \end{aligned}$$

Notice here that the steps of this derivation are captured exactly by our formulation of β and η axioms: 1. the ability to η expand a co-variable, and 2. the ability to perform β reduction immediately to break apart a structure once the constructor is seen. We also have similar ς rules for functions

$$(\varsigma^{\to}) \qquad M \cdot K = \mathbf{in}\, x \Rightarrow \langle M \| \mathbf{in}\, y \Rightarrow \langle x \| y \cdot K \rangle \rangle$$
$$(\varsigma^{\to}) \qquad V \cdot K = \mathbf{in}\, x \Rightarrow \langle \mathbf{out}\, \alpha \Leftarrow \langle x \| V \cdot \alpha \rangle \| K \rangle$$

which are again derivable by a similar procedure of η expansion and β reduction. These particular ς axioms for functions are interesting because they were left out of Wadler's dual sequent calculus [111], however, they were implicitly present in the equational theory [112] as a consequence of the β and η axioms. This same procedure words for all the definable (co-)data types.

Remark 13. Aggressively applying the various ς rules, so that structures are only built out values and co-values are, gives a procedure for producing a program in a sub-syntax that corresponds to focalization described in Section 6. The focalized sub-syntax corresponds to A-normal forms [100], which give a name to the results of all non-trivial computations in a program. In particular, the ς rules for functions provides a system for transforming a general program in $\overline{\lambda}\mu\tilde{\mu}$ into an equivalent program in either of the two

sub-syntaxes, $\overline{\lambda}\mu\tilde{\mu}_Q$ or $\overline{\lambda}\mu\tilde{\mu}_T$, by choosing either a call-by-value or call-by-name strategy. *End remark* 13.

7.3 Composing strategies

We have present a general framework for describing all the basic connectives discussed so far, giving a mechanism for extending the syntax and semantics of the sequent calculus to account for a wide variety of new structures. However, what about the connectives that mixed polarities in Section 6.5? Can we include the shifts, dual negation, and "primordial" function type into our notion of user-defined data and (co-)data types? Also, what happened to polarized logic's promise of using multiple evaluation strategies in a single program? Is there a way to instantiate the parametric equational theory with more than one strategy at the same time?

The answer to all of these questions is to take a step back from polarized logic and look at the more general case. Separating types into different classifications is not a new idea, and shows up in several type systems in the form of *kinds*. Effectively, kinds classify types in the same way that types classify terms, *i.e.,* kinds are types "one level up the chain." Therefore, perhaps we can use kinds to fill in the essential role for distinguishing evaluation strategies that was filled by polarities in polarized logic. So that the way that polarized languages use polarities to denote evaluation order, we may use kinds to denote evaluation order. That way, we can instantiate the parametric equational theory with a *composite* strategy made up of several *primitive* strategies, and the kinds make sure that the strategies of terms and co-terms correctly line up.

For example, let's suppose we want a wholly call-by-value notion of pair, which we will suggestively denote by a kind named $\mathcal{V}$.[14] We can make this intent explicit by adding explicit kinds to the declaration of $\otimes$ from Figure 35:[15]

$$\begin{array}{l}\mathbf{data}\,(A:\mathcal{V})\otimes(B:\mathcal{V}):\mathcal{V}\,\mathbf{where}\\ \quad \mathsf{pair}:A:\mathcal{V},B:\mathcal{V}\vdash A\otimes B:\mathcal{V}|\end{array}$$

Here, we say that both components of the pair have some type of kind $\mathcal{V}$, and the pair type also is of kind $\mathcal{V}$. If we interpret the kind $\mathcal{V}$ as denoting the strategy $\mathcal{V}$, then this declaration gives us the basic pair type in the call-by-value instance of the parametric equational theory. However, suppose we also want a pair that has one call-by-name component, one call-by-value component, and is overall evaluated in a call-by-name way. We can signify this intent by declaring a different pair type that uses two different kinds, $\mathcal{N}$ and $\mathcal{V}$:

$$\begin{array}{l}\mathbf{data}\,\mathsf{MixedProduct}(A:\mathcal{V},B:\mathcal{N}):\mathcal{N}\,\mathbf{where}\\ \quad \mathsf{MixedPair}:A:\mathcal{V},B:\mathcal{N}\vdash\mathsf{MixedProduct}(A,B):\mathcal{N}|\end{array}$$

In this declaration, the fact that the type A has kind $\mathcal{V}$ denotes that the first component should be evaluated with the call-by-value strategy $\mathcal{V}$, whereas the second component and the pair as a whole should be evaluated with the call-by-name strategy $\mathcal{N}$.

