Introduction and Elimination, Left and Right

KLAUS OSTERMANN, University of Tübingen, Germany
DAVID BINDER, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
TIM SÜBERKRÜB, University of Tübingen, Germany
PAUL DOWNEN, University of Massachusetts Lowell, USA

Functional programming language design has been shaped by the framework of natural deduction, in which language constructs are divided into introduction and elimination rules for *producers* of values. In sequent calculus-based languages, left introduction rules replace (right) elimination rules and provide a dedicated sublanguage for *consumers* of values. In this paper, we present and analyze a wider design space of programming languages which encompasses four kinds of rules: Introduction and elimination, both left and right. We analyze the influence of rule choice on program structure and argue that having all kinds of rules enriches a programmer's modularity arsenal. In particular, we identify four ways of adhering to the principle that "the structure of the program follows the structure of the data" and show that they correspond to the four possible choices of rules. We also propose the principle of *bi-expressibility* to guide and validate the design of rules for a connective. Finally, we deepen the well-known dualities between different connectives by means of the proof/refutation duality.

CCS Concepts: • **Theory of computation** → *Abstract machines*; *Lambda calculus*; *Type theory*; *Proof theory*; *Linear logic*.

Additional Key Words and Phrases: Duality, Sequent Calculus, Natural Deduction

ACM Reference Format:
Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen. 2022. Introduction and Elimination, Left and Right. *Proc. ACM Program. Lang.* 6, ICFP, Article 106 (August 2022), 28 pages. https://doi.org/10.1145/3547637

1 INTRODUCTION

Undoubtedly, the λ-calculus has had a profound impact on functional programming: from language design, to implementation, to practical programming techniques. Through the Curry-Howard correspondence, we know that the λ-calculus — and likewise, λ-based functional languages — are oriented around the interplay between the *introduction* and *elimination* rules of types as first formulated in natural deduction (ND). This natural deduction style of programming nicely allows for a quite "natural" way of combining sub-problems in programs, like basic operations of function composition $f\ (g\ x)$ and swapping the pair x as $(\mathbf{Snd}\ x, \mathbf{Fst}\ x)$. The natural compositional style is afforded by the fact that all expressions in the λ-calculus *produce* exactly one result that is implicitly taken by *exactly one* consumer: namely, the enclosing context of that expression. While the single implicit consumer is natural for composition, it can be rather *unnatural* if we ever want to work

Authors' addresses: Klaus Ostermann, University of Tübingen, Germany, klaus.ostermann@uni-tuebingen.de; David Binder, University of Tübingen, Germany, david.binder@uni-tuebingen.de; Ingo Skupin, University of Tübingen, Germany, skupin@informatik.uni-tuebingen.de; Tim Süberkrüb, University of Tübingen, Germany, tim.sueberkrueb@student.uni-tuebingen.de; Paul Downen, University of Massachusetts Lowell, USA, Paul_Downen@uml.edu.

2475-1421/2022/8-ART106
https://doi.org/10.1145/3547637

with more than one consumer. In these cases, we must reify the implicit consumer to make it explicit — such as resorting to continuation-passing style [Reynolds 1972] or control operators like `call/cc` — which leads to the rather asymmetric situation of having a mix of one special implicit output with additional explicit outputs.

When might functional programmers want to juggle multiple consumers at the same time? Consider the familiar filter function, naturally expressible in any typed functional language:

$$\begin{array}{lll} \mathit{filter} & : (a \to \mathit{Bool}) \to [a] \to [a] \\ \mathit{filter}\ p\ [] & = [] \\ \mathit{filter}\ p\ (x :: xs) & = \textbf{If}\ p\ x\ \textbf{Then}\ x :: \mathit{filter}\ p\ xs \\ & \qquad\qquad\ \textbf{Else}\ \mathit{filter}\ p\ xs \end{array}$$

This function successfully removes any elements from a given list xs which fail the test p, leaving only those elements x for which $(p\ x)$ is true. From a quick inspection of the definition, we can see that the output of this function always fits its specification. So what's wrong? The problem is with *efficiency*. It's quite likely that a large chunk of the filtered output will be the same as the input, and yet *filter* will reallocate a new cons cell for every (::) in the output, even if that same list already exists in memory. As an extreme case, if we filter with the constant function $\mathit{const}\ x\ y = x$, the call $\mathit{filter}\ (\mathit{const}\ \mathit{True})\ xs$ will allocate and return an entirely new list that is equal to xs.

