Duality in Action

Paul Downen
Department of Computer & Information Science, University of Oregon, Eugene, OR, USA

Zena M. Ariola
Department of Computer & Information Science, University of Oregon, Eugene, OR, USA

Abstract

The duality between "true" and "false" is a hallmark feature of logic. We show how this duality can be put to use in the theory and practice of programming languages and their implementations, too. Starting from a foundation of constructive logic as dialogues, we illustrate how it describes a symmetric language for computation, and survey several applications of the dualities found therein.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Duality, Logic, Curry-Howard, Sequent Calculus, Rewriting, Compilation

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.1

Category Invited Talk

Funding This work is supported by the NSF under Grants No. 1719158 and No. 1423617.

1 Introduction

Mathematical logic, through the Curry-Howard correspondence [25], has undoubtably proved its usefulness in the theory of computation and programming languages. It gave us tools to reason effectively about the behavior of programs, and serves as the backbone for proof assistants that let us formally specify and verify program correctness. We've found that the same correspondence with logic provides a valuable inspiration for the implementation of programming languages, too. The entire computer industry is based on the difference between the *ability to know something* versus *actually knowing it*, and the fact that real resources are needed to go from one to the other. In other words, the cost of an answer is just as important as its correctness. Thankfully, logic provides solutions for both.

We start with a story on the nature of "truth" (Section 2), and investigate different logical foundations with increasing nuance. The *classical* view of ultimate truth is quite different from constructive truth, embodied by intuitionistic logic, requiring that proofs be backed with evidence. However, the intuitionistic view of truth sadly discards many of the pleasant dualities of classical logic. Instead, we can preserve duality in constructivity by re-imagining logic not as a solitary exercise, but as a dialogue between two disagreeing characters: the optimistic Sage who argues in favor, and the doubtful Skeptic who argues against. Symmetry is restored – still backed by evidence – when both sides can enter the debate.

This dialogic notion of constructive classical logic can be seen as a symmetric language for describing computation (Section 3). The Sage and Skeptic correspond to producers and consumers of information; their debate corresponds to interaction in a program. The two-sided viewpoint brings up many dualities that are otherwise hidden implicitly in today's programming languages: questions versus answers, programs versus contexts, construction versus destruction, and so on. But more than this, the symmetric calculus allows us to express more types – and more relationships between them – than possible in the conventional programming languages used today.

From there, we survey several applications of computational duality (Section 4) across both theoretical and practical concerns. The theory of the untyped λ-calculus can be improved by viewing functions as codata (Section 4.1). Duality can help us design and analyze different forms of loops found in programs and proofs (Section 4.2). Compilers use intermediate languages to help generate code and perform optimizations, and logic can be put to action at this middle stage in the life of a program (Section 4.3). To bring it all together, a general-purpose method based on orthogonality provides a framework for developing models of safety that let us prove that well-typed programs do what we want (Section 4.4).

2 Logic as Dialogues

One of the most iconic principles of classical logic is the *law of the excluded middle*, $A \vee \neg A$: everything is either true or false. This principle conjures ideas of an omniscient notion of truth. That once all is said and done, every claim must fall within one of these two cases. While undoubtedly useful for proving theorems, the issue with the law of the excluded middle is that we as mortals are not omniscient: we cannot decide for everything, *a priori*, which case it is. As a consequence, reckless use of the excluded middle means that even if we know something must be true, we might not know exactly *why* it is true.

Consider this classic proof about irrational power [20].

▶ **Theorem 1.** *There exist two irrational numbers, x and y, such that x^y is rational.*

Proof. Since $\sqrt{2}$ is irrational, consider $\sqrt{2}^{\sqrt{2}}$. *This exponent is either rational or not.*

- If $\sqrt{2}^{\sqrt{2}}$ rational, then $x = y = \sqrt{2}$ are two irrational numbers (coincidentally the same) whose exponent is rational (by assumption).
- Otherwise, $\sqrt{2}^{\sqrt{2}}$ must be irrational. In this case, observe that the exponent $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$ simplifies down to just 2, because $\sqrt{2}^2 = 2$, like so: $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}^2} = \sqrt{2}^2 = 2$. Therefore, the two chosen irrational numbers are $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$ whose exponent is the rational number 2. ◀

On the one hand, this proof shows Theorem 1 is true in the sense that appropriate values for x and y cannot fail to exist. On the other hand, this proof fails to actually demonstrate *which* values of x and y satisfy the required conditions; it only presents two options without definitively concluding which one is correct. The root problem is in the assertion that the "exponent is either rational or not." If we had an effective procedure to decide which of the two options is correct, we could simply choose the correct branch to pursue. But alas, we do not. Depending on an undecidable choice results in a failure to provide a concrete example verifying the truth of the theorem. Can we do better?

2.1 Constructive truth

In contrast to the proof of Theorem 1, constructive logic demands that proofs construct real evidence to back up the truth of a claim. The most popular constructive logic is *intuitionistic logic*, wherein a proposition A is only considered true when a proof produces specific evidence that verifies the truth of A [3, 24]. As such, the basic logical connectives are interpreted intuitionistically in terms of the shape of the evidence needed to verify them.

Conjunction Evidence for $A \wedge B$ consists of both evidence for A and evidence for B.
Disjunction Evidence for $A \vee B$ can be either evidence for A or evidence for B.

Existence Evidence for $\exists x{:}D.P(x)$ consists of a specific example value $n \in D$ (*e.g.,* a concrete number when the domain of objects D is $\mathbb{N}$) along with evidence for $P(n)$.

Universal Evidence for $\forall x{:}D.P(x)$ is an algorithm that, applied to any possible value n in the domain D, provides evidence for $P(n)$.

Negation Evidence for $\neg A$ is a demonstration that evidence for A generates a contradiction.

The most iconic form of evidence is for the existential quantifier $\exists x{:}D.P(x)$. Intuitionistically, we must provide a real example for x such that $P(x)$ holds. Instead, classically we are not obligated to provide any example, but only need to demonstrate that one cannot fail to exist, as in Theorem 1. This is why intuitionistic logic rejects the law of the excluded middle as a principle that holds uniformly for every proposition. Without knowing more about the details of A, we have no way to know how to construct evidence for A or for $\neg A$. But still, $A \vee \neg A$ is never false; intuitionistic logic admits there may be things not yet known.

Intuitionistic logic is famous for its connection with computation, the λ-calculus, and functional programming [25]. Constructivity also gives us a more nuanced lens to study logics. For example, one way of understanding and comparing different logics is through the propositions they prove true. In this sense, intuitionistic and classical logic are different because classical logic accepts that $A \vee \neg A$ is true in general for any A, but intuitionistic logic does not. But this reduces logics to be merely nothing more than the set of their true propositions, irrespective of the reason *why* they are true. In a world in which we care about evidence, this reductive view ignores all evidence. Instead, we can go a step further to also compare the informational content of evidence provided by different logics.

In this sense, intuitionistic logic does very well in describing why propositions are true, especially compared to classical logic. The evidence supporting the truth of different connectives (like conjunction and disjunction) and quantifiers (like existential and universal) are tailor-made to fit the situation. But the evidence demonstrating falsehood is another story. Indeed, intuitionistic logic does *not* speak directly about what it means to be false. Rather, it instead says indirectly that "not A is true," *i.e.,* $\neg A$. In this case, the evidence of falsehood is rather poor, and always cast in the same form as a hypothetical: truth would be contradictory. For example, concrete evidence that $\forall x{:}\mathbb{N}.\ x + 1 \neq 3$ is false should be a specific counterexample for which the property fails; the same informational content as the evidence needed to prove $\exists x{:}\mathbb{N}.x + 1 = 3$ is true. For example, choosing 2 for x leads to $2 + 1 \neq 3$, which is obviously wrong. Yet, an intuitionistic proof of $\neg\forall x{:}\mathbb{N}.x + 1 \neq 3$ is under no such obligation to provide a specific counterexample, it only needs to show that a counterexample cannot fail to exist. The intuitionistic treatment of falsehood sounds awfully similar to the noncommittal vagueness of classical truth. Can we do better?

2.2 Constructive dialogues

The famous asymmetry of intuitionism is reflected by its biased treatment of the two basic truth values: it demands concretely constructed evidence of truth, but leaves falsehood as the mere shadow left behind from the absence of truth. This models the scenario of a solitary Sage building evidence to support a grand theorem. When the wise Sage delivers a claim we can be sure it is true – and verify the evidence for ourselves – but what if the Sage is silent? Is that passive evidence of falsehood, or just merely an artifact that work takes time? What is missing is a devil's advocate to actively argue the other side.

In reality, the uncharted frontier on the edge of current knowledge is occupied by contentious debate. Before something is fully known, there is a space where multiple people can honestly hold different, conflicting claims, even though they are all ultimately interested

in discovering the same shared truth. There is no need to be confined to the isolated work of cloistered ivory towers. Instead, there can be a dialogue between disagreeing parties, who influence one another and poke holes in questionable lines of reasoning. The search for truth is then found inside the dialogue of debate, of (at least) two sides exchanging probing questions and rebutting answers, where the victorious side defeats their opponent by eventually constructing the complete body of evidence that finally proves their position.

To keep things simple, let's assume the proposition A is under dispute by only two people: the Sage and the Skeptic. Whereas the Sage is optimistically trying to prove A is true, as before, the Skeptic is doubtful and asserts A is false. The dispute over A is resolved by the process of dialogue between the Sage and the Skeptic. But who is responsible for providing the first piece of evidence supporting their claim? Whoever has the *burden of proof*.

A *positive burden of proof* is when the Sage must provide evidence supporting that A is true. The shape of evidence for A's truth follows the shape of the disputed proposition A, and shares similarities with the evidence of truth for the same intuitionistic logical concepts.

Conjunction Evidence for $A \otimes B$ is both evidence for A and evidence for B.
Disjunction Evidence for $A \oplus B$ is either evidence for A or evidence for B.
Existence Evidence for $\exists x{:}D.P(x)$ is an example value $n \in D$ along with evidence for $P(n)$.
Negation Evidence for $\ominus A$ is the same as evidence against A.

Notice that new symbols are used for the connectives, and the evidence for negation is completely different. Both changes are due to the fact that there are other logical concepts that demand evidence of falsehood, rather than truth. These involve a *negative burden of proof*, where the Skeptic must provide evidence supporting that A is false. Just like the positive burden of proof (and contrary to intuitionistic logic), the shape of the evidence against A depends on the shape of A.

Conjunction Evidence against $A \,\&\, B$ is either evidence against A or evidence against B.
Disjunction Evidence against $A \parr B$ is both evidence against A and evidence against B.
Universal Evidence against $\forall x{:}D.P(x)$ is a counterexample value $n \in D$ (*e.g.,* a concrete number when D is $\mathbb{N}$) along with evidence against $P(n)$.
Negation Evidence against $\neg A$ is the same as evidence for A.

Now we can see that the new symbols for conjunction and disjunction disambiguate between the positive and negative burdens of proof, which carry complementary forms of evidence. In contrast, the two quantifiers $\exists$ and $\forall$ are not duplicated, but rather arranged to prioritize "finite" evidence (one specific example or counter example in the domain) instead of "infinite" hypothetical evidence (a general algorithm for generating evidence based on any object in the domain). Furthermore, there are two different notions of negation, the positive $\ominus A$ and negative $\neg A$, internalizing the duality between evidence for and against. The construction of evidence for or against each connectives is captured by these inference rules with two judgments: A **true** directly verifies A's truth and A **false** directly refutes it.

$$\frac{A \textbf{ true} \quad B \textbf{ true}}{A \otimes B \textbf{ true}} \qquad \frac{A \textbf{ true}}{A \oplus B \textbf{ true}} \qquad \frac{B \textbf{ true}}{A \oplus B \textbf{ true}} \qquad \frac{n \in D \quad P(n) \textbf{ true}}{\exists x{:}D.P(x) \textbf{ true}} \qquad \frac{A \textbf{ false}}{\ominus A \textbf{ true}}$$

$$\frac{A \textbf{ false} \quad B \textbf{ false}}{A \parr B \textbf{ false}} \qquad \frac{A \textbf{ false}}{A \,\&\, B \textbf{ false}} \qquad \frac{B \textbf{ false}}{A \,\&\, B \textbf{ false}} \qquad \frac{n \in D \quad P(n) \textbf{ false}}{\forall x{:}D.P(x) \textbf{ false}} \qquad \frac{A \textbf{ true}}{\neg A \textbf{ false}}$$

What does the other party without the burden of proof do? While they can wait to rebut the specific evidence they are given, it may take a long time (perhaps forever) for that evidence to be constructed. And absence of evidence does not imply the evidence of

absence. For example, the Skeptic may doubt a universal conjecture, but cannot come up with a counterexample that shows it false yet; this alone does not prove the conjecture true. Instead, in the face of negative burden of proof, the Sage can prove truth with a hypothetical argument that no such evidence against exists: systematically consider all possible evidence for the falsehood of A and show that each one leads to a contradiction. Dually, the Skeptic – waiting for the positive burden of proof to be fulfilled – can prove falsehood by hypothetically refuting all evidence of truth, showing all possible evidence for the truth of A leads to a contradiction. These proofs by contradiction are captured by the following inference rules for a proposition A (having positive burden of truth) and B (having negative burden of proof) using a third and final judgment **contra** representing a logical contradiction.

$$\frac{\begin{array}{c}\overline{A\ \mathbf{true}}\\ \vdots\\ \mathbf{contra}\end{array}}{A\ \mathbf{false}} \qquad \frac{\begin{array}{c}\overline{B\ \mathbf{false}}\\ \vdots\\ \mathbf{contra}\end{array}}{B\ \mathbf{true}}$$

We can now see that the evidence for $\neg A$'s truth hasn't changed from Section 2.1. To show $\neg A$ true via proof by contradiction, we assume evidence that $\neg A$ is false – the same as assuming evidence A is true – and derive a contradiction. In contrast, $\ominus A$ is entirely new.

2.3 The duality of constructive evidence

Viewing logic as a dialogue between an advocate and adversary – rather than just a lone advocate building constructions by themself – already improves the evidence of falsehood by giving the adversary a voice. Moreover, it improves some pleasant symmetries of truth with a more nuanced library of logical connectives expressing the full range of burden of proof.

For example, consider the classical law of double-negation elimination, $\neg\neg A \implies A$ (where $\implies$ stands for implication): if A cannot be untrue, then A is true. Intuitionists reject this law because the evidence for $\neg\neg A$ is much weaker than for A. For example, the evidence for $\neg\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ is a hypothetical argument that only says that it is contradictory for $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ to lead to a contradiction. In contrast, one example of direct evidence for $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ is the witness that for $x = 3$ and $y = 9$, we have $3^2 = 9$. One possible conclusion, taken by intuitionists, is that double-negation elimination is just incompatible with constructive evidence. But another conclusion is that the wrong negation has been used. Instead, consider the shape evidence for $\ominus\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ given by the more refined, dual definitions of $\ominus$ and $\neg$ in Section 2.2: evidence proving $\ominus\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ true consists of evidence proving $\neg\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ false, which in turn is the same as just evidence proving $\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\ x^2 = y$ true. So while $\neg\neg A \implies A$ for a generic A might not be considered constructive, $\ominus\neg A \implies A$ definitively is.

More generally, we can look at how negation interacts with the other logical connectives. In classical logic, the de Morgan laws describe how negation distributes over dual connectives, converting between conjunction ($\wedge$) and disjunction ($\vee$) as well as existential ($\exists$) and universal ($\forall$) quantifiers, like so (where $\iff$ means "if and only if"):

$$\neg(A \vee B) \iff (\neg A) \wedge (\neg B) \qquad \neg(\exists x{:}D.P(x)) \iff \forall x{:}D.\neg P(x)$$
$$\neg(A \wedge B) \iff (\neg A) \vee (\neg B) \qquad \neg(\forall x{:}D.P(x)) \iff \exists x{:}D.\neg P(x)$$

However, not all of these laws hold intuitionistically. In particular, $\neg(A \wedge B) \not\Longrightarrow (\neg A) \vee (\neg B)$ because knowing that the combination of A and B is contradictory is not enough to show definitively which of A or B is contradictory. Likewise, $\neg(\forall x{:}D.P(x)) \not\Longrightarrow \exists x{:}D.\neg P(x)$ because, as we have seen before, knowing that it is contradictory for $P(x)$ to be universally true does not point out the specific element of D where P fails.