Now that we are looking at programs with multiple different strategies running around, we need to be able make sure that only terms and co-terms from the same strategy interact with one another. Otherwise, we could lose determinism of the equational theory. However, a full typing discipline is overkill. After all, the parametric equational theory, when instantiated with a single primitive strategy, did not need to use types to maintain determinism. Therefore, we use a type-agnostic kind system for making sure that all commands are well-kinded. By "type-agnostic," we mean that we are checking the properties like $M :: \mathcal{S}$, that is M is a term of some unknown type that has the kind $\mathcal{S}$. The kind system for the structural core of the sequent calculus is shown in Figure 37, and unremarkably resembles the ordinary type system except at "one level up." The whole point of the system is shown in the Cut rule that only allows commands between term and co-term of the same kind. The main property that distinguishes this from an ordinary type system is that we "forget" the types, effectively collapsing them down into a single universal type for each kind, similar to a generalized version of Zeilberger's [117] "bi-typed" system.

For example, for a term or co-term of type $\mathsf{MixedProduct}(A,B)$, we only know that they belong to some type of the kind $\mathcal{N}$, giving us the following two inference rules:

$$\frac{\Gamma\vdash M::\mathcal{V}|\Delta\quad\Gamma\vdash M'::\mathcal{N}|\Delta}{\Gamma\vdash\mathsf{MixedPair}(M,M')::\mathcal{N}|\Delta}$$

$$\frac{S::(\Gamma,x::\mathcal{V},y::\mathcal{N}\vdash\Delta)}{\Gamma|\mathbf{case}\,[\mathsf{MixedPair}(x,y)\Rightarrow S]::\mathcal{N}\vdash\Delta}$$

In order to make the kind of a term explicitly apparent in the syntax, we can annotate variables with their kind, like $x^{\mathcal{V}}$ and $y^{\mathcal{N}}$. Recall that the β rules do not make reference to the chosen strategy in any way, they are only responsible for breaking apart structures. This means that the β rules are completely unaffected by the use of composite strategies. For instance, we may run a program using $\mathsf{MixedProduct}$ in the same way as the call-by-value $\otimes$:

$$\begin{array}{l}\langle\mathsf{MixedPair}(M,M')\|\mathbf{case}\,[\mathsf{MixedPair}(x^{\mathcal{V}},y^{\mathcal{N}})\Rightarrow S]\rangle\\ =_\beta\langle M\|\mathbf{in}\,x^{\mathcal{V}}\Rightarrow\langle M'\|\mathbf{in}\,y^{\mathcal{N}}\Rightarrow S\rangle\rangle\\ =_{\tilde{\mu}_V}\langle M\|\mathbf{in}\,x^{\mathcal{V}}\Rightarrow S\left\{M'/y^{\mathcal{N}}\right\}\rangle\end{array}$$

Notice that as before, the input abstractions take over for determining evaluation order in even when multiple primitive strategies are in play. In this case, we are allowed to substitute M' for $y^{\mathcal{N}}$ since M' is a value in the sense of the strategy $\mathcal{N}$. However, we must first evaluate M before substituting it for $x^{\mathcal{V}}$, since we judge both M and the input abstraction for $x^{\mathcal{V}}$ according to the strategy $\mathcal{V}$.

Finally, using this notion of kinds to distinguish between different primitive strategies, we can give (co-)data declarations for the various connectives of polarized logic that mix polarities in terms of declarations using the two kinds $\mathcal{V}$ and $\mathcal{N}$. Types for the shifts, dual negations, and "primordial" functions (and the dual "primordial" substitution) are given in Figure 36. Notice how the kinds point out similar patterns as were used in polarized logic: all the data structures are call-by-value and all the co-data structures are call-by-name. Additionally, with the exception of the shifts, all the input parameters are call-by-value and all the output parameters are call-by-name.

Remark 14. Recall from Section 7.2 that although the η axioms for data and co-data types do not reference the chosen strategy, their expressive power is affected by the substitution principle, which is in turn affected by the choice of values and co-values. In light of this observation, if we were forced to pick only *one* strategy for all data types and *one* strategy for all co-data types, it would make sense to pick the strategies that would give us the strongest equational theories. Therefore, if we want to make the η axiom for a data type as strong as possible, we should choose the call-by-value $\mathcal{V}$ strategy, since by substitution *every* co-term of that data type is equivalent to a case abstraction on the structure of the type. Likewise, if we want to make the η axiom for a co-data type as strong as possible, we should choose the call-by-name $\mathcal{N}$ strategy,

[14] Here we use the name $\mathcal{V}$ to mean both a strategy (a set of values and co-values) and a kind (a "type of types"), so even though the two are different things, the clash in naming is meant to make obvious the connection between the kind and the strategy. Both kinds and strategies are used in very different places, so the meaning of $\mathcal{V}$ can usually be distinguished from context.