Instead, we would rather that a call like $\mathit{filter}\ (\mathit{const}\ \mathit{True})\ xs$ notices its output will be equal to xs, so that it can return the *exact same* object x that was already allocated in the heap. In a more general case, it may be that only some suffix of xs all passes the test p; if so, $\mathit{filter}\ p\ xs$ should return a list that only allocates new cons cells for the prefix of the list that is changed, connected to an identical pointer to the same suffix of the original xs list in the heap. And in any case, $\mathit{filter}\ p\ xs$ should only traverse the list xs once, and when it has reached the end of the list, it should return immediately and not have to unwind a call-stack first, so it would need to be written in a partially tail-recursive way. This is quite a tall order to satisfy in a conventional functional language. Shivers and Fisher [2004] showed how to express efficient filtering using *multi-return functions*. However, this solution complicates ordinary first-class functions with another concern (multiple different return paths); going against orthogonal language design philosophy of keeping separate features separate.

The classical sequent calculus (SC) can also be interpreted Curry-Howard-style as a prototypical programming language. Sequent-style languages turn several implicit aspects of the λ-calculus into explicit, symmetric entities. There is a context containing many named outputs, dual to the many named inputs in the context of free variables. Besides terms which primarily produce results, there are terms which primarily consume inputs. Elimination rules are replaced by left rules operating on consumers. Computation happens inside of a *command* $\langle e \parallel f \rangle$, which connects the output of a producer e with the input of a consumer f. For instance, the consumer $\mathbf{Fst} \fatsemi f$ should be read as "project the implicit pair to its first component and then continue with f" and is reduced as $\langle (e_1, e_2) \parallel \mathbf{Fst} \fatsemi f \rangle \triangleright \langle e_1 \parallel f \rangle$. Compared to ND, programs in SC have an "inside-out" structure, which Wadler [2003] has compared to the external plumbing of the Pompidou center in Paris. Yet, whereas gazing upon the Pompidou center may be a beautiful sight, sifting through the bureaucratic plumbing of a large-scale sequent-style program is not.

Still, the use of left rules lets the classical sequent calculus express *new kinds of types* which are not expressible in conventional functional languages. These new, exotic types would let us decompose a complex feature like multi-return functions into more basic parts. But must we always suffer through painful bureaucracy to access these types for programming? No! Our insight is that we can combine the best of both natural deduction (introduction versus elimination) and sequent

$$
\begin{array}{ll}
(\circ) & : b \prec \neg(a \longrightarrow b) \prec a \\
\langle x \mid\mid \alpha \circ \mathbf{Not}\ f \rangle & = \langle f\ x \mid\mid \alpha \rangle \\
\\
\mathit{filter} & : (a \longrightarrow \mathit{Bool}) \longrightarrow [a] \longrightarrow [a] \\
\langle \mathit{filter}\ p\ xs \mid\mid \mathit{start} \rangle & = \langle \mathbf{Handle}\ (\mathit{filter/pass}\ p\ xs)\ \mathbf{with}\ (\mathit{start} \circ \mathbf{Not}\ (\mathit{const}\ xs)) \\
 & \quad \mid\mid \mathit{start} \rangle \\
\\
\mathit{filter/pass} & : (a \longrightarrow \mathit{Bool}) \longrightarrow [a] \longrightarrow [a] \parr () \\
\langle \mathit{filter/pass}\ p\ [] \mid\mid [\mathit{diff}, \mathit{same}] \rangle & = \langle () \mid\mid \mathit{same} \rangle \\
\langle \mathit{filter/pass}\ p\ (x :: xs) \mid\mid [\mathit{diff}, \mathit{same}] \rangle & = \mathbf{If}\ p\ x \\
 & \quad \mathbf{Then}\ \langle \mathit{filter/pass}\ p\ xs \mid\mid [\mathit{diff} \circ \mathbf{Not}\ (x ::), \mathit{same}] \rangle \\
 & \quad \mathbf{Else}\ \ \langle \mathit{filter/pass}\ p\ xs \mid\mid [\mathit{diff}, \mathit{diff} \circ \mathbf{Not}\ (\mathit{const}\ xs)] \rangle
\end{array}
$$

Fig. 1. Parsimonious filter function

calculus (left versus right) styles in the same program, making use of exotic new types of control flow that juggle multiple consumers while still enjoying pleasantly natural, functional composition.