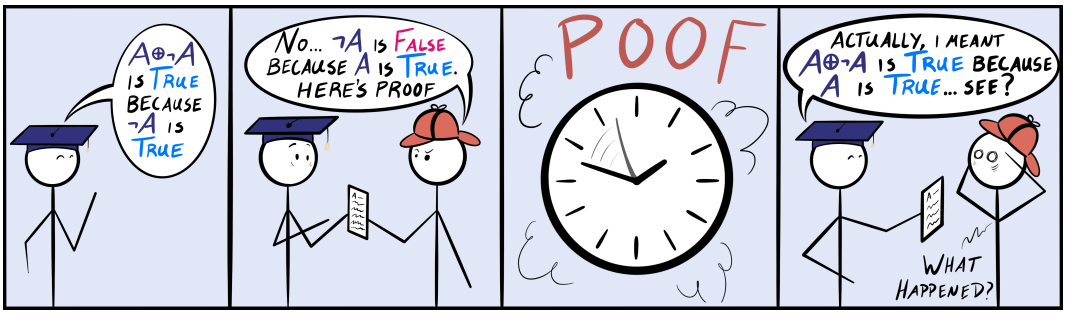

Figure 1 Law of excluded middle $A \oplus \neg A$ as a miraculous feat of time travel.

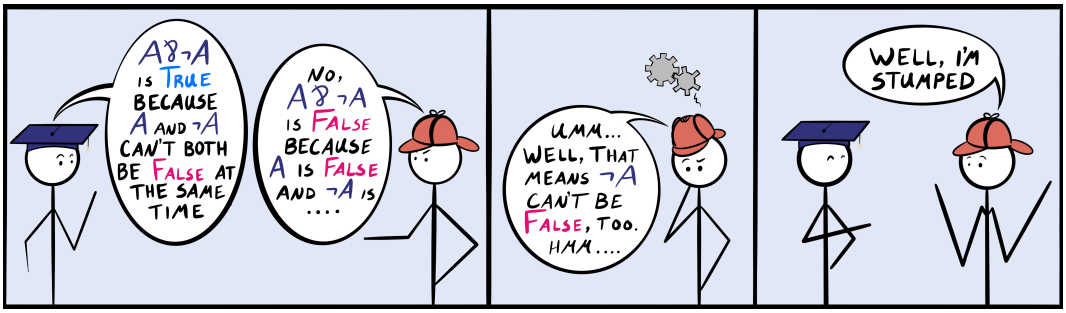

Figure 2 Law of excluded middle $A \parr \neg A$ as a mundane contradiction of falsehood.

Again, this problem with the asymmetry of the De Morgan laws can be seen as the classical logician being too vague about the burden of proof in their connectives. Rephrasing, we get the following symmetric versions of the De Morgan laws in terms of $\neg$ and $\ominus$ that are nonetheless constructive:

$$\begin{array}{rcl@{\qquad}rcl} \neg(A \oplus B) & \iff & (\neg A) \,\&\, (\neg B) & \ominus(A \,\&\, B) & \iff & (\ominus A) \oplus (\ominus B) \\ \neg(A \otimes B) & \iff & (\neg A) \parr (\neg B) & \ominus(A \parr B) & \iff & (\ominus A) \otimes (\ominus B) \\ \neg(\exists x{:}D.P(x)) & \iff & \forall x{:}D.\neg P(x) & \ominus(\forall x{:}D.P(x)) & \iff & \exists x{:}D.\ominus P(x) \end{array}$$

Note the new meanings of the previously offensive directions. On the one hand, evidence for $\ominus(A \,\&\, B)$ consists of evidence against $A \,\&\, B$ that boils down to either evidence against A or evidence against B; exactly the same as the evidence for $(\ominus A) \oplus (\ominus B)$. On the other hand, evidence against $\neg(A \otimes B)$ is the same as evidence for $A \otimes B$ which consists of evidence for both A and B simultaneously; exactly the same as the evidence against $(\neg A) \parr (\neg B)$. Similarly, evidence for $\ominus(\forall x{:}D.P(x))$ is a specific counterexample n in D such that $P(n)$ is false, which is exactly the same evidence needed to prove $\exists x{:}D.\ominus P(x)$ true.

Finally, let's return to the troublesome law of the excluded middle, $A \vee \neg A$ that we started with. Now equipped with two different versions of disjunction, we can understand this law constructively in two very different ways. The first understanding is based on the connection of classical logic with control [23], which represents the excluded middle as the seemingly impossible choice $A \oplus \neg A$. This proposition is true through a cunning act of bait and switch as shown in Figure 1. First, the Sage (in the blue academic square cap) baselessly asserts that $\neg A$ is true hoping that this is ignored. Later the Skeptic (in the Sherlock Holmesian brown deerstalker) can call the Sage's bluff by providing evidence that A is in fact true. In response, the Sage miraculously turns back the clock and changes their claim, instead asserting that A is true by using the Skeptic's own evidence against them. Now, the use of

time travel to change answers might seem a bit excessive, but luckily there is a much more mundane understanding based on the more modest $A \parr \neg A$. This proposition is true, almost trivially, as a basic contradiction shown in Figure 2, based on the fact that evidence for A is identical to evidence against $\neg A$. Here, the Sage merely asserts that A cannot be both true and false at the same time, to which the Skeptic has no retort. Thus, restoring the balance between true and false does a better job of explaining the constructive evidence of both classical and intuitionistic logic.

3 Computing with Duality

What does a calculus for writing logical dialogues look like? In order to prepare for representing hypothetical arguments, we will use a logical device called a *sequent* written:

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

that groups together multiple propositions into a single package revolving around a central *entailment* denoted by the turnstyle ($\vdash$). This sequent can be read as "if A_1, A_2, …, A_n are all true, then something among B_1, B_2, …, B_m must be true," or more simply "the conjunction of the left $(A_1, \ldots, A_n)$ implies the disjunction of the right $(B_1, \ldots, B_m)$." In order to understand the practical meaning of the compound sequent, it can help to look at special cases where it contains at most one proposition, forcing either the left or the right side of entailment to be empty (denoted by $\bullet$).

True The sequent $\bullet \vdash A$ means that A is true. The assumption is trivial because the conjunction of nothing is true (asserting everything in an empty set passes some test is a vacuously true statement). Since A is the only option on the right, A must be true.

False The sequent $A \vdash \bullet$ means that A is false. The conclusion is impossible because the disjunction of nothing is false (asserting that a true element is found among an empty set is immediately false). Since assuming A is true implies falsehood, A must be false.

Contradiction The sequent $\bullet \vdash \bullet$ denotes a contradiction. Following the reasoning above, $\bullet \vdash \bullet$ means "true implies false," which is just plainly impossible.

Thus far, this is just rephrasing the basic judgments we had discussed in Section 2.2 (therein written A **true**, A **false**, and **contra**, respectively). What is more interesting is how these forms of logical judgments can be reinterpreted as analogous forms of expressions in a calculus for representing computation as interaction.

Production The typing judgment $\bullet \vdash v : A \mid$ means that the *term* v *produces* information of type A. By analogy with Section 2.2, v represents the Sage who is trying to prove that A is true, and the value returned by v represents the evidence (of type A) that verifies the veracity of their claim.

Consumption The typing judgment $\mid e : A \vdash \bullet$ means that the *coterm* (*a.k.a* continuation) e *consumes* information of type A. The coterm e is analogous to the Skeptic who is trying to prove that A is false. In this sense, the covalue returned by e represents the evidence of a counter argument (of type A), which refutes values of type A.

Computation The typing judgment $c : (\bullet \vdash \bullet)$ means that the *command* c is an *executable statement*. Commands are the computational unit of the language where all reductions happen; each step of reduction corresponds to the back-and-forth dialogue between the Sage and the Skeptic. The fundamental form of commands is an interaction $\langle v \| e \rangle$ between a term v and a coterm e. The command $\langle v \| e \rangle$ means that the value returned by v is given to e as input, or dually the covalue constructed by e inspects v's output.

Note that, whereas terms $\bullet \vdash v : A \mid$ produce output (*i.e.,* provide answers) and coterms $\mid e : A \vdash \bullet$ consume input (*i.e.,* ask questions), the command $c : (\bullet \vdash \bullet)$ does not produce or consume anything itself, and acts as an isolated computation. To interact with a command, it is necessary to provide for free *variables* x which stand for places to read inputs and free *covariables* α standing for places to send outputs. Open commands with free (co)variables have the more general typing judgment

$$c : (x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash \alpha_1 : B_1, \alpha_2 : B_2, \ldots, \alpha_m : B_m)$$

As shorthand, we use Γ to denote a list of inputs $x_1 : A_1, \ldots, x_n : A_n$ and Δ to denote a list of outputs $\alpha_1 : B_1, \ldots, \alpha_m : B_m$. Similar to open commands of type $c : (\Gamma \vdash \Delta)$, we also have open terms $\Gamma \vdash v : A \mid \Delta$ and open coterms $\Gamma \mid e : A \vdash \Delta$ which might also use free (co)variables in Γ and Δ. Reference to these free (co)variables looks like this:[1]

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ VarR \qquad\qquad \frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta}\ VarL$$

As another example, the typing rule for safe interactions in a command $\langle v \| e \rangle$ corresponds to the Cut rule, which only connects together a producer and consumer that agree on a shared type A of information being exchanged:

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

The exciting part of this language is the way it renders the many dualities in logic directly in its syntax. We know that true is dual to false, and for the same reason things on the left of a sequent (*i.e.,* to the left of $\vdash$) are dual to things on the right. In this sense, the turnstyle $\vdash$ serves as an axis of duality in logic. The same axis exists in the form of commands $\langle v \| e \rangle$, where the left and right components are dual to one another. The most direct way to see this duality is in the exchange of answers and questions between the two sides of a command.

$$\xRightarrow{\text{Answers}} \quad \langle v \| e \rangle \quad \xLeftarrow[\text{Questions}]{}$$

However, there are many other dualities besides the answer-question dichotomy to explore along this same axis. While we imagine that information flows left-to-right, it turns out that control flows right-to-left. There is the construction-destruction dynamic between the creation of concrete evidence and the inspection of it, which can be arranged in either direction. Likewise, abstraction over types and hidden information gives rise to dual notions of generics (à la parametric polymorphism in functional languages and Java generics) which hide information in the consumer/client and modules (à la the SML module system) which hide information in the producer/server. So now let's consider how each of these computational dualities manifest themselves in the logical foundation of this language.

[1] The rules are named with an R and L because their conclusion below the horizontal line of inference introduces a new term on the *Right* of the turnstyle ($\vdash$) and a new coterm on the *Left*, respectively. This naming convention comes from the sequent calculus, which we will follow throughout the paper.

3.1 Positive burden of proof as data

In the constructive dialogues of Section 2.2, consider the case where the Sage has the positive burden of truth, and is responsible for constructing a concrete piece of evidence that backs up their claim that some proposition is true. The shape of the Sage's evidence depends on the proposition in question, and will contain enough information to fully justify truth in a way the Skeptic can examine. In computational terms, constructing this positive form of evidence corresponds to *constructing values* of a data type. In this sense, the Sage constructing evidence of A's truth is analogous to a producer v which constructs a value of type A.

For example, consider the basic cases for positive evidence of conjunction ($A \otimes B$) and disjunction ($A \oplus B$). The evidence of the conjunction $A \otimes B$ is made up of a combination of evidence v of A along with evidence w of B. In other words, it is a pair (v, w) of the tuple type $A \otimes B$. In contrast, the evidence of the disjunction $A \oplus B$ is a choice of either evidence v for A or evidence w for B. In other words, it is one of the two tagged values $\iota_1 v$ or $\iota_2 w$ of the sum type $A \oplus B$. These constructions are captured by the following typing rules, which resemble the inference rules for $A \otimes B$ **true** and $A \oplus B$ **true** in Section 2.2:

$$\frac{\Gamma \vdash v : A \mid \Delta \quad \Gamma \vdash w : B \mid \Delta}{\Gamma \vdash (v, w) : A \otimes B \mid \Delta} \otimes R \qquad \frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \vdash \iota_1 v : A \oplus B \mid \Delta} \oplus R_1 \qquad \frac{\Gamma \vdash w : B \mid \Delta}{\Gamma \vdash \iota_2 w : A \oplus B \mid \Delta} \oplus R_2$$

How, then, might the Skeptic respond to the evidence contained in these values? In general, the Skeptic is only obligated to show that evidence following these rules cannot be constructed, because their existence would lead to a contradiction. This corresponds to *pattern matching* or *deconstructing* on the shape of all possible values of a data type. A rebuttal of $A \otimes B$ is a process demonstrating a contradiction c given any generic pair $(x, y) : A \otimes B$, *i.e.,* in the context of two generic values $x : A$ and $y : B$. Similarly, a rebuttal of $A \oplus B$ is a process that demonstrates two different contradictions: c_1 which responds to a tagged value $\iota_1 x : A \oplus B$ (*i.e.,* in the context of a generic value $x : A$) and c_2 which responds to a tagged value $\iota_2 y : A \oplus B$ (*i.e.,* in the context of $y : B$). The two rebuttals are captured by the deconstructing consumers $\tilde{\mu}(x, y).c$ and $\tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2]$ given by these typing rules:

$$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x, y).c : A \otimes B \vdash \Delta} \otimes L \qquad \frac{c_1 : (\Gamma, x : A \vdash \Delta) \quad c_2 : (\Gamma, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2] : A \oplus B \vdash \Delta} \oplus L$$

Although more intricate, the evidence for or against an existential follows this same pattern of constructing values in the term and deconstructing them in the coterm. For simplicity, assume that the quantifiers' domain ranges over other types. $\exists X.B$ describes values of type B, which might reference a hidden type X. This kind of information hiding corresponds to modules in a program where the code implementing the module is written with full knowledge of a specific type X, but the client code using the module does not know which type was used for X. To be explicit about the module's hidden choice for X, we can use the (Sage's) constructor form (A, v) which means to produce the value v whose type depends on A. The client (Skeptic) side can unpack a generic value (evidence) of the form (X, y) to run a command (demonstrate a contradiction), which looks like $\tilde{\mu}(X, y).c$. This pair of construction-deconstruction looks like:[2]

[2] The $\exists L$ rule has the additional side condition $X \notin FV(\Gamma \vdash \Delta)$, meaning the type variable X is not found among the free variables of environments Γ and Δ. The side condition makes sure that X stands for a truly generic type parameter, which would be ruined if Γ and Δ constrained X with additional assumptions about it. Similar side conditions weren't needed in $\otimes L$ and $\oplus L$ because ordinary variables x, y cannot be referenced by types in Γ and Δ without dependent types. Alternatively, we could have also introduced yet another environment $\Theta = X, Y, Z, \ldots$ for keeping track of the free type variables in the sequent, as is often done in the type systems in polymorphic languages like System F [22].

$$\frac{\Gamma \vdash v : B\{A/x\} \mid \Delta}{\Gamma \vdash (A, v) : \exists X.B \mid \Delta}\ \exists R \qquad \frac{c : (\Gamma, y : B \vdash \Delta) \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}(X, y).c : \exists X.B \vdash \Delta}\ \exists L$$

3.2 Negative burden of proof as codata

If the positive burden of truth corresponds to constructing values of a data type, then what is the computational interpretation of the negative burden of proof? Applying syntactic duality of our symmetric calculus – that is, flipping the roles of producers v and consumers e in the command $\langle v \| e \rangle$ to get the analogue of $\langle e \| v \rangle$ – leaves us only one answer: *constructing covalues* of a codata type, which are defined in terms of observations rather than values. This corresponds to the evidence constructed by the Skeptic within a negative burden of proof, which has a different shape depending on the proposition A being argued against. Thus, the Skeptic's evidence can be represented by a consumer e of type A.