[15] Adding explicit kinds to a data type declaration is not new; it is supported by GHC with the extension "kind signatures." Rather, the new idea is to have the kind impact the meaning of a term by denoting its evaluation strategy.

$$\begin{array}{ll}
\mathbf{data}\downarrow(A:\mathcal{N}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\uparrow(A:\mathcal{V}):\mathcal{N}\,\mathbf{where}\\
\quad \mathsf{shift}^{\mathcal{V}}:A:\mathcal{N}\vdash\downarrow A:\mathcal{V}| & \quad \mathsf{shift}^{\mathcal{N}}:|\uparrow A:\mathcal{N}\vdash A:\mathcal{V}\\
\mathbf{data}\,(A:\mathcal{N})_{\top}:\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(A:\mathcal{V})^{\perp}:\mathcal{N}\,\mathbf{where}\\
\quad \mathsf{Dual}^{\mathcal{V}}:\vdash A_{\top}:\mathcal{V}|A:\mathcal{N} & \quad \mathsf{Dual}^{\mathcal{N}}:A:\mathcal{V}|A^{\perp}:\mathcal{N}\vdash\\
\mathbf{data}\,(A:\mathcal{V})-(B:\mathcal{N}):\mathcal{V}\,\mathbf{where} & \mathbf{codata}\,(A:\mathcal{V})\to(B:\mathcal{N}):\mathcal{N}\,\mathbf{where}\\
\quad \mathsf{uncall}:A:\mathcal{V}\vdash(A-B):\mathcal{V}|B:\mathcal{N} & \quad \mathsf{call}:A:\mathcal{V}|(A\to B):\mathcal{N}\vdash B:\mathcal{N}
\end{array}$$

Figure 36. Declarations of composite strategy data and co-data types.

$$\frac{}{\Gamma, x::\mathcal{S}\vdash x::\mathcal{S}|\Delta}\ Var \qquad \frac{}{\Gamma|\alpha::\mathcal{S}\vdash\alpha::\mathcal{S},\Delta}\ CoVar \qquad \frac{\Gamma\vdash M::\mathcal{S}|\Delta \quad \Gamma|K::\mathcal{S}\vdash\Delta}{\langle M\|K\rangle::(\Gamma\vdash\Delta)}\ Cut$$

$$\frac{S::(\Gamma\vdash\Delta)}{\Gamma\vdash\mathbf{out}\,\alpha\Leftarrow S::\mathcal{S}|\Delta}\ AR \qquad \frac{S::(\Gamma\vdash\Delta)}{\Gamma|\mathbf{in}\,x\Rightarrow S::\mathcal{S}\vdash\Delta}\ AL$$

Figure 37. Type-agnostic kind system for the structural core of the sequent calculus.

since by substitution *every* term of that co-data type is equivalent to a co-case abstraction on the co-structure of the type. In this sense, the decision use of polarities (*i.e.,* the data/co-data divide) to determine evaluation strategy is the same as choosing strategies to get the *strongest* and *most universal* η principles for every (co-)data type. *End remark* 14.

8. Future directions

The sequent calculus holds some exciting potential as a computational model: intertwining multiple evaluation strategies, explaining the meaning of duality in programming languages, giving a rigorous framework for structures and pattern matching, showing a natural setting for first-class manipulation of control flow, providing a foundation for a message-passing style of programming, and revealing the connection between object-oriented and functional paradigms. However, there has been a lot of development of the λ-calculus as a model of programming, and the sequent calculus has a lot of catching up to do. In particular, the λ-calculus has been extended with several features, summarized in the λ-cube [15], that make it a rich substrate for studying programs and languages. What we need is a similar development of extensions to the sequent calculus. For instance, if we want to fulfill the promise that the sequent calculus provides a foundation of object-oriented programming, we will need to explain phenomenon like other computational effects, self-referential objects (the "this" or "self" reference), subtyping, and parametric polymorphism (generics and packages). Additionally, the previous study of the computational interpretation of the sequent calculus observed similarities with abstract machines. This suggests that a sequent-based language would be a good fit as an intermediate language of a compiler, bridging between a high-level programming language and lower-level implementation.

8.1 Computational effects

We have already discussed in Section 4.3 how the sequent calculus naturally expresses a form of computational effect that allows for manipulation over control flow, equivalent to control operators like Scheme's [74] callcc, or Felleisen's [40] $\mathcal{C}$. However, we would also like to explain other computational effects, like mutable state, exceptions, external input and output, *etc*.

One effect that is particularly interesting from a theoretical standpoint, and which is related to classical control, is *delimited control* [27, 28, 36, 39], which enjoys a remarkable completeness property: any effect that can be simulated by a monad can be implemented by delimited control [43–45]. The general notion of delimited control effects has spawned a number of variations and extensions, including differences in the way context delimiters are scoped [28, 36] and the ability to give names to delimiters [31, 34, 57, 75]. However, we would rather understand delimited control from more basic principles. So far, the understanding of classical control[8, 11, 64] in terms of Parigot's $\lambda\mu$-calculus [89] has been extended to delimited control in different ways: adding a dynamic co-variable in the call-by-value setting [11], relating control effects to logical subtraction [10], or by relaxing the syntax in the call-by-name setting [64].