We propose having four styles of rules: introductions *and* eliminations on both the left *and* the right. Doing so lets us introduce new ways of programming that keeps all the familiar features we already know as they are, and *also* lets us talk about new types that can't be expressed in conventional functional languages. For example, we can decompose the idea of multi-return functions [Shivers and Fisher 2004] into several orthogonal features: regular functions (of type $T_1 \longrightarrow T_2$), computations juggling two continuations (of type $T_1 \parr T_2$), functions transforming a consumer of T_1s to a consumer of T_2s (of type $T_1 \prec T_2$), and the reversal between producers and consumers (of type $\neg T$).

Together, these features let us write the efficient[1] filtering function using familiar programming concepts (first-class functions and exception handling) and reusable combinators (like *const* above and the following composition $\circ$ of a function with a continuation) based on a function *filter/pass* which either filters a list by removing at least one failing element, or finds element passes (Figure 1). The rest of this paper will go into the detail needed to read, understand, and specify how this example works.

So, should we program in ND style or in SC style? We say: both! The purpose of this work is to analyze and complete the logic calculus design space opened up by ND and SC and investigate the interdependency between logical calculus style and program structure. More specifically, we make the following contributions:

- We investigate the usage of all four kinds of rules, introduction and elimination, both left and right. We identify four different sub-calculi: The *intro calculus* (which corresponds to classical SC), the *right calculus* (which corresponds to ND), the *elimination calculus* (which has only left and right elimination rules), and the *left calculus* (which features left introduction and elimination rules).
- We analyze the influence of these calculi on program structure (and hence modularity, extensibility etc.). Specifically, we argue that the common programming design guideline "the structure of the program follows the structure of the data" can be interpreted in four ways, and that those four ways correspond to the four calculi described above.

[1]We are defining a calculus via a reduction semantics, and that semantics does not feature pointers, so we cannot directly talk about sharing and do not make any practical efficiency claims here. We will clarify that point in Section 7.

- We clarify the relation between the different rules of a connective by the concept of *bi-expressibility*. Informally, bi-expressibility means that the left introduction rule is as powerful as the right elimination rule and the right introduction rule is as powerful as the left elimination rule.
- We deepen the known dualities between connectives by using the *proof/refutation duality* [Tranchini 2012]. Specifically, we show that the typing, term-level representation, and reduction of "dual" connectives can be derived mechanically, which gives rise to the possibility of a new form of *consumer/producer polymorphism*, in which a term can be interpreted as both a producer of type T or as a consumer of the dual of T.

All theorems presented in this paper have been mechanized and proven in Coq (submitted as supplementary material).

The remainder of this paper is structured as follows: In Section 2, we give an informal introduction to the language framework we present and describe the influence of rule choice on program structure. In Section 3, we present a small core language featuring functions as well as positive sums and products. In Section 4, we introduce bi-expressibility and present an operational semantics of the language. In Section 5, we demonstrate how to mechanically derive the dual connectives (cofunctions, negative sums and products) and elaborate on the idea of duality polymorphism. In Section 6 we describe extensions of the calculus framework with logical constants and universal and existential types. In Section 7.2 we present examples to illustrate the new possibilities of programming in a language with all four kinds of rules. Section 8 presents related and future work and Section 9 concludes.