Consider the basic cases for negative evidence against conjunctions ($A \,\&\, B$) and disjunctions ($A \parr B$). Contrary to before, the evidence against a conjunction comes in one of two forms: either evidence e against A or evidence f against B. In other words, it is a first projection $\pi_1 e$ or second projection $\pi_2 f$ out of a product type $A \,\&\, B$. The evidence against a disjunction instead has just one form, containing both evidence e against A and evidence f against B. Taken together, this is a pair $[e, f]$ – dual to a tuple of values – of the type $A \parr B$. These constructions of consumers are captured by the following typing rules, which resemble the inference rules for $A \,\&\, B$ **false** and $A \parr B$ **false** from Section 2.2:

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \mid \pi_1 e : A \,\&\, B \vdash \Delta}\ \&L_1 \quad \frac{\Gamma \mid f : B \vdash \Delta}{\Gamma \mid \pi_2 f : A \,\&\, B \vdash \Delta}\ \&L_2 \quad \frac{\Gamma \mid e : A \vdash \Delta \quad \Gamma \mid f : B \vdash \Delta}{\Gamma \mid [e, f] : A \parr B \vdash \Delta}\ \parr L$$

If the Skeptic is now constructing concrete evidence, then the Sage must be the one responding to it in some way. This proof of truth involves arguing that the Skeptic cannot possibly argue against the proposition: every potential piece of negative evidence that might be constructed leads to a contradiction. The computational interpretation of the Sage's response corresponds to an object that defines a reaction to every possible observation on it, which can be written via *copattern matching* [1] which *deconstructs* the shape of its observer.

A rebuttal in favor of $A \,\&\, B$ is a process that demonstrates two different contradictions: c_1 which responds to a generic first projection $\pi_1 \alpha : A \,\&\, B$, and c_2 which responds to a generic second projection $\pi_2 \beta : A \,\&\, B$. Instead, a rebuttal in favor of $A \parr B$ responds with just one contradiction c, given a generic $[\alpha, \beta] : A \parr B$ that combines both pieces of negative evidence (α against A and β against B). The two rebuttals in favor of $A \,\&\, B$ and $A \parr B$ are captured by the copattern-matching producers $\mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2)$ and $\mu[\alpha, \beta].c$, respectively, given by these two typing rules:

$$\frac{c_1 : (\Gamma \vdash \alpha : A, \Delta) \quad c_2 : (\Gamma \vdash \beta : B, \Delta)}{\Gamma \vdash \mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2) : A \,\&\, B \mid \Delta}\ \&R \qquad \frac{c : (\Gamma \vdash \alpha : A, \beta : B, \Delta)}{\Gamma \vdash \mu[\alpha, \beta].c : A \parr B \mid \Delta}\ \parr R$$

Universal quantification can be derived mechanically as the dual of existential quantification, where the roles of information hiding have been flipped between the implementor and client. With the polymorphic type $\forall X.B$ – describing values of type B that are generic in type X – it is now the clients using values of type $\forall X.B$ that get to choose X. For example, consider the polymorphic function $\forall X.X \to X$: the callers of this function get to choose the specific type for X – it could be integers, booleans, lists, *etc.*– before passing an argument of that type to receive a returned value of the same type. The implementor which

produces a value of type $\forall X.B$ must instead be generic in X: it cannot know which X was chosen because different clients might all choose different specializations for X. Thus, the implementation (Sage) side can unpack a generic covalue (evidence) of the form $[X, \beta]$ to run a command (demonstrate a contradiction), which looks like $\mu[X, \beta].c$ corresponding to System F's $\Lambda X.v$ [22]. These (de)constructors follow rules dual to $\exists R$ and $\exists L$:

$$\frac{\Gamma \mid e : B\{A/X\} \vdash \Delta}{\Gamma \mid [A, e] : \forall X.B \vdash \Delta}\ \forall L \qquad \frac{c : (\Gamma \vdash \beta : B, \Delta) \quad X \notin FV(\Gamma \vdash \Delta)}{\Gamma \vdash \mu[A, \beta].c : \forall X.B \mid \Delta}\ \forall R$$

3.3 The two dual negations

Now that we have introduced the computational content of both the positive and negative burden of proof, we can finally examine the nature of negation which reverses these two roles. In Section 2.2, we had two different forms of negation: $\ominus A$ is described by positive evidence in favor of it, whereas $\neg A$ is described by negative evidence against it. Following our analogy, $\ominus A$ corresponds to a data type: the Sage's evidence in favor of $\ominus A$, written (e), contains *specific* evidence e against A. The Skeptic then responds by showing why *any* construction of the form $(\alpha) : \ominus A$ leads to a contradiction c, as expressed by these typing rules:

$$\frac{\Gamma \mid e : A \vdash \Delta}{\Gamma \vdash (e) : \ominus A \mid \Delta}\ \ominus R \qquad \frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \mid \tilde{\mu}(\alpha).c : \ominus A \vdash \Delta}\ \ominus L$$

The other negation $\neg A$ is its dual codata type: the Skeptic's evidence against $\neg A$, written $[v]$, contains *specific* evidence v in favor (*i.e.,* producing a value) of A. The Sage then responds by showing why *any* construction of the form $[x] : \neg A$ leads to a contradiction c, as in:

$$\frac{\Gamma \vdash v : A \mid \Delta}{\Gamma \mid [v] : \neg A \vdash \Delta}\ \neg L \qquad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \vdash \mu[x].c : \neg A \mid \Delta}\ \neg R$$

3.4 Proof by contradiction as control

We have talked about many different indirect proofs and (co)terms: those that show how potential constructions lead to a contradiction (*i.e.,* command), rather than giving a concrete construction itself. These include all the coterms which pattern-match on specific values of data types, as well as all the terms which copattern-match on the specific covalues of codata types. But in practical programming languages, we aren't forced to always match on the shape of a value. We can also just give any value a name, as in the expression $\mathbf{let}\ z = v\ \mathbf{in}\ w$ found in many functional languages. What does this look like in our symmetric language? We could generalize coterms like $\tilde{\mu}(x, y).c$ to just the generic $\tilde{\mu}z.c$ which names their input before running a command c (just like $\mathbf{let}\ z = v\ \mathbf{in}\ w$ names v before running w). The dual of the generic $\tilde{\mu}$ is a generic μ: the term $\mu\alpha.c$ names its output before running a command c.[3] The typing rules for these two dual abstractions correspond to the two forms of proof by contradiction from Section 2.2: if assuming A **true** leads to a contradiction, then A **false**; and dually if assuming A **false** leads to a contradiction, then A **true**.

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

Notice how these two rules can be seen as simplifications of matching rules on the left ($\otimes L$, $\oplus L$, $\exists L$) and right ($\& R$, $\parr R$, $\forall R$) to not depend on the structure of the abstracted type.

[3] The term $\mu\alpha.c$ gets the simpler name because it came first in Parigot's $\lambda\mu$-calculus [31] for classical logic. The dual coterm $\tilde{\mu}x.c$ was derived after in the sequent calculus [4] for call-by-value computation.

Although generic μ and $\tilde{\mu}$ might seem innocuous, they can have a serious impact on computational power. Whereas $\tilde{\mu}$ corresponds to the pervasive (and relatively innocent) feature of value-naming as expressed by basic let-bindings, μ corresponds to a notion of *control effect* equivalent to Scheme's `call/cc` operator [7]. In terms of a logic, μ can also increase the propositions that can be proven true.

For example, consider the two different interpretations of the law of the excluded middle from Section 2.3. The negative version, $A \parr \neg A$ corresponds to the term $\mu[\alpha, [x]].\langle x \| \alpha \rangle$ written in terms of nested copatterns. Intuitively, this term is isomorphic to the identity function, $\lambda x.x : A \to A$, and it's typing derivation (*i.e.,* proof) is given like so:

$$\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \alpha : A, \beta : \neg A}\, VarR \quad \dfrac{}{x : A \mid \alpha : A \vdash \alpha : A, \beta : \neg A}\, VarL}{\langle x \| \alpha \rangle : (x : A \vdash \alpha : A, \beta : \neg A)}\, Cut}{\vdash \mu[x].\langle x \| \alpha \rangle : \neg A \mid \alpha : A, \beta : \neg A}\, \neg R \quad \dfrac{}{\mid \beta : \neg A \vdash \alpha : A, \beta : \neg A}\, VarL}{\dfrac{\langle \mu[x].\langle x \| \alpha \rangle \| \beta \rangle : (\vdash \alpha : A, \beta : \neg A)}{\vdash \mu[\alpha, \beta].\langle \mu[x].\langle x \| \alpha \rangle \| \beta \rangle : A \parr \neg A \mid}\, \parr R}\, Cut$$

Notice how – in addition to the core Cut and Var rules – we only use the type-specific matching rules for $\parr$ and $\neg$ here. There is no need to resort to the generic $ActR$ or $ActL$.

In contrast, the positive law of the excluded middle, $A \oplus \neg A$, corresponds to the term $\mu\alpha.\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle$. Notice the use of the generic $\mu\alpha \ldots$, requiring the $ActR$ rule in its typing derivation (omitting the names for Var and Cut rules):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \vdash x : A \mid \alpha : A \oplus \neg A}}{x : A \vdash \iota_1 x : A \oplus \neg A \mid \alpha : A \oplus \neg A}\, \oplus R_1 \quad \dfrac{}{x : A \mid \alpha : A \oplus \neg A \vdash \alpha : A \oplus \neg A}}{\langle \iota_1 x \| \alpha \rangle : (x : A \vdash \alpha : A \oplus \neg A)}\, Cut}{\vdash \mu[x].\langle \iota_1 x \| \alpha \rangle : \neg A \mid \alpha : A \oplus \neg A}\, \neg R}{\vdash \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle : A \oplus \neg A \mid \alpha : A \oplus \neg A}\, \oplus R_2 \quad \dfrac{}{\mid \alpha : A \oplus \neg A \vdash \alpha : A \oplus \neg A}}{\dfrac{\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle : (\vdash \alpha : A \oplus \neg A)}{\vdash \mu\alpha.\langle \iota_2 \mu[x].\langle \iota_1 x \| \alpha \rangle \| \alpha \rangle : A \oplus \neg A \mid}\, ActR}$$

Whereas $A \parr \neg A$ is like the simple identity function, the term of type $A \oplus \neg A$ invokes a serious manipulation of control flow. Intuitively, this term corresponds to the Scheme expression:

```
(call/cc (lambda (alpha)
           (cons 2 (lambda (x) (alpha (cons 1 x))))))
```

Here, the "time travel" needed to implement the positive law of the excluded middle is expressed by the control operator `call/cc`. Before doing anything else, the current continuation is saved (in `alpha`), just in case we need to change our answer. Then, we first return the second option (represented by a numerically-labeled cons-cell `(cons 2 ...)`) containing a function. If that function is ever called with a value `x` of type A, then we invoke the continuation `alpha` which rolls back the clock and lets us change our answer to the first option `(cons 1 x)`: deftly giving back the value we were just given.

3.5 A symmetric system of computation

Thus far, we have only discussed how to build objects (producers and consumers) following this two-sided method of interaction. That alone does not tell us how to compute; we also need to know how the interaction unfolds over time.

$$
\begin{array}{llll}
(\beta_\otimes) & \langle (v,w) \| \tilde{\mu}(x,y).c \rangle = \langle v \| \tilde{\mu}x.\langle w \| \tilde{\mu}y.c \rangle \rangle & (\eta_\otimes) & \tilde{\mu}(x,y).\langle (x,y) \| \alpha \rangle = \alpha \quad (\alpha : A \otimes B) \\
(\beta_\oplus) & \langle \iota_i v \| \tilde{\mu}[\iota_i x_i.c_i] \rangle = \langle v \| \tilde{\mu}x_i.c_i \rangle & (\eta_\oplus) & \tilde{\mu}[\iota_i x_i.\langle \iota_i x_i \| \alpha \rangle] = \alpha \quad (\alpha : A \oplus B) \\
(\beta_\exists) & \langle (A,v) \| \tilde{\mu}(X,y).c \rangle = \langle v \| \tilde{\mu}y.c\{A/X\} \rangle & (\eta_\exists) & \tilde{\mu}(X,y).\langle (X,y) \| \alpha \rangle = \alpha \quad (\alpha : \exists X.B) \\
(\beta_\ominus) & \langle (e) \| \tilde{\mu}(\alpha).c \rangle = \langle \mu\alpha.c \| e \rangle & (\eta_\ominus) & \tilde{\mu}(\beta).\langle (\beta) \| \alpha \rangle = \alpha \quad (\alpha : \ominus A) \\
(\beta_\&) & \langle \mu(\pi_i \alpha_i.c_i) \| \pi_i e \rangle = \langle \mu\alpha_i.c_i \| e \rangle & (\eta_\&) & \mu(\pi_i \alpha_i.\langle x \| \pi_i \alpha_i \rangle) = x \quad (x : A \,\&\, B) \\
(\beta_\parr) & \langle \mu[\alpha,\beta].c \| [e,f] \rangle = \langle \mu\alpha.\langle \mu\beta.c \| f \rangle \| e \rangle & (\eta_\parr) & \mu[\alpha,\beta].\langle x \| [\alpha,\beta] \rangle = x \quad (x : A \parr B) \\
(\beta_\forall) & \langle \mu[X,\beta].c \| [A,e] \rangle = \langle \mu\beta.c\{A/x\} \| e \rangle & (\eta_\forall) & \mu[X,\beta].\langle x \| [X,\beta] \rangle = x \quad (x : \forall X.B) \\
(\beta_\neg) & \langle \mu[x].c \| [v] \rangle = \langle v \| \tilde{\mu}x.c \rangle & (\eta_\neg) & \mu[y].\langle x \| [y] \rangle = x \quad (x : \neg A)
\end{array}
$$

Plus compatibility, symmetry, reflexivity, and transitivity.

Figure 3 Equational reasoning for (co)pattern matching in the dual core sequent calculus.

One of the simplest ways of viewing the computation of interaction is through the axioms which characterize the equality of expressions. These axioms, given in Figure 3, come in two main forms. The β family of laws say what happens when a matching term and coterm of a type meet up in a command. For example, when the tuple construction (v, w) meets up with a tuple deconstruction $\tilde{\mu}(x, y).c$, the interaction can be simplified with $\beta_\otimes$ by matching the structure of (v, w) with the pattern (x, y), and bind v to x and w to y (with the help of the generic $\tilde{\mu}$). When there is a choice like in the sum type $A \oplus B$, then the appropriate response is selected by $\beta_\oplus$. When the right construction $\iota_2 v$ meets up with the sum deconstruction $\tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2]$, then the result is c_2 with v bound to y from the matching pattern $\iota_2 y$. The same kind of matching happens for the codata types, but with the roles reversed. Instead, it is the coterm side that is constructed, like the second projection $\pi_2 e$ of a product type $A \,\&\, B$, and the term side selects a response, like the term $\mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2)$ which matches with $\pi_2 e$ by binding e to β and running c_2 as per $\beta_\&$. Note that the β rules for both negations ($\ominus$ and $\neg$) end up swapping the two sides of a command.

The other family of laws are the η axioms, which give us a notion of *extensionality*. In each case, the η axioms say that deconstructing a structure and reconstructing it exactly as it was before does nothing. The side where this simplification applies depends on the type of the structure in question. For data types, the consumer does the deconstructing, so the $\eta_\otimes$, $\eta_\oplus$, $\eta_\exists$, and $\eta_\ominus$ axioms apply to a generic unknown coterm – represented by the covariable α – waiting to receive its input. Whereas for codata types, the producer does the deconstructing, so the $\eta_\&$, $\eta_\parr$, $\eta_\forall$, and $\eta_\neg$ axioms apply to a generic unknown term – represented by the variable x – waiting to receive an output request.