Additionally, the $\lambda\mu$-calculus, as a language for classical logic in the style of natural deduction, has a relation to the sequent calculus [111]. Therefore, it would be interesting to look at how the sequent calculus might be similarly extended with delimited control. Of particular interest is the relationship between logical subtraction and control effects. For all the basic (co-)data types in Figure 35, the ones whose constructors have exactly one conclusion ($\oplus$, $\otimes$, 1, 0, &, $\top$, $\to$) have a good understanding in programming languages. The other connectives, that make use of multiple conclusions, are not so well understood. Perhaps the key to understanding these types in a more traditional setting relies on some more advanced manipulation of control flow.

Another general framework for effects that has recently gained attention is the notion of algebraic effect handlers [16, 18, 94] which extends programs with an *administrator* that handles requests for primitives effectful operations. Data and co-data in the sequent calculus may be seen as a way of describing the interface of interaction between the program and the administrator. In particular, the dual nature of data and co-data suggests two dual ways of describing the third party that defines an effect: *third-party as an administrator*, where there is an external, abstract definition for the behavior of primitive operations and the program must yield execution when reaching an effectful operation, or *third-party as a resource*, where the program carries around extra, external information that may be scrutinized in order to perform arbitrary effect-dependent behavior. The sequent calculus already gives a rich language for interacting with contexts described as structures and abstract processes; extending the sequent calculus with a notion of algebraic effects would extend this language to programmable com-

putational effects, similar to Felleisen's proposal for abstract continuations [41].

8.2 Induction and co-induction

An important step toward making the sequent calculus into a full-fledged model of general computation is to provide some form of looping or recursion.[16] Functional programming languages like SML and proof assistants like Coq rely heavily on *inductive data types* such as lists and binary trees. These inductive data structures allow the program to represent arbitrarily large, but finite information, which is then analyzed by recursively looping over the structure of the data. Currently, we have a large set of tools at our disposal for reasoning about programs that make use of inductively defined data. For instance, the Calculus of Inductive Constructions [67], which lies at the heart of Coq, gives a rich theory for inductively constructed data that allows for proving properties about recursive programs; for example, one may want to check that an inductive program for processing lists always terminates.

On the other hand, there is a another notion of recursion dual to induction, called *co-induction*, which is also an important programming tool. Co-inductive types give a model for an infinite source of information, like a character input stream, or a continually running process like a server or operating system that should always be responsive. Additionally, co-recursion, the general form of co-induction, lies at the heart of self-referential objects in object-oriented languages. Unfortunately, our tools for reasoning about co-inductive programs are less well-developed than those for inductive programs. For example, the current extension of the Calculus of Inductive Constructions with co-inductive data structures breaks the type system since evaluation does not preserve types [48, 88]. This disparity between induction and co-induction is unsatisfying. Since the two concepts are dual to one another, if we know how to express one we should also know how to express the other.

Instead, it would be interesting to utilize the natural duality in the sequent calculus to treat induction and co-induction as equal and opposite approaches to recursion. It follows that the dual to an inductive data type is a *co-inductive co-data type*, where the consumer is an arbitrarily large but finitely constructed observation, and the producer is a recursive process for responding to observations. For example, the type of infinite streams is defined by the observations for retrieving the head and the tail of a stream, and a value for producing a stream is an abstract infinite process defined by structural recursion on the possible observations. This view coincides with Hagino's [58] model of induction and co-induction that defines co-data types by a list of destructors [59], and gives a calculus for the more recent development of co-patterns [3] used to formulate strongly normalizing co-induction in functional languages [2]. In the sequent calculus, both induction and co-induction are unified under the umbrella of structural recursion. Intuitively, an inductive consumer sees its input and thinks, "Hmm, that's a rather large collection of information, I better process it bit-by-bit," whereas a co-inductive producer sees its request and thinks, "Hmm, that's a rather large question, I better answer it bit-by-bit." The difference between the two is a matter of orientation: induction consists of a finitely constructed producer and a looping consumer, whereas co-induction consists of a finitely constructed consumer and a looping producer. In this setting, for anything that can be achieved inductively, the dual follows co-inductively and vice versa.

[16] We already have encodings of fixpoint combinators that exist in the untyped language, similar to the untyped λ-calculus, but it would be interesting to have a more primitive treatment of recursion in the language. Especially when we consider restricted forms of recursion that are well-behaved and do not get stuck in an infinite loop.

8.3 Subtyping

In a typical object-oriented language, subtyping allows for casting an object into one with less specified behavior. For example, in a language using a form of duck typing like Go, an object that responds to the messages M_1, M_2, and M_3 may be used in a context that only requires that object to have a response for the message M_1, because it will never receive an unknown message. Duality in the sequent calculus also gives us the opposite form of subtyping, where a data structure that is built out of the constructors F_1 and F_2 may be used in a context that considers cases for additional constructors. The general principle, in terms of data and co-data, is that a structure and case abstraction may safely interact with one another so long as the structure matches one of the cases that was taken into consideration. Subtyping allows for a more liberal use of types: instead of restricting interactions between exactly the same type, we may allow for interactions where one side has a type that considers more cases than necessary or uses fewer possible constructors.