2 MOTIVATION

To get started, let's analyze the interaction between logical structure and program structure. As an example, consider the connective $\oplus$ as an algebraic data type in a typical functional language. Values of the type $X \oplus Y$ are built using the constructors **Left** and **Right** — these correspond nicely to the two introduction rules of $X \oplus Y$ in natural deduction. To use values of type $X \oplus Y$, we can employ a **Case**-expression that pattern-matches on the two possible alternatives — likewise corresponding exactly to natural deduction's elimination rule for $X \oplus Y$. Using these tools, we can swap these two options — transforming an unknown value $z : X \oplus Y$ into a result of $Y \oplus X$ — by matching on the originally-chosen option and replacing it with the opposite constructor like so:

$$\begin{aligned} \textbf{Case}\ z\ \{\ \textbf{Left}\ x \quad &\mapsto \textbf{Right}\ x; \\ \textbf{Right}\ y &\mapsto \textbf{Left}\ y\ \} \end{aligned}$$

Rather than always thinking of producing some (implicit) output using introductions and eliminations, a sequent-based language does away with eliminations altogether. Instead, it uses the dichotomy between *producers* which implicitly return some result as an output — just like we are used to in conventional functional programming — versus *consumers* which implicitly take an input to use analogous to continuations. The same swapping operation — converting an unknown $z : X \oplus Y$ into a $Y \oplus X$ — can be written in sequent style as:

$$\begin{aligned} \langle z \mid\mid \textbf{Match}\ \{\ \textbf{Left}\ x \quad &\mapsto \langle \textbf{Right}\ x \mid\mid \alpha \rangle \\ \textbf{Right}\ y &\mapsto \langle \textbf{Left}\ y \mid\mid \alpha \rangle \} \rangle \end{aligned}$$

Notice the several differences in this version of the same program. The top-most operation is a *command* of the form $\langle e \mid\mid f \rangle$ which connects the implicit output returned from a producer e into the implicit input expected by a consumer f. Unlike producers and consumers, commands have no implicit input or output. Rather than the **Case** expression — which has an explicit input

Table 1. Four different ways to swap the components of $z : X \oplus Y$ and send to consumer $\alpha : Y \oplus X$

Calculus	Program
Right	$\mathbf{Case}\ z\ \{\mathbf{Left}\ x \mapsto \langle \mathbf{Right}\ x \parallel \alpha\rangle; \mathbf{Right}\ y \mapsto \langle \mathbf{Left}\ y \parallel \alpha\rangle\}$
Intro	$\langle z \parallel \mathbf{Match}\ \{\mathbf{Left}\ x \mapsto \langle \mathbf{Right}\ x \parallel \alpha\rangle; \mathbf{Right}\ y \mapsto \langle \mathbf{Left}\ y \parallel \alpha\rangle\}\rangle$
Left	$\langle z \parallel \mathbf{Match}\ \{\mathbf{Left}\ x \mapsto \langle x \parallel \mathbf{Right} \fatsemi \alpha\rangle; \mathbf{Right}\ y \mapsto \langle y \parallel \mathbf{Left} \fatsemi \alpha\rangle\}\rangle$
Elim	$\mathbf{Case}\ z\ \{\mathbf{Left}\ x \mapsto \langle x \parallel \mathbf{Right} \fatsemi \alpha\rangle; \mathbf{Right}\ y \mapsto \langle y \parallel \mathbf{Left} \fatsemi \alpha\rangle\}$

used to implicitly produce some result — we use a **Match**, which forms a new consumer implicitly expecting an input of type $X \oplus Y$. The **Match** consumer has two branches pattern-matching on its implicit input — one for each possibility between **Left** and **Right** — and because consumers have no implicit output, both possible branches lead to commands. In either case, the swapped sum value is explicitly "returned" (that is, passed via a command) to α, which gives a name to the previously-implicit output of the whole operation.

Under our analysis of comparing different structures of logic — and their impact on their corresponding programs — natural deduction has only *right* rules. In other words, every rule of natural deduction is concerned with concluding the *truth* of propositions (traditionally written on the right-hand side of the hypothetical turnstyle $\vdash$, hence the name "right"). By the proofs-as-programs paradigm, this corresponds to the fact that every primitive tool that typical functional languages have for working with its various types of information — both introductions and eliminations — are inherently concerned with *producing* something. The constructors **Left** and **Right** *produce* unique values of the sum type $X \oplus Y$. The **Case** expression uses an $X \oplus Y$ value, yes, but only in the service of *producing* some other result (which may not necessarily be another sum type). In contrast, the application of the sequent calculus as the basis for a programming language has revealed a different way of organizing programs. Instead, it pairs rules working on the *right* (which look just like natural deduction's introduction rules) with rules working on the *left*. These left rules can be seen as ways to *refute* propositions, which are concerned with the *falsehood* of propositions (or equivalently, with *assumed truth* of propositions, traditionally written on the left-hand side of $\vdash$, hence the name "left"). But importantly, no matter which side of the divide the rules are focused, the sequent calculus has only *introduction* rules.