But equational axioms are quite far from a real implementation in a machine. They give the ultimate freedom of choice on where the rules can apply (in any context, due to compatibility) and in which direction (due to symmetry). In reality, a machine implementation will make a (deterministic) choice on the next step to take, and always move forward. This is modeled by the operational semantics given in Figure 4, where each step $c \mapsto c'$ applies *exactly* to the top of the command itself. The happy coincidence of a dual calculus based on the sequent calculus is that its operational semantics *is* an abstract machine [10], since there is never a search for the next redex which is always found at the top. Thus, this style of calculus is a good framework for studying the low-level details of computation needed to implement languages in real machines.

Call-by-value definition of values (V_+) and covalues (E_+):

$$\begin{aligned} \mathit{Value}_+ \ni V_+, W_+ &::= x \mid (V_+, W_+) \mid \iota_1 V_+ \mid \iota_2 V_+ \mid (A, V_+) \mid (E_+) \\ &\quad \mid \mu(\pi_1 \alpha.c_1 \mid \pi_2 \beta.c_2) \mid \mu[\alpha, \beta].c \mid \mu[X, \beta].c \mid \mu[x].c \\ \mathit{CoValue}_+ \ni E_+, F_+ &::= e \end{aligned}$$

Call-by-name definition of values (V_-) and covalues (E_-):

$$\begin{aligned} \mathit{Value}_- \ni V_-, W_- &::= v \\ \mathit{CoValue}_- \ni E_-, F_- &::= \alpha \mid [E_-, F_-] \mid \pi_1 E_- \mid \pi_2 E_- \mid [A, E_-] \mid [V_-] \\ &\quad \mid \tilde{\mu}[\iota_1 x.c_1 \mid \iota_2 y.c_2] \mid \tilde{\mu}[x, y].c \mid \tilde{\mu}(X, y).c \mid \tilde{\mu}(\alpha).c \end{aligned}$$

Reduction rules for call-by-value ($s = +$) and call-by-name ($s = -$) evaluation.

$$\begin{array}{llll} (\beta^s_\otimes) & \langle (V_s, W_s) \| \tilde{\mu}(x, y).c \rangle \mapsto c\{V_s/x, W_s/y\} & (\varsigma^s_\otimes) & \langle (v, w) \| E_s \rangle \mapsto \langle v \| \tilde{\mu}x.\langle w \| \tilde{\mu}y.\langle (x, y) \| E_s \rangle \rangle \rangle \\ (\beta^s_\oplus) & \langle \iota_i V_s \| \tilde{\mu}[\iota_i x_i.c_i] \rangle \mapsto c_i\{V_s/x_i\} & (\varsigma^s_\oplus) & \langle \iota_i v \| E_s \rangle \mapsto \langle v \| \tilde{\mu}x.\langle \iota_i x \| E_s \rangle \rangle \\ (\beta^s_\exists) & \langle (A, V_s) \| \tilde{\mu}(X, y).c \rangle \mapsto c\{A/X, V_s/y\} & (\varsigma^s_\exists) & \langle (A, v) \| E_s \rangle \mapsto \langle v \| \tilde{\mu}x.\langle (A, x) \| E_s \rangle \rangle \\ (\beta^s_\ominus) & \langle (E_s) \| \tilde{\mu}(\alpha).c \rangle \mapsto c\{E_s/\alpha\} & (\varsigma^s_\ominus) & \langle (e) \| E_s \rangle \mapsto \langle \mu\alpha.\langle (\alpha) \| E_s \rangle \| e \rangle \\ (\beta^s_\&) & \langle \mu(\pi_i \alpha_i.c_i) \| \pi_i E_s \rangle \mapsto c_i\{E_s/\alpha_i\} & (\varsigma^s_\&) & \langle V_s \| \pi_i e \rangle \mapsto \langle \mu\alpha.\langle V_s \| \pi_i \alpha \rangle \| e \rangle \\ (\beta^s_\parr) & \langle \mu[\alpha, \beta].c \| [E_s, F_s] \rangle \mapsto c\{E_s/\alpha, F_s/\beta\} & (\varsigma^s_\parr) & \langle V_s \| [e, f] \rangle \mapsto \langle \mu\alpha.\langle \mu\beta.\langle V_s \| [\alpha, \beta] \rangle \| f \rangle \| e \rangle \\ (\beta^s_\forall) & \langle \mu[X, \beta].c \| [A, E_s] \rangle \mapsto c\{A/x, E_s/\beta\} & (\varsigma^s_\forall) & \langle V_s \| [A, e] \rangle \mapsto \langle \mu\alpha.\langle V_s \| [A, \alpha] \rangle \| e \rangle \\ (\beta^s_\neg) & \langle \mu[x].c \| [V_s] \rangle \mapsto c\{V_s/x\} & (\varsigma^s_\neg) & \langle V_s \| [v] \rangle \mapsto \langle v \| \tilde{\mu}x.\langle V_s \| [x] \rangle \rangle \end{array}$$

In each of the ς^s rules, assume that (v, w), $\iota_i v$, (A, v), and (e) are not in Value_s, respectively, and $\pi_i e$, $[e, f]$, $[A, e]$, and $[v]$ are not in $\mathit{CoValue}_s$, respectively.

Figure 4 Operational semantics for (co)pattern matching in the dual core sequent calculus.

The difference between the β rules in the operational semantics (Figure 4) from the ones in the equational theory (Figure 3) is that the operational rules completely resolve the matching in one step. Rather than forming new bindings with generic μs and $\tilde{\mu}$s, the components of the construction (on either side) are substituted directly for the (co)pattern variables. To do so, we need to use a notion of *evaluation* strategy which informs us which terms can be substituted for variables (we call these *values*) and which coterms can be substituted for covariables (we call these *covalues*, which represent *evaluation contexts*).

Call-by-value evaluation simplifies terms first before substituting them for variables, so it has a quite restrictive notion of value (V_+) for constructed values like (V_+, W_+) and $\iota_i V_+$, but all coterms represent call-by-value evaluation contexts (hence every e is substitutable). Dually, call-by-name evaluation will substitute any term for a variable (hence a value V_- could be any v), but only certain coterms represent evaluation contexts: for example, the projection $\pi_1 E_-$ only represents an evaluation context because E_- does, but $\pi_1 e$ does not when e does not need its input yet.

The other cases of reduction are handled by the ς rules, which say what to do when a construction isn't a (co)value yet. In a call-by-value language like OCaml, the term `(1+2, 3+4)` first evaluates the two components before returning the pair value `(3, 7)`. This scenario is handled by the $\varsigma^+_\otimes$ step, which lifts the two computations in the tuple to the top of the command, replacing $\langle (1 + 2, 3 + 4) \| \alpha \rangle$ with $\langle 1 + 2 \| \tilde{\mu}x.\langle 3 + 4 \| \tilde{\mu}y.\langle (x, y) \| \alpha \rangle \rangle \rangle$; now we know that the next step is to simplify $1 + 2$ before binding it to x.

Equational axioms for $\mu\tilde{\mu}$ in both call-by-value ($s = +$) and call-by-name ($s = -$) reduction:

$$(\beta^s_\mu) \quad \langle \mu\alpha.c \| E_s \rangle = c\{E_s/\alpha\} \qquad (\eta_\mu) \quad \mu\alpha\langle v \| \alpha \rangle = v \qquad (\alpha \notin FV(v))$$
$$(\beta^s_{\tilde{\mu}}) \quad \langle V_s \| \tilde{\mu}x.c \rangle = c\{V_s/x\} \qquad (\eta_{\tilde{\mu}}) \quad \tilde{\mu}x.\langle x \| e \rangle = e \qquad (x \notin FV(e))$$

Operational semantics for $\mu\tilde{\mu}$ in both call-by-value ($s = +$) and call-by-name ($s = -$):

$$(\beta^s_\mu) \quad \langle \mu\alpha.c \| E_s \rangle \mapsto c\{E_s/\alpha\} \qquad (\beta^s_{\tilde{\mu}}) \quad \langle V_s \| \tilde{\mu}x.c \rangle \mapsto c\{V_s/x\}$$

Figure 5 Rules for data flow and control flow in the dual core sequent calculus.

data $A \oplus B$ **where**	**data** $A \otimes B$ **where**	**data** $\ominus A$ **where**	**data** $\exists F$ **where**
$\iota_1 : A \vdash A \oplus B \mid$	$(\_,\_) : A, B \vdash A \otimes B \mid$	$(\_) : \vdash \ominus A \mid A$	$(\_,\_) : F\ A \vdash \exists F \mid$
$\iota_2 : B \vdash A \oplus B \mid$			
codata $A \,\&\, B$ **where**	**codata** $A \parr B$ **where**	**codata** $\neg A$ **where**	**codata** $\forall F$ **where**
$\pi_1 : \mid A \,\&\, B \vdash A$	$[\_,\_] : \mid A \parr B \vdash A, B$	$[\_] : A \mid \neg A \vdash$	$[\_,\_] : \mid \forall F \vdash F\ A$
$\pi_2 : \mid A \,\&\, B \vdash B$			

Figure 6 (Co)Data declarations of the core connectives and quantifiers.

The last piece of the puzzle is what to do with the generic μs and $\tilde{\mu}$s. Fortunately, these are simpler than the individual rules for the various connectives and quantifiers. A coterm $\tilde{\mu}x.c$ binds its partner to x wholesale, without inspecting it further, and likewise $\mu\alpha.c$ binds its entire partner to α. These two actions are captured by $\beta^s_{\tilde{\mu}}$ and β^s_μ in Figure 5 which, like the rules in Figure 4, are careful to only substitute values and covalues. This careful consideration of substitutability prevents the fundamental critical pair between μ and $\tilde{\mu}$:

$$c_1\{\tilde{\mu}x.c_2/\alpha\} \leftarrow\!\!\mapsto_{\beta^+_\mu} \langle \mu\alpha.c_1 \| \tilde{\mu}x.c_2 \rangle \mapsto\!\!\rightarrow_{\beta^-_{\tilde{\mu}}} c_2\{\mu\alpha.c_1/x\}$$

This restriction is necessary for both the equational axioms as well as the operational reduction steps (which are identical in name and result). These restrictions ensure that the operational semantics is *deterministic* and the equational theory is *consistent* (*i.e.,* not all commands are equated). Similarly, the η axioms for μ and $\tilde{\mu}$ say that binding a (co)variable just to use it immediately does nothing. While the η laws in Figures 3 and 5 are not themselves necessary for computation, they do give us a hint on how to keep going when we might get stuck. Specifically, the ς rules from Figure 4 can be derived from $\beta\eta$ equality, showing that these two families of axioms are *complete* for specifying computation [8].

▶ **Observation 2.** *If* $c \mapsto_{\beta\varsigma} c'$ *then* $c =_{\beta\eta} c'$.

3.6 (Co)Data in the wild

The connectives from Sections 3.1 and 3.2 originally arose from the field of logic, but that doesn't mean they are disconnected from programming. Indeed, the concept of data and codata they embody can be found to some degree in programming languages that are already in wide use today, although not in their full generality.

First, we can imagine a mechanism for declaring new connectives as (co)data types which list their patterns of construction. For example, all the connectives we have seen so far are given declarations in Figure 6. Each (term or coterm) constructor is given a type signature in the form of a sequent: input parameters are to the left of $\vdash$, and output parameters are to the right. For data types, constructors build a value returned as output, whose type is given in a special position to the right of the turnstyle between it and the vertical bar (*i.e.,* the A in $\cdots \vdash A \mid \ldots$). Dually for codata types, constructors build a covalue that takes an input, whose type is given in the special position on the left between the turnstyle and the vertical bar (*i.e.,* the A in $\cdots \mid A \vdash \ldots$).

This notion of data type corresponds to *algebraic data types* in typed functional languages. For example, the declarations for $A \oplus B$ and $A \otimes B$ correspond to the following Haskell declarations for sum (`Either`) and pair (`Both`) types:

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

```
data Both a b where
  Pair :: a -> b -> Both a b
```

Even the existential quantifier corresponds to a Haskell data type, whose constructor introduces a new generic type variable `a` not found in the return type `Exists f`.

```
data Exists f where Pack :: f a -> Exists f
```

However, the negation $\ominus A$ does not correspond to any data type in Haskell. That's because $\ominus A$'s constructor requires *two outputs* (notice the two types to the right of the turnstyle: the main $\ominus A$ plus the additional output parameter A). This requires some form of continuations or control effects, which is not available in a pure functional language like Haskell.

The dual notion of codata type corresponds to *interfaces* in typed object-oriented languages. For example, the declaration for $A \,\&\, B$ corresponds to the following Java interface for a generic `Product`:

```
interface Product<A,B> { A first(); B second(); }
```

Java's type system is not strong enough to capture quantifiers.[4] However, if its type system were extended so that generic types could range over other parameterized generic types, we could declare a `Forall` interface corresponding to the $\forall$ quantifier:

```
interface Forall<F> { F<A> specialize<A>(); }
```

Unfortunately, the types $A \parr B$ and $\neg A$ suffer the same fate as $\ominus A$; their constructors require a number of outputs different from 1: $[\alpha, \beta]$ has two outputs (both α and β), and $[x]$ has no outputs (x is an input, not an output). So they cannot be represented in Java without added support for juggling multiple continuations.

The possibilities for modeling additional information in the constructions of the type – representing *pre-* and *post-conditions* in a program – become much more interesting when we look at indexed (co)data types. For a long time, functional languages have been using *generalized algebraic data types* (GADTs), also known as *indexed data types*, that allow each constructor return a value with a more constrained version of that type. The classic example

[4] Unlike Haskell, Java does not support generic type variables with higher kinds. The Haskell declaration of `Exists f` relies on the fact that the type variable `f` has the kind `* -> *`, *i.e.,* `f` stands for a *function* that turns one type into another.

of indexed data types is representing expression trees with additional typing information. For example, here is a data type representing a simple expression language with literal values, plus and minus operations on numbers, an "is zero" test, and an if-then-else expression:

$$
\begin{array}{rl}
\textbf{data}\ \texttt{Expr}\ X\ \textbf{where} & \\
\quad \texttt{Literal} : & X \vdash \texttt{Expr}\ X \quad | \\
\quad \texttt{Plus} : & \texttt{Expr Int},\ \texttt{Expr Int} \vdash \texttt{Expr Int} \quad | \\
\quad \texttt{Minus} : & \texttt{Expr Int},\ \texttt{Expr Int} \vdash \texttt{Expr Int} \quad | \\
\quad \texttt{IsZero} : & \texttt{Expr Int} \vdash \texttt{Expr Bool} \quad | \\
\quad \texttt{IfThenElse} : & \texttt{Expr Bool},\ \texttt{Expr}\ X,\ \texttt{Expr}\ X \vdash \texttt{Expr}\ X \quad |
\end{array}
$$

The type parameter X acts as an index, and it lets us constrain the types of values an expression can represent. For example, `IsZero` expects an integer and returns a boolean. This lets us write a typed evaluation function $\mathit{eval} : \texttt{Expr}\ X \to X$, and not worry about mistyped edge cases because the type system rules out poorly-constructed expressions.