It would be interesting to give a sequent calculus presentation of subtyping that defines the subtyping relationship as a generalized guarantee of safe interaction between a producer and a consumer of two different types. In addition to establishing the dual relationship between subtyping of data and co-data types, it would also be interesting to consider how variance might arise from the directional flow of information and interplay between producers and consumers, *i.e.,* input and output types, in (co-)data type definitions. For example, the correct subtype relation, written $A :< A'$ meaning that A is a subtype of A', for the function type is:

$$\frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'}$$

Notice that the subtype relationship for the return types of the function follows the same direction as the function itself (is *covariant*), but the subtyping for the argument types is reversed (is *contravariant*). This comes from the fact that the consumer for function types, a function call, contains another consumer (which is covariant, as it follows the current flow of information) in addition to a producer (which is contravariant, as it reverses the flow of information). It would be interesting to see if this same pattern follows not only for ordinary (co-)data types, like pairs and sums, but also more exotic types that correspond to connectives for logical negation and subtraction. In general, this pattern would allow for a generic principle for inferring the variance of arbitrary user-defined data and co-data types.

8.4 Parametric polymorphism

Another type feature needed to model modern programming languages is *quantification*, *i.e.,* parametric polymorphism. In particular, *universal quantification* plays an important role in both functional and modern object-oriented languages, showing up as parametric polymorphism in languages like SML and Haskell, and as generics in Java and C#. Universal quantification allows the programmer to write generic functions once and for all. For example, the identity function, which just returns its argument unchanged, may be given the type $\forall a.a \to a$, so that the same identity function may be used at any type, rather than defining a different identity function for each specific type at which it is used. The dual concept to universal quantification is *existential quantification*, which can be used to model representation hiding in terms of modules or packages in programming languages [60]. For example, we may represent an abstract definition of integer sets, with operations for the constant empty set, building singleton sets ($\mathsf{Int} \to Set$), and set union ($Set \to Set \to Set$), as the existential type $\exists a.a \otimes (\mathsf{Int} \to a) \otimes (a \to a \to a)$. This type provides the essential operations, but hides the actual type used to represent set values. While the code

defining the set operations may use knowledge about the internal representation, any consumer is prevented from using this knowledge.

It would be interesting to extend the sequent calculus with a notion of universal quantification. One approach could be based on type abstraction and instantiation, similar to System F [49, 95] which avoids the known problems of combining parametric polymorphism with computational effects [109, 113] like mutable references and control operators. Since type abstraction in System F is conceptually a function taking a type as a parameter, it should be modeled as co-data. Using the duality of the sequent calculus, a formulation for existential quantification as a data structure, containing the packaged term and the hidden type, comes out automatically from the co-data type for universal quantification. The form of modularity given by existential data types is different from the form given by ordinary co-data types, similar to the differences between abstract data types and procedural abstractions [96]. As a much, much longer term goal, it would be interesting to consider how these dualities may relate to dependent types. In particular, the dependent Σ type is conceptually a more advanced notion of existential quantification, $\exists$. However, Σ types are represented as products defined by projection (*i.e.,* co-data) rather than the constructed pair representation of $\exists$ (*i.e.,* data). This suggests an alternate form of dependent universal quantification that is dual to Σ types, and is expressed as a data structure similar to sum types, filling in the following table of quantification:

	co-data	data
universal	$\forall$	?
existential	Σ	$\exists$

Furthermore, one feature that has been recently added as an extension to Haskell is *kind polymorphism* [114], allowing for types (and thereby constructors) to vary in the kinds that they inhabit. Following the discussion in Section 7.3 about the role of kinds in classifying evaluation strategies, if we were to extend the parametric sequent calculus with kind polymorphism, would this also imply some notion of *strategy polymorphism*?

8.5 Intermediate languages

In lieu of translating a program directly to machine code, modern compilers often use one or more intermediate languages for representing and reasoning about programs. Intermediate languages can aid in the process of optimizing and translating programs in two different ways: they provide a simpler core system that is easier to reason about, or they expose lower-level execution details that the compiler may take advantage of. For example, GHC, the primary compiler for Haskell, uses a core intermediate language FC [108] based of an extension of System Fω [50], a λ-calculus with parametric polymorphism and data types. GHC's core language provides a simpler setting for understanding Haskell programs by translating many of the advanced Haskell features, like type classes and GADTs, to a foundational core. Alternately, some compilers like SML/NJ use a continuation-passing style (CPS) intermediate language [5] which inverts a program so that the current step of evaluation is in focus and the evaluation context is reified into a *continuation*, a function waiting for a value before resuming evaluation, thereby explicitly exposing the evaluation order in the syntax of the program.