We can compare these two logical styles of programming — one based on natural deduction and the other on the sequent calculus — by modifying the elimination rule slightly. When in the context of a command with an explicit consumer named α, a **Case** does not need to implicitly produce some result, but can instead indicate what command to execute next by sending either result to α. In effect, this "absorbs" the consumer α into the **Case** like so:

$$\langle \mathbf{Case}\ e\ \{\mathbf{Left}\ x \mapsto e_1;\ \mathbf{Right}\ y \mapsto e_2\} \parallel \alpha\rangle = \mathbf{Case}\ e\ \{\mathbf{Left}\ x \mapsto \langle e_1 \parallel \alpha\rangle; \mathbf{Right}\ y \mapsto \langle e_2 \parallel \alpha\rangle\}$$

With this in mind, we present the two styles of sum-swapping in Table 1, in each case connecting an input $z : X \oplus Y$ to an output $\alpha : Y \oplus X$. The first line uses only right rules (both introduction and eliminations on the right) in a typical functional style. The next line illustrates how the same program can be written using only introduction rules (both on the left and the right) in sequent style. The gray parts indicate how we can connect the program to an explicitly given producer z.

But what about other combinations of rules? If we can make due with only right rules or only introductions, can we also write the same program using only left rules and only eliminations? Yes! The third line shows again the same program but this time using only left rules. $\mathbf{Left} \fatsemi \alpha$ and $\mathbf{Right} \fatsemi \alpha$ are the left elimination rules for a consumer α of type $Y \oplus X$. $\mathbf{Left} \fatsemi \alpha$ should be read as:

Table 2. Computation from $x : (\top \mathbin{\&} X) \mathbin{\&} \top$ to $\alpha : \bot \oplus ((X \oplus \bot) \oplus \bot)$

Calculus	Program	Program Structure
Right	$\langle \mathbf{Right}\ (\mathbf{Left}\ (\mathbf{Left}\ (\mathbf{Snd}\ (\mathbf{Fst}\ x))) \mid\mid \alpha \rangle$	α outside-in, x inside-out
Intro	$\langle x \mid\mid \mathbf{Fst} \fatsemi (\mathbf{Snd} \fatsemi \tilde{\mu} y.\langle \mathbf{Right}\ (\mathbf{Left}\ (\mathbf{Left}\ y)) \mid\mid \alpha \rangle) \rangle$	x outside-in, α outside-in
Left	$\langle x \mid\mid \mathbf{Fst} \fatsemi (\mathbf{Snd} \fatsemi (\mathbf{Left} \fatsemi (\mathbf{Left} \fatsemi (\mathbf{Right} \fatsemi \alpha)))) \rangle$	x outside-in, α inside-out
Elim	$\langle \mathbf{Snd}\ (\mathbf{Fst}\ x) \mid\mid \mathbf{Left} \fatsemi (\mathbf{Left} \fatsemi (\mathbf{Right} \fatsemi \alpha)) \rangle$	α inside-out, x inside-out

Inject the implicit value into the left component of a sum and then continue with α.[2] Finally, the last line shows how the program can be formulated using only elimination rules. As we will later see, every program can be written in each of these styles, and there is a simple and systematic way to transition between them.

But what difference does it make which style we choose?

One of the main principles of programming that is taught in most introductory programming courses (such as Felleisen et al. [2001]) is that the structure of a program follows the structure of its input. A less common but equally valid principle is that the structure of a program follows the structure of its output [Gibbons 2021]. Both styles can also be reversed in that a program may also be "inside-out", that is, it follows the structure of the input or output from the inside to the outside and not vice versa. Such a structure is common, for instance, in continuation-passing style. The choice of style has a major influence on the modularity properties of the program: How easily it can be read and understood, how extensible it is, and so forth.