The dual of indexed data types are *indexed codata types*, which let us constrain each observation of the codata type to only accept certain inputs which model another form of pre- and post-conditions [18]. For example, we can embed a basic socket protocol – for sending and receiving information along an address – inside this indexed codata type:

$$
\begin{array}{rl}
\textbf{codata}\ \texttt{Socket}\ X\ \textbf{where} & \\
\quad \texttt{Bind} : \texttt{String} & \mid \texttt{Socket Raw} \vdash \texttt{Socket Bound} \\
\quad \texttt{Connect} : & \mid \texttt{Socket Bound} \vdash \texttt{Socket Live} \\
\quad \texttt{Send} : \texttt{String} & \mid \texttt{Socket Live} \vdash () \\
\quad \texttt{Receive} : & \mid \texttt{Socket Live} \vdash \texttt{String} \\
\quad \texttt{Close} : & \mid \texttt{Socket Live} \vdash ()
\end{array}
$$

A new `Socket` starts out as `Raw`. We can `Bind` a `Socket Raw` to an address, after which it is `Bound` and can be `Connect`ed to make it `Live`. A `Socket Live` represents a connection we can use to `Send` and `Receive` messages, and is discarded by a `Close`.[5]

4 Applications of Duality

So a constructive view of symmetric classical logic gives us a dual language for expressing computation as interaction. Does this form of duality have any application in the broader scope of programs? Yes! Let's look at a few examples where computational duality can be put into action for solving problems in programming.

4.1 Functions as Codata

There is a delicate trilemma in the theory of the untyped λ-calculus: one cannot combine non-strict weak-head reduction, function extensionality, and computational effects. The specific reduction we are referring to follows two properties: "non-strict" means that functions are called without evaluating their arguments first, and "weak-head" means that evaluation stops at a λ-abstraction. Function extensionality is captured by the η law – $\lambda x.\ f\ x = f$ – from the foundation of the λ-calculus. And finally effects could be anything – from mutable

[5] This interface can be further improved by linear types, which ensure that outdated states of the `Socket` cannot be used, and forces the programmer to properly `Close` a `Socket` instead of leaving it hanging.

state to exceptions – but for our purposes, non-termination introduced by general recursion is enough. That is to say, the infinite loop $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ already expressible in the "pure" untyped λ-calculus counts as an effect.

So what is the problem when all three are combined in the same calculus? The conflict arises when we observe a λ-abstraction as the final result of evaluation. Because of weak-head reduction, any λ-abstraction counts as a final result, including $\lambda x.\Omega x$. Because of extensionality, the η law says that $\lambda x.\Omega x$ is equivalent to Ω. Taken together, this means that a program that ends immediately is the same as one that loops forever: an inconsistency.

4.1.1 Efficient head reduction

One way to fix the trilemma is to change from weak-head reduction to *head reduction*. With head reduction, evaluation no longer stops at a λ-abstraction. Instead, head reduction looks inside of λs to keep going until a head-normal form of the shape $\lambda x_1 \ldots \lambda x_n.x_i\ M_1 \ldots M_m$ is found. But going inside λs means that evaluation has to deal with open terms, *i.e.,* terms with free variables in them. How can we perform head reduction efficiently, when virtually all efficient implementations assume that evaluation only handles closed terms?

Our idea is to look at functions as yet another form of *codata*, just like $A \parr B$ and $A \,\&\, B$. Following the other declarations in Figure 6, the type of functions can be defined as:

$$\textbf{codata}\ A \to B\ \textbf{where}\ \_ \cdot \_ : A \mid A \to B \vdash B$$

This says that the coterm which observes a function of type $A \to B$ has the form of a *call stack* $v \cdot e$, where v is the argument (of type A), and e represents a kind of "return pointer" (expecting the returned B). The stack-like nature can be seen in the way a chain of function arrows requires a stack of arguments; for instance a coterm of type $Int \to Int \to Int \to Int$ has the stack shape $1 \cdot 2 \cdot 3 \cdot \alpha$, where α is a place to put the result.

Rather than the usual λ-abstraction, the codata view suggests that we can instead write functions in terms of copattern matching: $\mu[x \cdot \beta].c$ is a function of type $A \to B$ where c is the command to run in the scope of the (co)variables $x : A$ and $\beta : B$. Both forms of writing functions are equivalent to one another (via general μ):

$$\mu[x \cdot \beta].c = \lambda x.\mu\beta.c \qquad \lambda x.v = \mu[x \cdot \beta].\langle v \| \beta \rangle \qquad (\beta \notin FV(v))$$

This way, the main rule for reducing a call-by-name function call is to match on the structure of a call stack (recall from Section 3.5 that call-by-name covalues are restricted to E_-, so covalue call stacks have the form $v \cdot E_-$ in call-by-name) like so:

$$\langle \mu[x \cdot \beta].c \| v \cdot E_- \rangle \mapsto c\{v/x, E_-/\beta\}$$

But what happens when we encounter a function at the top-level? This is represented by the command $\langle \mu[x \cdot \beta].c \| \mathtt{tp} \rangle$ where $\mathtt{tp}$ is a constant standing in for the empty, top-level context. Normally, we would be stuck, so instead lets look at functions from the other side. A call stack $v \cdot E_-$ is similar to a pair (v, w). In some programming languages, we access a pair by matching on its structure (analogous to $\tilde{\mu}(x, y).c$). But in other languages, we are given primitive projections for accessing its fields. We can make the same change with functions: rather than matching on the structure of a call (with $\mu[x \cdot \beta].c$ or $\lambda x.v$), we can instead *project out of a call stack* [26]. The projection $\mathtt{arg}[v \cdot E_-]$ gives us the argument v and $\mathtt{ret}[v \cdot E_-]$ gives us the return pointer E_-. These two projections let us keep going when a function reaches the top level, by projecting the argument and return pointer out of $\mathtt{tp}$:

$$\langle \mu[x \cdot \beta].c \| \mathtt{tp} \rangle \mapsto c\{\mathtt{arg}\,\mathtt{tp}\,/x, \mathtt{ret}\,\mathtt{tp}\,/\beta\}$$

This goes "inside" the function, and yet there are no free variables in sight. Instead, the would-be free x is replaced with the placeholder $\mathtt{arg}\,\mathtt{tp}$, and we get a *new* "top-level" context $\mathtt{ret}\,\mathtt{tp}$, which stands for the context expecting the result of an implicit call with $\mathtt{arg}\,\mathtt{tp}$.

As we keep going, we may return another function to $\mathtt{ret}\,\mathtt{tp}$, and the process continues with the new placeholder argument $\mathtt{arg}[\mathtt{ret}\,\mathtt{tp}]$ and the next top-level $\mathtt{ret}[\mathtt{ret}\,\mathtt{tp}]$. Rewriting these rules in terms of the more familiar λ-abstractions, we get the following small abstract machine for *closed* head reduction, which says what to do when a function is called (with $w \cdot E_-$) or returned to any of the series of "top-level" contexts ($\mathtt{ret}^n\, E_-$):

$$\begin{aligned}
\langle v\ w \| E_- \rangle &\mapsto \langle v \| w \cdot E_- \rangle \\
\langle \lambda x.v \| w \cdot E_- \rangle &\mapsto \langle v\{w/x\} \| E_- \rangle \\
\langle \lambda x.v \| \mathtt{ret}^n\,\mathtt{tp} \rangle &\mapsto \langle v\{\mathtt{arg}[\mathtt{ret}^n\,\mathtt{tp}]/x\} \| \mathtt{ret}^{n+1}\,\mathtt{tp} \rangle
\end{aligned}$$

For example, the η-expansion of the infinite loop Ω also loops forever, instead of stopping:

$$\langle \lambda x.\Omega x \| \mathtt{tp} \rangle \mapsto \langle \Omega\ (\mathtt{arg}\,\mathtt{tp}) \| \mathtt{ret}\,\mathtt{tp} \rangle \mapsto \langle \Omega \| \mathtt{arg}\,\mathtt{tp} \cdot \mathtt{ret}\,\mathtt{tp} \rangle \mapsto\!\!\!\!\mapsto \langle \Omega \| \mathtt{arg}\,\mathtt{tp} \cdot \mathtt{ret}\,\mathtt{tp} \rangle \dots$$

4.1.2 Effective confluence

A similar issue arises when we consider confluence of the reduction theory. In particular, the call-by-name version of η for functions can be expressed as simplifying the deconstruction-reconstruction detour $\mu[x \cdot \beta].\langle v \| x \cdot \beta \rangle \to_{\eta^-_\to} v$, similar to Figure 3.[6] We might expect that $\beta\eta$ reduction is now confluent like it is in the λ-calculus. Unfortunately, it is not, due to a critical pair between function extensionality and a general μ (_ stands for an unused (co)variable):[7]

$$\mu\_.c \leftarrow_{\eta^-_\to} \mu[x \cdot \beta].\langle \mu\_.c \| x \cdot \beta \rangle \to_{\beta^-_\mu} \mu[x \cdot \beta].c$$

Can we restore confluence of function extensionality in the face of control effects? Yes! The key to eliminating this critical pair is to replace the $\eta^-_\to$ rule with an alternative extensionality rule provided by viewing functions as codata types, equipped with projections out of their constructed call stacks. Under this view, every function is equivalent to a $\mu\alpha.c$, where $\mathtt{arg}\,\alpha$ replaces the argument, and $\mathtt{ret}\,\alpha$ replaces the return continuation. Written as a reduction that replaces copatterns with projections, we have:

$$\mu[x \cdot \beta].c \to_{\mu_\to} \mu\alpha.c\{\mathtt{arg}\,\alpha/x, \mathtt{ret}\,\alpha/\beta\}$$

Analogously, the $\mu_\to$ rule can be understood in terms of the ordinary λ-abstraction as $\lambda x.v \to \mu\alpha.\langle v\{\mathtt{arg}\,\alpha/x\} \| \mathtt{ret}\,\alpha \rangle$. If all functions immediately reduce to a general μ, then how can we execute function calls? The steps of separating the argument and the result are done by the rules for projection, which have their own form of β-reduction along with a different extensionality rule $surj_\to$ capturing the *surjective pair* property of call stacks:

$$\mathtt{arg}[v \cdot E_-] \to_{\beta_{\mathtt{arg}}} v \qquad \mathtt{ret}[v \cdot E_-] \to_{\beta_{\mathtt{ret}}} E_- \qquad [\mathtt{arg}\,E_-] \cdot [\mathtt{ret}\,E_-] \to_{surj_\to} E_-$$

The advantage of these rules is that they are confluent in the presence control effects [27]. Even though surjective pairs can cause non-confluence troubles in general [28], the coarse distinction between terms and coterms is enough to resolve the problem for call-by-name call stacks. Moreover, these rules are strong enough to simulate the λ-calculus' $\beta\eta$ laws:

---

[6] This is the call-by-name version of $\mu[x \cdot \beta].\langle y \| x \cdot \beta \rangle \to_{\eta_\to} y$ because we have substituted a call-by-name value $v \in \mathit{Value}_-$ for the variable y.

[7] Note, this is not just a problem with copatterns; the same issue arises in Parigot's $\lambda\mu$-calculus with ordinary λ-abstractions and η law: $\mu\_.c \leftarrow \lambda x.(\mu\_.c)\ x \to \lambda x.\mu\_.c$.

$$(\lambda x.v)\ w = \mu\alpha.\langle \mu[x \cdot \beta].\langle v \| \beta \rangle \| w \cdot \alpha \rangle \to_{\mu_\to} \mu\alpha.\langle \mu\gamma.\langle v\{\mathtt{arg}\,\gamma/x\} \| \mathtt{ret}\,\gamma \rangle \| w \cdot \alpha \rangle$$
$$\to_{\beta^-_\mu} \mu\alpha.\langle v\{\mathtt{arg}[w \cdot \alpha]/x\} \| \mathtt{ret}[w \cdot \alpha] \rangle \twoheadrightarrow_{\beta_{\mathtt{ret}}\beta_{\mathtt{arg}}} \mu\alpha.\langle v\{w/x\} \| \alpha \rangle \to_{\eta_\mu} v\{w/x\}$$
$$\lambda x.(v\ x) = \mu[x \cdot \beta].\langle v \| x \cdot \beta \rangle \to_{\mu_\to} \mu\alpha.\langle v \| [\mathtt{arg}\,\alpha] \cdot [\mathtt{ret}\,\alpha] \rangle \to_{surj_\to} \mu\alpha.\langle v \| \alpha \rangle \to_{\eta_\mu} v$$

4.2 Loops in Types, Programs, and Proofs

Thus far, we've only talked about *finite* types of information: putting together a fixed number of things. However, real programs are full of loops. Many useful types are self-referential, letting them model information whose size is bounded but arbitrarily large (like lists and trees), or whose size is completely unbounded (like infinite streams). Programs using these types need to be able to loop over arbitrarily large data sets, and generate infinite objects in streams. Once those loops are introduced, reasoning about programs becomes much harder. Let's look at how duality can help us understand the least understood loops in types, programs, and proofs.

4.2.1 (Co)Recursion

Lists and trees – which cover structures that could be any size, as long as they're finite – are modeled by the familiar concept of *inductive data types* found in all mainstream, typed functional programming languages. The dual of these are *coinductive codata types*, which is a relatively newer feature that is finding its way into more practical languages for programming and proving. We already saw instances of both of these as `Expr` and `Socket` from Section 3.6. The canonical examples of (co)inductive (co)data are the types for natural numbers and infinite streams, which are defined like so:

data `Nat` **where**	**codata** `Stream` X **where**
`Zero` : $\vdash$ `Nat` |	`Head` : | `Stream` $X \vdash X$
`Succ` : `Nat` $\vdash$ `Nat` |	`Tail` : | `Stream` $X \vdash$ `Stream` X

The recursive nature of these two types are in the fact that they have constructors that take parameters of the type being declared: `Succ` takes a `Nat` as input before building a new `Nat`, whereas `Tail` consumes a `Stream` X to produce a new `Stream` X as output.

To program with inductive types, functional languages allow programmers to write recursive functions that match on the structure of its argument. For example, here is a definition of the addition function *plus*:

$$plus\ \mathtt{Zero}\ x = x \qquad\qquad plus\ (\mathtt{Succ}\ y)\ x = plus\ y\ (\mathtt{Succ}\ x)$$

We know that this function is well-founded – that is, it always terminates on any input – because it's structurally recursive: the first argument shown in red shrinks on each recursive call, where $\mathtt{Succ}\,y$ is replaced with the smaller y. The second argument in blue doesn't matter; it can grow from x to $\mathtt{Succ}\,x$ since we already have a termination condition.

Translating this example into the dual language reveals that the same notion of structural recursion covers both induction *and* coinduction [16]. Instead of defining *plus* as matching on just its arguments, we can define it as matching on the structure of its entire call stack α in the command $\langle plus \| \alpha \rangle$. Generalizing to the entire call stack lets us write coinductive definitions using the same technique. For example, here is the definition of *plus* in the dual language alongside *count* which corecursively produces a stream of numbers from a given starting point (*i.e.*, $count\ x = x, x+1, x+2, x+3, \ldots$):

$$\langle \mathit{plus} \| \mathtt{Zero} \cdot x \cdot \alpha \rangle = \langle x \| \alpha \rangle \qquad \langle \mathit{plus} \| \mathtt{Succ}\, y \cdot x \cdot \alpha \rangle = \langle \mathit{plus} \| y \cdot \mathtt{Succ}\, x \cdot \alpha \rangle$$
$$\langle \mathit{count} \| x \cdot \mathtt{Head}\, \alpha \rangle = \langle x \| \alpha \rangle \qquad \langle \mathit{count} \| x \cdot \mathtt{Tail}\, \alpha \rangle = \langle \mathit{count} \| \mathtt{Succ}\, x \cdot \alpha \rangle$$

Both definitions are well-founded because they are structurally recursive, but the difference is the structure they are focused on within the call stack. Whereas the *value* $\mathtt{Succ}\, y$ shrinks to y in the recursive call to *plus*, it's the *covalue* $\mathtt{Tail}\, \alpha$ that shrinks to α in the corecursive call to *count*. In both, the growth in blue doesn't matter, since the red always shrinks.