It would be interesting to test if the parametric sequent calculus is a good fit as an intermediate representation in a compiler. On the one hand, we illustrated in Section 4 how the sequent calculus forms a model of computation that is similar to an abstract machine [23, 76], where a function call is reified into a literal call-stack in the calculus. This presents a representation of the program in which the most relevant information about its execution behavior is lifted up to the top of the syntax tree. Lifting up the current step of execution is especially relevant in a lazy language where local bindings contain delayed computation, and may be represented explicitly in the sequent calculus [14]. On the other hand, we discussed in Section 7 how the sequent calculus still admits high-level reasoning in the form of equational reasoning and term reduction with simple β and η principles, similar to the λ-calculus. In particular, it would be interesting to experiment with a sequent-based typed intermediate language as an alternative to GHC's core language FC, to test if it provides a better setting for some of the program analyses and optimizations performed by the compiler.

GHC makes extensive use of analyzing the calling patterns for functions. For instance, as part of its simplification procedure, GHC expresses some domain-specific optimizations as a set of user-definable rewrite rules [72], which may transform a program into an equivalent but (hopefully) more efficient version. As an example, consider the `map` operation, which applies a function to every element in a list. In a pure language like Haskell, mapping a function g over a list and then mapping a second function f over that result is the same thing as mapping the composed function $f \circ g$ over the original list. We can express this knowledge to the compiler as a rewrite rule:

```
forall f g xs. map f (map g xs) = map (f.g) xs
```

Intuitively, to use this rewrite rule on an expression, the compiler must first (1) find a place where `map` is used and (2) check if the arguments to `map` match the pattern of the rule. These steps are hindered by the λ-calculus representation of GHC's core intermediate language, since the identifier `map` is buried at the bottom of the tree with its arguments floating in its context. For example, as was shown in Figure 7, the sequent calculus representation places a function like `map` at the top of the syntax tree with all of its arguments listed in order. Furthermore, other optimization procedures for specializing recursive functions [69] rely on searching for calls to function identifiers with particular call-patterns. In the sequent calculus, these call-patterns are subsumed by the ordinary notion of pattern-matching on a co-term (representing the calling context) as the dual to pattern-matching on a term.

The main advancement of GHC's current intermediate language FC is the addition of *equality constraints* [108], which are witnesses that two types are the same, and allow for type-safe casting. Equality constraints have shown to be a useful tool for representing various type features and extensions of Haskell, including GADTs [71, 104], type families [103], and operationally free `newtype` declarations. As it turns out, by correcting the variance properties for function type equality, FC's notion of equality constraints falls exactly into a subtyping regime, with the addition that every subtyping relationship is symmetric so that casting may be performed in both directions. It would be interesting to study FC's explicit equality constraints from the framework of subtyping in the sequent calculus. In particular, how does explicit casting in FC fit into the interplay between terms, co-terms, and the interactions between them. Understanding this question would help clarify the management of type-safe casts in the semantics of FC, and open up the possibility of subtyping in GHC's intermediate language, and possibly later even extending Haskell itself with subtyping.

Other specific points that might be interesting to look into along this line of research is to:

- Study the correctness of compiler optimizations expressed in the sequent calculus in terms of *observational equivalence* [86]. The sequent calculus already gives a setting for explicitly working with contexts, but the notion of co-terms may need to be extended to directly formulate observational equivalence.

- Study strictness analysis for lazy functional languages in terms of the intermediate sequent language with multiple evaluation strategies (Section 7.2), specifically call-by-need and call-by-value for Haskell. This would incorporate both the currently lazy intermediate language and strict constructs into a single a unified language, different from the approach taken by Strict Core [17].
- Study the run-time behavior of unboxed types [70] in an intermediate lazy language in terms of multiple evaluation strategies in the sequent calculus, in order to upgrade strict unboxed types from "nearly" first-class to *fully* first-class citizens on par with lazy boxed types.
- Explore the connection between the *virtual function table* [35] (vtable) intermediate representation of objects in languages like C++ with the co-data representation in the sequent calculus. In a sequent calculus with (co-)data, encoding strict objects with vtables appears dual to encoding lazy functional case analysis as *return vectors* [68] in Haskell. Intriguingly, the use of vectored returns in GHC has been reconsidered [82] due to their negative performance impact on modern processors, which should prompt a re-examination of vtables.

References

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996. ISBN 978-0-387-94775-4.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ICFP*, pages 185–196, 2013.

[3] A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: programming infinite structures by observations. In *POPL*, pages 27–38, 2013.

[4] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[5] A. W. Appel and D. B. Macqueen. Standard ml of new jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, 1991.

[6] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[7] Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP*, pages 871–885, 2003.

[8] Z. M. Ariola and H. Herbelin. Control reduction theories: the benefit of structural substitution. *J. Funct. Program.*, 18(3):373–419, 2008.

[9] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *POPL*, pages 233–246, 1995.

[10] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP*, pages 40–53, 2004.