Here are two more streams defined by structural recursion on the shape of the stream projection $\mathtt{Head}\, \alpha$ or $\mathtt{Tail}\, \alpha$. *iterate* repeats the same function over and over on some starting value (*i.e.*, $\mathit{iterate}\ f\ x = x, f\ x, f(f\ x), f(f(f\ x)), \dots$) and *maps* modifies an infinite stream by applying a function to every element (*i.e.*, $\mathit{maps}\ f\ (x_1, x_2, x_3 \dots) = f\ x_1, f\ x_2, f\ x_3, \dots$):

$$\langle \mathit{iterate} \| f \cdot x \cdot \mathtt{Head}\, \alpha \rangle = \langle f \| x \cdot \alpha \rangle$$
$$\langle \mathit{iterate} \| f \cdot x \cdot \mathtt{Tail}\, \alpha \rangle = \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta \rangle \cdot \alpha \rangle$$
$$\langle \mathit{maps} \| f \cdot xs \cdot \mathtt{Head}\, \alpha \rangle = \langle f \| \mu\beta.\langle xs \| \mathtt{Head}\, \beta \rangle \cdot \alpha \rangle$$
$$\langle \mathit{maps} \| f \cdot xs \cdot \mathtt{Tail}\, \alpha \rangle = \langle \mathit{maps} \| f \cdot \mu\beta.\langle xs \| \mathtt{Tail}\, \beta \rangle \cdot \alpha \rangle$$

4.2.2 (Co)Induction

Examining the structure of (co)values isn't just good for programming; it's good for proving, too. For example, if we want to prove some property Φ about values of type $A \oplus B$, it's enough to show it holds for the (exhaustive) cases of $\iota_1 x_1 : A \oplus B$ and $\iota_1 x_2 : A \oplus B$ like so:

$$\frac{\Phi(\iota_1 x_1) : (\Gamma, x_1 : A \vdash \Delta) \quad \Phi(\iota_2 x_2) : (\Gamma, x_2 : B \vdash \Delta)}{\Phi(x) : (\Gamma, x : A \oplus B \vdash \Delta)}\ \oplus \mathit{Induction}$$

Exhaustiveness is key to ensure that all cases are covered and no possible value was left out. This becomes difficult to do directly for recursive types like `Nat`, because it represents an infinite number of cases $(0, 1, 2, 3, \dots)$. Instead, we can prove a property Φ indirectly through the familiar notion of *structural induction*: prove $\Phi(\mathtt{Zero})$ specifically and prove that the inductive hypothesis $\Phi(y)$ implies $\Phi(\mathtt{Succ}\, y)$ as expressed by this inference rule

$$\frac{\dfrac{}{\Phi(y) : (\Gamma, y : \mathtt{Nat} \vdash \Delta)}\ \mathit{IH} \\ \vdots \\ \Phi(\mathtt{Zero}) : (\Gamma \vdash \Delta) \quad \Phi(\mathtt{Succ}\, y) : (\Gamma, y : \mathtt{Nat} \vdash \Delta)}{\Phi(x) : (\Gamma, x : \mathtt{Nat} \vdash \Delta)}\ \mathtt{Nat}\mathit{Induction}$$

But how can we deal with coinductive codata types? There are also an infinite number of cases to consider, but the values don't follow the same, predictable patterns. Here is a conventional but questionable form of coinduction that takes the entire goal $\Phi(x)$ to be the coinductive hypothesis, as in:

$$\frac{\dfrac{\dfrac{}{\Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta)}\ \mathit{CoIH} \\ \vdots\ ??? \\ \Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta)}{\Phi(x) : (\Gamma, x : \mathtt{Stream}\, A \vdash \Delta)}\ \mathit{Questionable\ CoInduction}$$

But this rule obviously has serious problems: $CoIH$ could just be used immediately, leading to a viciously circular proof. To combat this clear flaw, other secondary, external checks and guards have to be put into place that go beyond the rule itself, and instead analyze the *context* in which $CoIH$ is used to prevent circular proofs. As a result, a prover can build a coinductive proof that follows all the rules, but run into a nasty surprise in the end when the proof is rejected because it fails some implicit guard. Can we do better?

Just like the way structural induction looks at the shape of values, structural coinduction looks at the shape of covalues which represent *contexts* [12]. For example, here is the coinductive rule dual to $\oplus$*Induction* for concluding that a property Φ holds for any output of $A \,\&\, B$ by checking the (exhaustive) cases $\pi_1\alpha_1 : A \,\&\, B$ and $\pi_2\alpha_2 : A \,\&\, B$:

$$\frac{\Phi(\pi_1\alpha_1) : (\Gamma \vdash \alpha_1 : A, \Delta) \quad \Phi(\pi_2\alpha_2) : (\Gamma \vdash \alpha_2 : A, \Delta)}{\Phi(\alpha) : (\Gamma \vdash \alpha : A \,\&\, B, \Delta)}\ \&\mathit{CoInduction}$$

Just like `Nat`, streams have too many cases ($\texttt{Head}\,\beta, \texttt{Tail}[\texttt{Head}\,\beta], \texttt{Tail}[\texttt{Tail}[\texttt{Head}\,\beta]], \dots$) to exhaustively check directly. So instead, here is the dual form of proof as `Nat`*Induction* for proving Φ for any observation α of type $\texttt{Stream}\,A$: it proves the base case $\Phi(\texttt{Head}\,\beta)$ directly, and then shows that the *coinductive hypothesis* $\Phi(\gamma)$ implies the next step $\Phi(\texttt{Tail}\,\gamma)$, like so:

$$\frac{\dfrac{}{\Phi(\gamma) : (\Gamma \vdash \gamma : \texttt{Stream}\,A, \Delta)}\ \mathit{CoIH}}{\vdots}$$
$$\frac{\Phi(\texttt{Head}\,\beta) : (\Gamma \vdash \beta : A, \Delta) \quad \Phi(\texttt{Tail}\,\gamma) : (\Gamma \vdash \gamma : \texttt{Stream}\,A, \Delta)}{\Phi(\alpha) : (\Gamma \vdash \alpha : \texttt{Stream}\,A, \Delta)}\ \texttt{Stream}\mathit{CoInduction}$$

Notice the similarities between this rule and the one for `Nat` induction. In the latter, even though the inductive hypothesis $\Phi(y)$ is assumed for a generic y, then there is no need for external checks because we are forced to provide $\Phi(\texttt{Succ}\,y)$ *for the very same* y. The information flow between the introduction of y in IH and its use in the final conclusion of $\Phi(\texttt{Succ}\,y)$ prevents viciously circular proofs. In the same way, the coinductive rule here assumes $\Phi(\gamma)$ for a generic γ, but we are forced to prove $\Phi(\texttt{Tail}\,\gamma)$ *for the very same* γ. In this case, there is an implicit control flow between the introduction of γ in $CoIH$ and its use in the final conclusion $\Phi(\texttt{Tail}\,\gamma)$. Thus, $CoIH$ can be used in any place it fits, without any secondary guards or checks after the proof is built; `Stream`*CoInduction* is sound as-is.

How can this form of coinduction be used to reason about corecursive programs? Consider this interaction between *maps* and *iterate*: *maps f* (*iterate f x*) = *iterate f* (*f x*). Written in the dual language, this property translates to an equality between commands: $\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \alpha\rangle = \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha\rangle$. We can prove this property (for any starting value x) using coinduction with these two cases:

$\alpha = \texttt{Head}\,\alpha'$. The base case follows by direct calculation with the definitions.

$$\begin{aligned}
\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \texttt{Head}\,\alpha'\rangle &= \langle f \| \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \texttt{Head}\,\beta\rangle \cdot \alpha'\rangle && (\mathit{maps}, \beta_\mu)\\
&= \langle f \| \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha'\rangle && (\mathit{iterate})\\
&= \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \texttt{Head}\,\alpha'\rangle && (\mathit{iterate})
\end{aligned}$$

$\alpha = \texttt{Tail}\,\alpha'$. First, assume the coinductive hypothesis ($CoIH$) which is generic in the value of the initial x: for all x, $\langle \mathit{maps} \| f \cdot \mu\beta.\langle \mathit{iterate} \| f \cdot x \cdot \beta\rangle \cdot \alpha'\rangle = \langle \mathit{iterate} \| f \cdot \mu\beta.\langle f \| x \cdot \beta\rangle \cdot \alpha'\rangle$. The two sides are equated by applying $CoIH$ with an updated value for x:

$$
\begin{aligned}
&\langle maps \| f \cdot \mu\beta.\langle iterate \| f \cdot x \cdot \beta\rangle \cdot \mathtt{Tail}\,\alpha'\rangle \\
&= \langle maps \| f \cdot \mu\beta.\langle iterate \| f \cdot x \cdot \mathtt{Tail}\,\beta\rangle \cdot \alpha'\rangle && (maps, \beta_\mu) \\
&= \langle maps \| f \cdot \mu\beta.\langle iterate \| f \cdot \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \beta\rangle \cdot \alpha'\rangle && (iterate) \\
&= \langle iterate \| f \cdot \mu\beta.\langle f \| \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \beta\rangle \cdot \alpha'\rangle && (CoIH\{\mu\gamma.\langle f \| x \cdot \gamma\rangle / x\}) \\
&= \langle iterate \| f \cdot \mu\gamma.\langle f \| x \cdot \gamma\rangle \cdot \mathtt{Tail}\,\alpha'\rangle && (iterate)
\end{aligned}
$$

4.3 Compilation and Intermediate Languages

In Section 3, we saw how a symmetric language based on the sequent calculus closely resembles the structure of an abstract machine, which helps to reveal the details of how programs are really implemented. This resemblance raises the question: does a language based on the sequent calculus be a good *intermediate language* (IL) used to compile programs to machine code? The λ-calculus' syntax structure buries the most relevant part of an expression. For example, applying f to four arguments is written as ((((f 1) 2) 3) 4); we are forced to search for the next step – f 1 – found at the bottom of the tree. Instead, the syntax of the dual calculus raises up the next step of a program to the top; the same application is written as $\langle f \| 1 \cdot (2 \cdot (3 \cdot (4 \cdot \alpha)))\rangle$, where calling f with 1 is the first part of the command.

We have found that the sequent calculus can in fact be used as an intermediate language of a compiler [17]. The feature of bringing out the most relevant expression to the top of a program is shared by other commonly-used representations like *continuation-passing style* (CPS) [2] and *static single assignment* (SSA) [5]. However, the sequent calculus is uniquely flexible. Unlike SSA which is an inherently imperative representation, the sequent calculus is a good fit for both purely functional and effectful languages. And unlike CPS, the sequent calculus preserves enough of the original structure of the program to enable high-level rewrite rules expressed in terms of the source, as done by the Glasgow Haskell Compiler (GHC). Besides these advantages, our experience with a sequent calculus IL has led the following new techniques, which apply more broadly to other compiler ILs, too.

4.3.1 Join points in control flow

Join points are places where separate lines of control flow come back together. They are as pervasive as the branching structures in a program. For example, the statement

```
if x > 100: print "x is large"
else:       print "x is small"
print "goodbye"
```

splits off in two different directions to print a different statement depending on the value of `x`. But in either case, both branches of control flow will rejoin at the shared third line to print `"goodbye"`. Compilers need to represent these join points for code generation and optimization, in a way that is efficient in both time and space. Ideally, we want to generate code to jump to the join point in as few instructions as possible. And it's not acceptable to copy the common code into each branch; this leads to a space inefficiency that can cause an exponential blowup in the size of the generated code.

In the past, GHC represented these join points as ordinary functions bound by a let-expression. For example, the function j in $\mathbf{let}\, j\ x = \ldots x \ldots \mathbf{in}\ \mathbf{if}\, z\, \mathbf{then}\, j\ 10\, \mathbf{else}\, j\ 20$ serves as the join point for both branches of the if-expression. Of course, this is space efficient,

since it avoids duplicating code of j. But a full-fledged function call is much less efficient than a simple jump. Fortunately, the local function j has some special properties: it is always used in tail-call position and never escapes the scope of the **let**. These properties let GHC compile the calls $j\ 10$ and $j\ 20$ as efficient jumps. Unfortunately, the necessary properties for optimization aren't stable under *other* useful optimizations. For example, it usually helps to push (strict) evaluation contexts inside of an if-then-else or case-expression. While semantically correct, this can break the tail-call property of join points like here:

$$\begin{array}{l} 3 + \mathbf{let}\, j\ y = 10 + (y + y) \\ \quad \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad\quad \iota_1 z_1 \to j\ z_1 \\ \quad\quad \iota_2 z_2 \to j\ (-z_2) \end{array} \quad \to \quad \begin{array}{l} \mathbf{let}\, j\ y = 10 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to 3 + (j\ z_1) \\ \quad \iota_2 z_2 \to 3 + (j\ (-z_2)) \end{array}$$

Before, j could be compiled as a join point, but after it is used in non-tail-call positions $3 + (j\ z_1)$ and $3 + (j\ (-z_2))$. To combat this issue, we developed a λ-calculus with purely-functional join points [29]. While this calculus ostensibly contains labels and jumps – which are indeed compiled to jumps into assembly code – from the outside there is no observable effect. Instead, this calculus gives rules for optimizing around join points while ensuring they are still compiled efficiently. The example above is rewritten like so, where the context $3 + \Box$ is now pushed into the code of the join point, rather than inside of the case-expression:

$$\begin{array}{l} 3 + \mathbf{join}\, j\ y = 10 + (y + y) \\ \quad \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad\quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad\quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array} \quad \to \quad \begin{array}{l} \mathbf{join}\, j\ y = 3 + 10 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array} \quad \to \quad \begin{array}{l} \mathbf{join}\, j\ y = 13 + (y + y) \\ \mathbf{in\ case}\, x\, \mathbf{of} \\ \quad \iota_1 z_1 \to \mathbf{jump}\, j\ z_1 \\ \quad \iota_2 z_2 \to \mathbf{jump}\, j\ (-z_2) \end{array}$$

Besides preserving the efficiency of j itself, this new form of code movement enables new optimizations. In this case, we can perform some additional constant folding of $3 + 10$, and other optimizations such as loop fusion can be expressed in this way as well.

4.3.2 Polarized primitive types

Another key feature found in the duality of logic is the *polarization* of different propositions. In terms of computation [33, 30], polarization is the combination of an "ideal" evaluation strategy based on the structure of types. Consider the η laws expressing extensionality of the various types in Figure 3. All the η laws for data types (*e.g.,* built with $\otimes$, $\oplus$, $\ominus$, and $\exists$) are about expanding covalues α. These laws are the strongest in the call-by-value strategy, which maximizes the number of covalues. Dually, the η laws for codata types (*e.g.,* built with $\&$, $\parr$, $\neg$, and $\forall$) are about expanding values x. These are the strongest in call-by-name.

Usually, we think of picking *one* evaluation strategy for a language. But this means that in either case, we are necessarily weakening extensionality of data or codata types (or both, if we choose something other than call-by-value or call-by-name). Instead, we can use a *polarized* language which improves η laws for *all* types by combining both strategies. This involves separating types into two different camps – the positive Type_+ and the negative Type_- – following our analogy of the burden of proof from Section 2.2 like so:

$$\begin{array}{rl} \mathit{Sign} \ni & s ::= + \mid - \\ \mathit{Type}_+ \ni & A_+, B_+ ::= X_+ \mid A_+ \oplus B_+ \mid A_+ \otimes B_+ \mid \exists X_s.A_+ \mid \ominus A_- \mid \downarrow A_- \\ \mathit{Type}_- \ni & A_-, B_- ::= X_- \mid A_- \,\&\, B_- \mid A_- \parr B_- \mid \forall X_s.A_- \mid \neg A_+ \mid \uparrow A_+ \end{array}$$

By separating types in two, we also have to add the *polarity shifts* $\downarrow A_-$ and $\uparrow A_+$ so they can still refer to one another. For example, the plain $A \oplus (B \,\&\, C)$ becomes $A_+ \oplus \downarrow(B_- \,\&\, C_-)$.