[11] Z. M. Ariola, H. Herbelin, and A. Sabry. A proof-theoretic foundation of abortive continuations. *Higher-Order and Symbolic Computation*, 20(4):403–429, 2007.

[12] Z. M. Ariola, A. Bohannon, and A. Sabry. Sequent calculi and abstract machines. *ACM Trans. Program. Lang. Syst.*, 31(4):13:1–13:48, May 2009. ISSN 0164-0925. doi: 10.1145/1516507.1516508. URL `http://doi.acm.org/10.1145/1516507.1516508`.

[13] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In *TLCA*, volume 6690 of *lncs*, 2011.

[14] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In *FLOPS*, pages 32–46, 2012.

[15] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[16] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Mathematics and Computation*, abs/1203.1539, 2012.

[17] M. C. Bolingbroke and S. L. P. Jones. Types are calling conventions. In *Proceedings of the Second ACM SIGPLAN Symposium on Haskell*, pages 1–12, 2009.

[18] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *TACS*, pages 244–272, 1994.

[19] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2:33, 346–366, 1932.

[20] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, Apr. 1936.

[21] T. Coquand. *Une théorie des Constructions*. Dissertation, University Paris 7, Jan. 1985.

[22] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173 – 202, 1987. ISSN 0167-6423.

[23] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming*, pages 233–243, 2000.

[24] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. *Theoretical Computer Science*, pages 165–181, 2010.

[25] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume 1. North-Holland, 1958.

[26] V. Danos, J.-B. Joinet, and H. Schellinx. LKQ and LKT: sequent calculi for second order logic based upon dual linear decompositions of the classical implication. In *Advances in Linear Logic*, volume 222, pages 211–224. Cambridge University Press, 1995.

[27] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark, 1989.

[28] O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[29] M. Davis. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions*. Dover Publication, 1965.

[30] N. de Bruijn. Automath, a language for mathematics. Technical Report 66-WSK-05, Technological University Eindhoven, Nov. 1968.

[31] P. Downen and Z. M. Ariola. A systematic approach to delimited control with multiple prompts. In *ESOP*, pages 234–253, 2012.

[32] P. Downen and Z. M. Ariola. The duality of construction. In *ESOP*, 2014.

[33] M. Dummett. The logical basis of methaphysics. In *The William James Lectures, 1976*. Harvard University Press, Cambridge, Massachusetts, 1991.

[34] R. K. Dybvig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06):687–730, 2007.

[35] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. ISBN 0-201-51459-1.

[36] M. Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190, 1988.

[37] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

[38] M. Felleisen and D. Friedman. Control operators, the secd machine, and the lambda calculus. In *Formal Descriptions of Programming Concepts*, pages 193–219, 1986.

[39] M. Felleisen and D. P. Friedman. A reduction semantics for imperative higher-order languages. In *PARLE (2)*, pages 206–223, 1987.

[40] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[41] M. Felleisen, M. Wand, D. P. Friedman, and B. F. Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *LISP and Functional Programming*, pages 52–62, 1988.

[42] A. Filinski. Declarative continuations and categorical duality. Master's thesis, Computer Science Department, University of Copenhagen, 1989.

[43] A. Filinski. Representing monads. In *POPL*, pages 446–457, 1994.

[44] A. Filinski. Representing layered monads. In *POPL*, pages 175–188, 1999.

[45] A. Filinski. Monads in action. In *POPL*, pages 483–494, 2010.

[46] M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3-4):259–288, 1993.

[47] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *Collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[48] E. Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. Ph.D. thesis, Ecole Normale Supérieure de Lyon, Dec. 1996.

[49] J.-Y. Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. Fenstad, editor, *Second Scandinavian Logic Symposium*, number 63 in Studies in Logic and the Foundations of Mathematics, pages 63–92. North Holland, 1971.

[50] J. Y. Girard. *Interprtation fonctionnelle et elimination des coupures de l'arithmtique d'ordre suprieur*. These d'tat, Universit de Paris 7, 1972.

[51] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[52] J.-Y. Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1(3):255–296, 1991.

[53] J.-Y. Girard. On the unity of logic. *Annals of Pure Applied Logic*, 59 (3):201–217, 1993.

[54] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[55] K. Gödel. On undecidable propositions of formal mathematical systems. in [29], 1934. lecture notes taken by Stephen C. Kleene and J. Barkley Rosser.

[56] T. Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[57] C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming Languages and Computer Architecture '95*, pages 12–23, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

[58] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, pages 140–157, 1987.

[59] T. Hagino. Codatatypes in ml. *J. Symb. Comput.*, 8(6):629–650, 1989.

[60] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[61] H. Herbelin. A lambda-calculus structure isomorphic to gentzen-style sequent calculus structure. In *CSL*, pages 61–75, 1994.

[62] H. Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de λ-termes et comme calcul de stratégies gagnantes*. Ph.D. thesis, University Paris 7, Jan. 1995.

[63] H. Herbelin. C'est maintenant qu'on calcule. In *Habilitation à diriger les reserches*, 2005.