Once this separation of types has occurred, we can bring them back together and intermingle both within a single language. The distinction can be made explicit in a refined *Cut* rule, which is the only rule which creates computation, so that the type (and its sign) becomes part of the program:

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

Since there is no longer one global evaluation strategy, we instead use *types* to determine the order. The additional annotation in commands let us drive computation with more nuance, referring to the sign s of the command to determine the priorities of μ and $\tilde{\mu}$ computations:

$$(\beta^s_\mu) \qquad \langle \mu\alpha.c | A{:}s | E_s \rangle = c\{E_s/\alpha\} \qquad (\beta^s_{\tilde{\mu}}) \qquad \langle V_s | A{:}s | \tilde{\mu}x.c \rangle = c\{V_s/x\}$$

The advantage of this more nuanced form of computation is that the types of the language express the nice properties that usually only hold up in an idealized, pure theory; however, now they hold up in the pragmatic practice that combines all manner of computational effects like control flow, state, and general recursion. For example, we might think that curried and uncurried functions – $A \to (B \to C)$ versus $(A \otimes B) \to C$ – are exactly the same. In both Haskell and OCaml, they are not, due to interactions with non-termination or side effects. But in a polarized language, they are the same, even with side effects.

These ideal properties of polarized types let us encode a vast array of user-defined data and codata types into a small number of basic primitives. We can choose a perfectly symmetric basis of connectives found in Section 3 [11] or an asymmetric alternative that is suited for purely functional programs [9]. The ideal properties provided by polarization can be understood in terms of the dualities of evidence in Section 2.3. For example, the equivalence between the propositions $\ominus\neg A$ and A corresponds to an *isomorphism* between the polarized types $\ominus\neg A_+$ and A_+ (and dually $\neg \ominus A_-$ and A_-). Intuitively, the only (closed) values of type $\ominus\neg A$ have exactly the form $([V_v])$, which is in bijection with the plain values V_v. And coterms of those two types are also in bijection due to the optimized η laws. All the de Morgan equivalences in Section 2.3 correspond to type isomorphisms, too. For example, the only (closed) values of $\ominus\forall X_s.B_-$ have the form $([A_s, E_-])$, which is in bijection with $\exists X_s. \ominus B_-$'s (closed) values of the form $(A_s, (E_-))$. In contrast, the other negation $\neg\downarrow\forall X_s.B_-$ includes abstract values of the form $\mu[x].c$, which are *not* isomorphic to the more concrete values $(A_s, \mu[x].c)$ of $\exists X_s.\neg\downarrow B_-$ that witness their chosen A_s. Thus, constructivity, computation, and full de Morgan symmetry depend on both polarized negations.

Polarization itself only accounts for call-by-value and call-by-name evaluation. However, other evaluation strategies are sometimes used in practice for pragmatic reasons. For example, implementations of Haskell use call-by-need evaluation, which can lead to better asymptotic performance than call-by-name. How do other evaluation strategies fit? We can add additional signs – besides $-$ and $+$ – that stand in for other strategies like call-by-need. But do we need to duplicate the basic primitives? No! We only need additional shifts that convert between the new sign(s) with the original $+$ and $-$, four in total:

$$\begin{array}{ll}
\textbf{data}\downarrow^s(X : s) : + \textbf{ where} & \textbf{data}\Uparrow^s(X : +) : s \textbf{ where} \\
\quad \texttt{Box}_s : \ X : s \vdash \downarrow^s X : + \ \mid & \quad \texttt{Return}_s : \ X : + \vdash \Uparrow^s X : s \ \mid \\
\textbf{codata}\uparrow_s(X : s) : - \textbf{ where} & \textbf{codata}\Downarrow_s(X : -) : s \textbf{ where} \\
\quad \texttt{Eval}_s : \ \mid \uparrow_s X : - \vdash X : s & \quad \texttt{Enter}_s : \ \mid \Downarrow_s X : s \vdash X : -
\end{array}$$

4.3.3 Static calling conventions

Systems languages like C give the programmer fine-grained control over low-level representations and calling conventions. When defining a structure, the programmer can choose if values are stored directly or indirectly (*i.e., boxed*) as a pointer into the heap. When calling a function, the programmer can choose how many arguments are passed at once, and if they are passed directly in the call stack, or indirectly by reference. High-level functional languages save programmers from these details, but at the cost of using less efficient – but more uniform – representations and calling conventions. Is there a way to reconcile both high-level ease and low-level control?

It turns out that polarization also provides a logical foundation for efficient representations and calling conventions, too. Decades ago [32], Haskell implementors designed a way to add unboxed representations into the compiler IL, making it possible to more efficiently pass values directly in registers. However, doing so required extending the language, because unboxed values *must* be call-by-value, and the types of unboxed values are different from the other, ordinary Haskell types. This sounds awfully similar to polarization: unboxed values correspond to positive data types, which have a different polarity from Haskell's types.

With this inspiration, we considered the dual problem: what do negative types correspond to? If an unboxed pair (V_+, W_+) is described by the positive type $A_+ \otimes B_+$, then does an *unboxed call stack* $V_+ \cdot E_-$ correspond to the *negative function type* $A_+ \to B_-$? In [19], we found that negative functions correspond to a more primitive type of functions found in the machine, where the power of the polarized η law lets us express the arity of functions statically in the type. Static arity is important for optimizing higher-order functions. In

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith f _      _      = []
```

we cannot compile `f a b` as a simple binary function call even though `f`'s type suggests so. It might be that `f` only expects one argument, then computes a closure expecting the next. Instead, using negative functions, which are fully extensional, lets us statically optimize `zipWith` to pass both arguments to `f` at once.

However, this approach runs into some snags in practice, due to *polymorphism*. In order to be able to statically compile code, we sometimes need to know the representation of a type (to move its values around) or the calling convention of a type (to jump to its code in the correct environment). But if a type is unknown – because it's some polymorphic type variable – then that runtime information is unknown at compile time. A solution to this problem is given in [21], which introduces the idea of storing the runtime representation of values in the *kind* of their type. So even when a type is not known statically, their kind is. Following this idea, we combined the kind-based approach with function arity by storing both representations and *calling conventions* in kinds [14].

This can be seen as a refinement on the course-grained polarization from Section 4.3.2. Rather than just a basic sign – such as $-$ or $+$ – types are described by a pair of both a representation and a calling convention. Positive types like $A \otimes B$ can have interesting representations (their values can be tuples, tagged unions, or machine primitives) but have a plain convention (their terms are always just evaluated to get the resulting value). In contrast, negative types like $A \to B$ can have interesting conventions (they can be called with several arguments, which can have their own representations by value or reference) but have a plain representation (they are just stored as pointers). This approach lets us integrate efficient calling conventions in a higher-level language with polymorphism, and also lets us be polymorphic in representations and calling conventions *themselves*, introducing new forms of statically-compilable code re-use.

4.4 Orthogonal Models of Safety

We've looked at several applications based on the dual calculus in Section 3, but how do we know the calculus is *safe*? That is, what sorts of safety properties do the typing rules provide? For example, in certain applications, we might want to know for sure that well-typed programs, like the ones in Section 4.2, always terminate. We also might want a guarantee that the $\beta\eta$ equational theory in Section 3.5 is actually consistent. To reason about the impact of types, we must identify the safety property we're interested in. This is done with a chosen set of commands $\perp\!\!\!\perp$ called the *pole* which contains only those commands we deem as "safe." Despite being tailor-made to classify different notions of safety, there are shockingly few requirements of $\perp\!\!\!\perp$. In fact, the only requirement is that the pole must be *closed under expansion*: $c \mapsto c' \in \perp\!\!\!\perp$ implies $c \in \perp\!\!\!\perp$. *Any* set of commands closed under expansion can be used for $\perp\!\!\!\perp$. This gives the general framework for modeling type safety a large amount of flexibility to capture different properties, types, and language features. So in the following, assume only that $\perp\!\!\!\perp$ is an arbitrary set closed under expansion, and the sign s can stand for either $+$ (call-by-value) or $-$ (call-by-name) throughout.

4.4.1 Orthogonality and intuitionistic negation

The central concept in these family of models is *orthogonality* given in terms of the chosen pole $\perp\!\!\!\perp$. At an individual level, a term and coterm are orthogonal to one another, written $v \perp\!\!\!\perp e$, if they form a command in the pole: $\langle v \| e \rangle \in \perp\!\!\!\perp$. Generalizing to groups, a set of terms $\mathbb{A}^+$ and a set of coterms $\mathbb{A}^-$ are orthogonal, written $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$, if every combination drawn from the two sets is orthogonal: $v \perp\!\!\!\perp e$ for all $v \in \mathbb{A}^+$ and $e \in \mathbb{A}^-$. Working with sets has the benefit that we can always find the *biggest* set orthogonal to another. That is, for any set of terms $\mathbb{A}^+$, there is a largest set of coterms called $\mathbb{A}^{+\perp\!\!\!\perp}$ such that $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^{+\perp\!\!\!\perp}$ (and vice versa for any coterm set $\mathbb{A}^-$, there is a largest $\mathbb{A}^{-\perp\!\!\!\perp} \perp\!\!\!\perp \mathbb{A}^-$), defined as:

$$e \in \mathbb{A}^{+\perp\!\!\!\perp} \iff \forall v \in \mathbb{A}^+ . \langle v \| e \rangle \in \perp\!\!\!\perp \qquad\qquad v \in \mathbb{A}^{-\perp\!\!\!\perp} \iff \forall e \in \mathbb{A}^- . \langle v \| e \rangle \in \perp\!\!\!\perp$$

The fascinating thing about this notion of orthogonality is that – despite the fact that it was designed for symmetric and classical systems – it so closely mimics the properties of negation from the asymmetric *intuitionistic* logic. For example, it enjoys the properties analogous to *double negation introduction* ($A \implies \neg\neg A$) and *triple negation elimination* ($\neg\neg\neg A \iff A$) where $\mathbb{A}^{\pm\perp\!\!\!\perp}$ corresponds to the negation of $\mathbb{A}^\pm$ (which could be either a set of terms or a set of coterms) and set inclusion $\mathbb{A}^\pm \subseteq \mathbb{B}^\pm$ corresponds to implication.

▶ **Lemma 3** (Orthogonal Introduction/Elimination). $\mathbb{A}^\pm \subseteq \mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp}$ *and* $\mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp\perp\!\!\!\perp} = \mathbb{A}^{\pm\perp\!\!\!\perp}$.

However, the classical principle of *double negation elimination* ($\neg\neg A \implies A$) does *not* hold for orthogonality: in general, $\mathbb{A}^{\pm\perp\!\!\!\perp\perp\!\!\!\perp} \not\subseteq \mathbb{A}^\pm$. This connection is not just a single coincidence. Orthogonality also has properties corresponding to the contrapositive ($A \implies B$ implies $\neg B \implies \neg A$) as well as all the intuitionistic directions of the de Morgan laws from Section 2.3 – where set union ($\mathbb{A}^\pm \cup \mathbb{B}^\pm$) denotes disjunction and intersection ($\mathbb{A}^\pm \cap \mathbb{B}^\pm$) denotes conjunction – but, again, *not* the classical-only directions like $\neg(A \wedge B) \implies (\neg A) \vee (\neg B)$.

Where does $\perp\!\!\!\perp$'s closure under expansion come into play? It lets us reason about sets of the form $\mathbb{A}^{\pm\perp\!\!\!\perp}$, and argue that they must contain certain elements by virtue of the way they *behave* with elements of the underlying $\mathbb{A}^\pm$, rather than the way they were built. For example, we can show that general μs and $\tilde{\mu}$s belong to orthogonally-defined sets, as long as their commands are safe under any possible substitution.

▶ **Observation 4.** *For any set of* values $\mathbb{A}^+$*, if* $c\{V_s/x\} \in \perp\!\!\!\perp$ *for all* $V_s \in \mathbb{A}^+$ *then* $\tilde{\mu}x.c \in \mathbb{A}^{+\perp\!\!\!\perp}$*. For any set of* covalues $\mathbb{A}^-$*, if* $c\{E_s/\alpha\} \in \perp\!\!\!\perp$ *for all* $E_s \in \mathbb{A}^-$ *then* $\mu\alpha.c \in \mathbb{A}^{-\perp\!\!\!\perp}$*.*

Proof. For all values, $V_s \in \mathbb{A}^+$, observe that $\langle V_s \| \tilde{\mu}x.c\rangle \mapsto_{\beta^s_{\tilde{\mu}}} c\{V_s/x\} \in \perp\!\!\!\perp$. Thus, $\langle V_s \| \tilde{\mu}x.c\rangle \in \perp\!\!\!\perp$ by closure under expansion, so $\tilde{\mu}x.c \in \mathbb{A}^{+\perp\!\!\!\perp}$ by definition. The other case is dual. ◁

Note the fact that Observation 4 starts with only a set of *values* or *covalues*, rather than general (co)terms. This (co)value restriction is necessary to ensure that the $\beta^s_{\tilde{\mu}}$ and β^s_{μ} rules can fire, which triggers the closure-under-expansion result. Formally, we write this restriction as $\mathbb{A}^{\pm\mathbb{V}}$ to denote the subset of $\mathbb{A}^{\pm}$ containing only (co)values, which is built into the very notion of candidates that model safety of individual types.

▶ **Definition 5** (Candidates). *A* reducibility candidate*,* $\mathbb{A} \in \mathcal{RC}$*, is a pair* $\mathbb{A} = (\mathbb{A}^+, \mathbb{A}^-)$ *of a set of terms* $(\mathbb{A}^+)$ *and set of coterms* $(\mathbb{A}^-)$ *that is:*
Sound *For all* $v \in \mathbb{A}^+$ *and* $e \in \mathbb{A}^-$*,* $\langle v \| e\rangle \in \perp\!\!\!\perp$ *(*i.e., $\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^-$*).*
Complete *If* $\langle v \| E_s\rangle \in \perp\!\!\!\perp$ *for all* covalues $E_s \in \mathbb{A}^-$ *then* $v \in \mathbb{A}^+$ *(*i.e., $\mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp} \subseteq \mathbb{A}^+$*).*
If $\langle V_s \| e\rangle \in \perp\!\!\!\perp$ *for all* values $V_s \in \mathbb{A}^+$*, then* $e \in \mathbb{A}^-$ *(*i.e., $\mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp} \subseteq \mathbb{A}^-$*).*
We write $v \in \mathbb{A}$ *as shorthand for* $v \in \mathbb{A}^+$ *and* $e \in \mathbb{A}$ *for* $e \in \mathbb{A}^-$*.*

There are two distinct ways of defining specific reducibility candidates. We could begin with a set $\mathbb{A}^+$ of terms, and build the rest of the candidate around the values of $\mathbb{A}^+$, or we can start with a set $\mathbb{A}^-$ of coterms, and build the rest around the covalues of $\mathbb{A}^-$. These are the *positive* ($\mathrm{Pos}(\mathbb{A}^+)$) and *negative* ($\mathrm{Neg}(\mathbb{A}^-)$) construction of candidates, defined as:

$$\mathrm{Pos}(\mathbb{A}^+) = (\mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}, \mathbb{A}^{+\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}) \qquad \mathrm{Neg}(\mathbb{A}^-) = (\mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp}, \mathbb{A}^{-\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp})$$

Importantly, these constructions are indeed reducibility candidates, meaning they are both sound and complete. But why are three applications of orthogonality needed instead of just two (like some other models in this family)? The extra orthogonality is needed because of the *(co)value restriction* $\mathbb{A}^{\pm\mathbb{V}}$ interleaved with orthogonality $\mathbb{A}^{\pm\perp\!\!\!\perp}$. Taken together, (co)value-restricted orthogonality has similar introduction and elimination properties as the general one (Lemma 3), but restricted to just (co)values rather than general (co)terms.

▶ **Lemma 6.** $\mathbb{A}^{\pm\mathbb{V}} \subseteq \mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}}$ *and* $\mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}\perp\!\!\!\perp\mathbb{V}} = \mathbb{A}^{\pm\mathbb{V}\perp\!\!\!\perp\mathbb{V}}$*.*

Thus, the final application of orthogonality takes these (co)values and soundly completes the rest of the candidate.[8]

4.4.2 An orthogonal view of types

With the positive and negative construction of candidates, we can define operations that are analogous to the positive and negative burden of proof from Section 2.2. Here, terms represent evidence of truth, and coterms represent evidence of falsehood, so the various connectives are built like so:

$$\begin{aligned}
\mathbb{A} \otimes \mathbb{B} &= \mathrm{Pos}\{(v,w) \mid v \in \mathbb{A}, w \in \mathbb{B}\} & \mathbb{A} \parr \mathbb{B} &= \mathrm{Neg}\{[e,v] \mid e \in \mathbb{A}, f \in \mathbb{B}\} \\
\mathbb{A} \oplus \mathbb{B} &= \mathrm{Pos}(\{\iota_1 v \mid v \in \mathbb{A}\} \cup \{\iota_2 w \mid w \in \mathbb{B}\}) & \mathbb{A} \,\&\, \mathbb{B} &= \mathrm{Neg}(\{\pi_1 e \mid e \in \mathbb{A}\} \cup \{\pi_2 f \mid f \in \mathbb{B}\}) \\
\ominus\mathbb{A} &= \mathrm{Pos}\{(e) \mid e \in \mathbb{A}\} & \neg\mathbb{A} &= \mathrm{Neg}\{[v] \mid v \in \mathbb{A}\}
\end{aligned}$$

[8] In fact, the simpler double-orthogonal constructions are valid, but only in certain evaluation strategies. In call-by-value, where $\mathbb{A}^{-\mathbb{V}} = \mathbb{A}^-$ because every coterm is a covalue, the positive construction simplifies to just the usual $\mathrm{Pos}(\mathbb{A}^+) = (\mathbb{A}^{+\perp\!\!\!\perp\perp\!\!\!\perp}, \mathbb{A}^{+\perp\!\!\!\perp})$ when $\mathbb{A}^+$ contains only values. Dually in call-by-name, the negative construction simplifies to just $\mathrm{Neg}(\mathbb{A}^-) = (\mathbb{A}^{-\perp\!\!\!\perp}, \mathbb{A}^{-\perp\!\!\!\perp\perp\!\!\!\perp})$ when $\mathbb{A}^-$ contains only covalues.

Similarly, evidence for or against the existential and universal quantifiers can be defined as operations taking a function $\mathbb{F} : \mathcal{RC} \to \mathcal{RC}$ over reducibility candidates, and producing a specific reducibility candidate that quantifies over *all* possible instances of $\mathbb{F}(\mathbb{B})$.[9]

$$\exists\mathbb{F} = \mathrm{Pos}\{(A, v) \mid \mathbb{B} \in \mathcal{RC}, v \in \mathbb{F}(\mathbb{B})\} \qquad \forall\mathbb{F} = \mathrm{Neg}\{[A, e] \mid \mathbb{B} \in \mathcal{RC}, e \in \mathbb{F}(\mathbb{B})\}$$

With a semantic version of the connectives, we have a direct way to translate each syntactic type to a reducibility candidate. The translation $[\![A]\!]\theta$ is aided by a map θ from type variables to reducibility candidates, and the cases of translation are now by the numbers:

$$[\![X]\!]\theta = \theta(X) \quad [\![A \otimes B]\!]\theta = [\![A]\!]\theta \otimes [\![B]\!]\theta \quad \ldots \quad [\![\forall X.B]\!]\theta = \forall(\lambda\mathbb{A}{:}\mathcal{RC}.[\![B]\!]\theta\{\mathbb{A}/X\})$$

Going further, we can translate typing judgments to logical statements.

$$\begin{aligned}
[\![c : (\Gamma \vdash \Delta)]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ c\{\sigma\} \in \perp\!\!\!\perp \\
[\![\Gamma \vdash v : A \mid \Delta]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ v\{\sigma\} \in [\![A]\!]\theta \\
[\![\Gamma \mid e : A \vdash \Delta]\!]\theta &= \forall\sigma \in [\![\Gamma \vdash \Delta]\!]\theta.\ e\{\sigma\} \in [\![A]\!]\theta
\end{aligned}$$

Each judgment is based on a translation of the environment, $\sigma \in [\![\Gamma \vdash \Delta]\!]\theta$, which says that σ is a syntactic substitution of (co)values for (co)variables such that $x\{\sigma\} \in [\![A]\!]\theta$ if $x : A$ is in Γ, and similarly for $\alpha : A$ in Δ. The main result is that typing derivations imply the truth of their concluding judgment, which follows by induction on the derivation.

▶ **Theorem 7** (Adequacy). *$c : (\Gamma \vdash \Delta)$ implies $[\![c : (\Gamma \vdash \Delta)]\!]\theta$ (and similar for (co)terms).*

4.4.3 Applications of adequacy

Adequacy (Theorem 7) may not seem like a special property, but the generality of the model means that it has many serious implications. We get different results by plugging in a different notion of safety for $\perp\!\!\!\perp$. The most basic corollary of adequacy is given by the most trivial pole: $\perp\!\!\!\perp = \{\}$ is vacuously closed under expansion since it is empty to start with. By instantiating adequacy with $\perp\!\!\!\perp = \{\}$, we get a notion of logical consistency, there is no derivation of a closed contradiction $c : (\bullet \vdash \bullet)$ since $[\![c : (\bullet \vdash \bullet)]\!]$ means that $c \in \{\}$.

▶ **Corollary 8** (Logical Consistency). *There is no well-typed $c : (\bullet \vdash \bullet)$.*

However, the most interesting results come from instances where $\perp\!\!\!\perp$ is not empty. For example, the set of terminating commands, $\{c \mid c \mapsto\!\!\!\!\twoheadrightarrow c' \not\mapsto\}$, is also closed under expansion. Defining $\perp\!\!\!\perp$ as this set ensures that all well-typed commands are terminating.

▶ **Corollary 9** (Termination). *If $c : (\Gamma \vdash \Delta)$ then $c \mapsto\!\!\!\!\twoheadrightarrow_{\beta\varsigma} c' \not\mapsto$.*

But perhaps the most relevant application to discuss here is how *constructivity* from Section 2 is reconciled with computation in Section 3. The notion of positive constructive evidence of $A \oplus B$ (Section 2.2) corresponds directly with the two value constructors: we have $\iota_1 V_1 : A_1 \oplus A_2$ and $\iota_2 V_2 : A_1 \oplus A_2$ for any value $V_i : A_i$. Similarly, the evidence in favor of $\exists X.B$ corresponds directly with the constructed value $(A, V) : \exists X.B$ where $V : B[A/X]$.

[9] Note that there is no connection between the syntactic type A used in (A, v) and $[A, e]$ and the actual reducibility candidate used in $\mathbb{F}(\mathbb{B})$ that classifies v and e. Just like System F's model of impredicativity [22], we can get away with this bald-faced lie because of *parametricity* of $\forall$ and $\exists$: the (co)term that unpacks (A, v) or $[A, e]$ is not allowed to react any differently based on the choice for A.

But both of these types also have the general μ abstractions $\mu\alpha.c : A \oplus B$ and $\mu\beta.c' : \exists X.B$, which do not directly correspond with either. How do we know that both of these μs will compute and eventually produce the required evidence? We can instantiate $\perp\!\!\!\perp$ with only the commands that do so. For $A \oplus B$ we can set $\perp\!\!\!\perp = \{c \mid c \mapsto\!\!\!\!\twoheadrightarrow \langle \iota_i V \| \alpha\rangle\}$, and for $\exists X.B$ we can set $\perp\!\!\!\perp = \{c \mid c \mapsto\!\!\!\!\twoheadrightarrow \langle (A, V) \| \alpha\rangle\}$; both of these definitions are closed under expansion, which is all we need to apply adequacy to compute the construction matching the type.

▶ **Corollary 10** (Positive Evidence). *If* $\bullet \vdash v : A_1 \oplus A_2 \mid$ *then* $\langle v \| \alpha\rangle \mapsto\!\!\!\!\twoheadrightarrow_{\beta^s \varsigma^s} \langle \iota_i V_s \| \alpha\rangle$ *such that* $V_s \in [\![A_i]\!]$. *If* $\bullet \vdash v : \exists X.B \mid$ *then* $\langle v \| \alpha\rangle \mapsto\!\!\!\!\twoheadrightarrow_{\beta^s \varsigma^s} \langle (A, V_s) \| \alpha\rangle$ *such that* $V_s \in [\![B]\!]\{[\![A]\!]/X\}$.

Dually, we can design similar poles which characterize the computation of negative evidence. For example, types like $A_1 \,\&\, A_2$ and $\forall X.B$ include general $\tilde{\mu}$ abstractions of the form $\tilde{\mu}x.c$ in addition to the constructed covalues $\pi_1 E_1 : A_1$, $\pi_2 E_2 : A_2$, and $[A, E] : \forall X.B$ that correspond to the negative evidence of these connectives. Luckily, we can set the global $\perp\!\!\!\perp$ to either $\{c \mid c \mapsto\!\!\!\!\twoheadrightarrow \langle x \| \pi_i E\rangle\}$ or $\{c \mid c \mapsto\!\!\!\!\twoheadrightarrow \langle x \| [A, E]\rangle\}$ to ensure that general $\tilde{\mu}$s compute the correct concrete evidence for these negative types.

▶ **Corollary 11** (Negative Evidence). *If* $\mid e : A_1 \,\&\, A_2 \vdash \bullet$ *then* $\langle x \| e\rangle \mapsto\!\!\!\!\twoheadrightarrow_{\beta^s \varsigma^s} \langle x \| \pi_i E_s\rangle$ *such that* $E_s \in [\![A_i]\!]$. *If* $\mid e : \forall X.B \vdash \bullet$ *then* $\langle x \| e\rangle \mapsto\!\!\!\!\twoheadrightarrow_{\beta^s \varsigma^s} \langle x \| [A, E_s]\rangle$ *such that* $E_s \in [\![B]\!]\{[\![A]\!]/X\}$.

This model is extensible with other language features, too, without fundamentally changing the shape of adequacy (Theorem 7). For example, because reducibility candidates are two-sided objects, there are two different ways to order them:

$$\mathbb{A} \sqsubseteq \mathbb{B} \iff \mathbb{A}^+ \subseteq \mathbb{B}^+ \text{ and } \mathbb{A}^- \subseteq \mathbb{B}^- \qquad \mathbb{A} \leq \mathbb{B} \iff \mathbb{A}^+ \subseteq \mathbb{B}^+ \text{ and } \mathbb{A}^- \supseteq \mathbb{B}^-$$

The first order $\mathbb{A} \sqsubseteq \mathbb{B}$ where both sides are in the same direction is analogous to ordinary set containment. However, the second order $\mathbb{A} \leq \mathbb{B}$ where the two sides are opposite instead denotes *subtyping* [15]. Besides modeling subtyping as a language feature itself, this idea is the backbone of several other type features, including (co)inductive types [12], intersection and union types [13], and indexed (co)data types [16]. It also lets us model non-determinism [15], where the critical pair between μ and $\tilde{\mu}$ is allowed.

We can also generalize the form of our model, to capture properties that are binary relations rather than unary predicates. This only requires that we make each of the fundamental components binary, without changing their overall structure. For example, the pole $\perp\!\!\!\perp$ is generalized from a *set* to a *relation* between commands that is closed under expansion: $c_1 \mapsto\!\!\!\!\twoheadrightarrow c_1' \perp\!\!\!\perp c_2' \twoheadleftarrow\!\!\!\!\mapsto c_2$ implies $c_1 \perp\!\!\!\perp c_2$. From there, reducibility candidates become a pair of term relation $v \; \mathbb{A}^+ \; v$ and coterm relation $e \; \mathbb{A}^- \; e'$, where soundness and completeness can be derived from the generalized notion of orthogonality between relations:

$$\mathbb{A}^+ \perp\!\!\!\perp \mathbb{A}^- \iff \forall (v \; \mathbb{A}^+ \; v'),\ (e \; \mathbb{A}^- \; e').\ \langle v \| e\rangle \perp\!\!\!\perp \langle v' \| e'\rangle$$

This lets us represent equalities between commands and (co)terms in the orthogonality model, and prove that the equational theory is consistent with contextual equivalence [6], *i.e.,* equal expressions produce the same result in any context. As a consequence, (co)values built with distinct constructors – such as ι_1 and ι_2 or π_1 and π_2 – are never equal.

▶ **Corollary 12** (Equational Consistency). *The equalities* $\Gamma \vdash \iota_1 V_s = \iota_2 V_s' : A \oplus B \mid \Delta$ *and* $\Gamma \mid \pi_1 E_s = \pi_2 E_s' : A \,\&\, B \vdash \Delta$ *are not derivable.*

5 Conclusion

Duality is not just an important idea in logic; it is also a useful tool to study and implement programs. By re-imagining constructive logic as a fair debate between multiple competing viewpoints, we derive a symmetric calculus that lets us transfer the logical idea of duality to computation. This modest idea has serious ramifications, and leads to several applications in both the theory and practice of programming languages. Moreover, it reveals new ideas and new relationships that are not expressible in today's languages. We hope the next generation of programming languages puts the full force of duality into programmers' hands.

References

1 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, 2013.

2 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

3 L. E. J. Brouwer. *Over de Grondslagen der Wiskunde*. PhD thesis, University of Amsterdam, 1907.

4 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 233–243. ACM, 2000.

5 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

6 Paul Downen. *Sequent Calculus: A Logic and a Language for Computation and Duality*. PhD thesis, University of Oregon, 2017.

7 Paul Downen and Zena M. Ariola. Compositional semantics for composable continuations: From abortive to delimited control. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 109–122. ACM, 2014.

8 Paul Downen and Zena M. Ariola. The duality of construction. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer Berlin Heidelberg, 2014.

9 Paul Downen and Zena M. Ariola. Beyond polarity: Towards a multi-discipline intermediate language with sharing. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, LIPIcs, pages 21:1–21:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

10 Paul Downen and Zena M. Ariola. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming*, 28:e3, 2018.

11 Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Logical Methods in Computer Science*, 16:13:1–13:57, 2020. `arXiv:1907.13227`.

12 Paul Downen and Zena M. Ariola. A computational understanding of classical (co)recursion. In *22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20. Association for Computing Machinery, 2020.

13 Paul Downen, Zena M. Ariola, and Silvia Ghilezan. The duality of classical intersection and union types. *Fundamenta Informaticae*, 170:1–54, 2019.

14 Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. Kinds are calling conventions. *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020.

15 Paul Downen, Philip Johnson-Freyd, and Zena Ariola. Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming*, 111, 2019.

16 Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. Structures for structural recursion. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 127–139. ACM, 2015.

17 Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 74–88. ACM, 2016.

18 Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. Codata in action. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 119–146. Springer International Publishing, 2019.

19 Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. Making a faster curry with extensional types. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, pages 58–70. ACM, 2019.

20 M. Dummett and R. Minio. *Elements of Intuitionism*. Oxford University Press, 1977.

21 Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 525–539. Association for Computing Machinery, 2017.

22 Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

23 Timothy G. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58. ACM, 1990.

24 Arend Heyting. Die formalen regeln der intuitionistischen logik. *Sitzungsbericht PreuBische Akademie der Wissenschaften*, pages 42–56, 1930.

25 William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

26 Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. First class call stacks: Exploring head reduction. In *Workshop on Continuations*, volume 212 of *WOC*, 2015.

27 Philip Johnson-Freyd, Paul Downen, and Zena M. Ariola. Call-by-name extensionality and confluence. *Journal of Functional Programming*, 27:e12, 2017.

28 Jan W. Klop and Roel C. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80(2):97–113, 1989.

29 Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 482–494. ACM, 2017.

30 Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

31 Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, pages 190–201. Springer-Verlag, 1992.

32 Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, 1991.

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