[64] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In *POPL*, pages 383–394, 2008.

[65] W. A. Howard. The formulae-as-types notion of constructions. In *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980. Unpublished manuscript of 1969.

[66] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[67] *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012.

[68] S. L. P. Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, 2 (2):127–202, 1992.

[69] S. L. P. Jones. Call-pattern specialisation for haskell programs. In *ICFP*, pages 327–337, 2007.

[70] S. L. P. Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA*, pages 636–666, 1991.

[71] S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP*, pages 50–61, 2006.

[72] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in ghc. In *In Haskell Workshop*, pages 203–233. ACM SIGPLAN, 2001.

[73] A. C. Kay. The early history of smalltalk. In *HOPL Preprints*, pages 69–95, 1993.

[74] R. Kelsey, W. D. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[75] O. Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. *Functional and Logic Programming*, pages 304–320, 2010.

[76] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[77] J. Lambek and P. J. Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[78] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, Mar. 2002.

[79] P. B. Levy. Call-by-push-value: A subsuming paradigm. In *TLCA*, pages 228–242, 1999.

[80] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

[81] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Program.*, 8(3):275–317, 1998.

[82] S. Marlow, A. R. Yakushev, and S. L. P. Jones. Faster laziness using dynamic pointer tagging. In *ICFP*, pages 277–288, 2007.

[83] C. Mcbride. The derivative of a regular type is its type of one-hole contexts (extended abstract). unpublished manuscript, available via `http://strictlypositive.org/diff.pdf`. conor mcbride and ross paterson. applicative programming with effects, 2001.

[84] C. McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL*, pages 287–295, 2008.

[85] E. Moggi. Computational λ-calculus and monads. In *Logic in Computer Science*, 1989.

[86] J. H. Morris. *Lambda Calculus Models of Programming Languages*. Ph.D. thesis, Massachusetts Institute of Technology, 1968.

[87] G. Munch-Maccagnoni. Focalisation and classical realisability. In *Computer Science Logic*, pages 409–423. Springer, 2009.

[88] N. Oury. Coinductive types and type preservation. Message on the coq-club mailing list, June 2008.

[89] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR 92*. Springer-Verlag, 1992.

[90] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[91] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Comput. Sci.*, 1:125–159, 1975.

[92] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS*, pages 342–356, 2002.

[93] D. Prawitz. On the idea of a general proof theory. *Synthese*, 27(1): 63–77, 1974.

[94] M. Pretnar. *The Logic and Handling of Algebraic Effects*. Ph.D. thesis, University of Edinburgh, 2010.

[95] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. ISBN 3-540-06859-7.

[96] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 13–23. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-07155-X. URL `http://dl.acm.org/citation.cfm?id=186677.186680`.

[97] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4): 363–397, 1998.

[98] J. C. Reynolds. What do types mean?: From intrinsic to extrinsic semantics. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 309–327. Springer-Verlag New York, Inc., New York, NY, USA, 2003. ISBN 0-387-95349-3. URL `http://dl.acm.org/citation.cfm?id=766951.766967`.

[99] S. Ronchi Della Rocca and L. Paolini. *The Parametric λ-Calculus: a Metamodel for Computation*. Springer-Verlag, 2004.

[100] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan. 1992. ISSN 1045-3563. doi: 10.1145/141478.141563. URL `http://doi.acm.org/10.1145/141478.141563`.

[101] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4): 289–360, 1993.

[102] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

[103] T. Schrijvers, S. L. P. Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP*, pages 51–62, 2008.

[104] T. Schrijvers, S. L. P. Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for gadts. In *ICFP*, pages 341–352, 2009.

[105] P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *MSCS*, 11(2):207–260, 2001.

[106] P. Selinger. Some remarks on control categories. *Manuscript*, 2003.

[107] S. Singh, S. P. Jones, U. Norell, F. Pottier, E. Meijer, and C. McBride. Sexy types—are we done yet? Software Summit, Apr. 2011. URL `https://research.microsoft.com/apps/video/dl.aspx?id=150045`.

[108] M. Sulzmann, M. M. T. Chakravarty, S. L. P. Jones, and K. Donnelly. System f with type equality coercions. In *TLDI*, pages 53–66, 2007.

[109] M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.

[110] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[111] P. Wadler. Call-by-value is dual to call-by-name. In *Proceedings of ICFP*, pages 189–201. ACM, 2003.

[112] P. Wadler. Call-by-value is dual to call-by-name–reloaded. *Term Rewriting and Applications*, pages 185–203, 2005.

[113] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[114] B. A. Yorgey, S. Weirich, J. Cretin, S. L. P. Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *TLDI*, pages 53–66, 2012.

[115] N. Zeilberger. Focusing and higher-order abstract syntax. In *POPL*, pages 359–369, 2008.

[116] N. Zeilberger. On the unity of duality. *Annals of Pure Applied Logic*, 153(1-3):66–96, 2008.

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