The observation that motivated this work is that the choice of rules determines which of these styles our programs will naturally have. We illustrate this in Table 2, which shows four ways of projecting out of a nested product type $(\top \mathbin{\&} X) \mathbin{\&} \top$ and injecting into a nested sum type $\bot \oplus ((X \oplus \bot) \oplus \bot)$. Our interest here is in four different calculi corresponding exactly to the four different program styles illustrated in Table 2.

The Intro variant uses Curien and Herbelin [2000]'s $\tilde{\mu}$ operator to reify the currently consumed value as a variable, which brings us to the last difference between the calculi that we want to point out. Producer expressions have explicit inputs (given by variables) and an implicit output (the current continuation).[3] Consumer expressions have explicit outputs (given by covariables) and an implicit input. Whenever we need to construct programs where the flow of the respective implicit input/output does not match the nesting structure of the program, we need to resort to μ or $\tilde{\mu}$ operators to reify that implicit input/output.

Here is another example: With right elimination rules, the composition of two functions g and h is easy to express: $\langle h\ (g\ x) \mid\mid \alpha \rangle$. However, when we have to express the same program using the left introduction rule for functions instead (which constructs a pair $e \cdot f$ consisting of a function argument e and a consumer f for the function result), we have to express the program as $\langle g \mid\mid x \cdot (\tilde{\mu} y.\langle h \mid\mid y \cdot \alpha \rangle) \rangle$. The "intermediate result" $g\ x$ must be given a name y because the flow of data does not correspond to the structure of the left introduction rule.

Last but not least, let's consider how modularity and extensibility depend on the program style we choose. We analyze the nesting structure of the term in relation to the nesting structure of the type. Informally, outside-in means that the term is nested similar to the type: subterms of the type correspond to subterms of the term. Inside-out means that inner nodes of the term structure

[2]The $\fatsemi$ operator can be thought of as left-to-right composition, i.e. $f \fatsemi g$ means "apply f, then g." The symbol was chosen to be a combination of regular right-to-left composition $\circ$ and a semicolon.

[3]In this example, we could have alternatively reified in the other direction using the dual μ operator, as $\langle \mathbf{Right}\ (\mathbf{Left}\ (\mathbf{Left}\ \mu\beta.\langle x \mid\mid \mathbf{Fst} \fatsemi (\mathbf{Snd} \fatsemi \beta) \rangle)) \mid\mid \alpha \rangle$. These two commands are contextually equivalent to one another.

Syntax

T	$::=$	X	*Types*
e	$::=$	$x \mid \mu\alpha.c$	*Producers*
f	$::=$	$\alpha \mid \tilde{\mu}x.c$	*Consumers*
c	$::=$	$\langle e \mid\mid f \rangle$	*Commands*
v	$::=$	x	*Values*
$\mathcal{E}[]$	$::=$	ϵ	*Focusing Context*
$\mathcal{F}[]$	$::=$	ϵ	*Cofocusing Context*
Γ	$::=$	$x : T, \Gamma \mid \epsilon$	*Producer Context*
Δ	$::=$	$\alpha : T, \Delta \mid \epsilon$	*Consumer Context*

Reduction

$$\langle v \mid\mid \tilde{\mu}x.c \rangle \quad \triangleright_{\tilde{\mu}} \quad c\{x := v\}$$
$$\langle \mu\alpha.c \mid\mid f \rangle \quad \triangleright_{\mu} \quad c\{\alpha := f\}$$

In ς rules, x is fresh, $e \notin v$.

$$\langle \mathcal{E}[e] \mid\mid f \rangle \quad \triangleright_{\varsigma} \quad \langle e \mid\mid \tilde{\mu}x.\langle \mathcal{E}[x] \mid\mid f \rangle \rangle$$
$$\langle v \mid\mid \mathcal{F}[e] \rangle \quad \triangleright_{\varsigma} \quad \langle e \mid\mid \tilde{\mu}x.\langle v \mid\mid \mathcal{F}[x] \rangle \rangle$$

Typing rules

$$\frac{x : T \ \in \Gamma}{\Gamma \vdash x : T \mid \Delta}\ (\text{R-VAR}) \qquad \frac{\alpha : T \ \in \Delta}{\Gamma \mid \alpha : T \vdash \Delta}\ (\text{L-VAR})$$

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash \mu\alpha.c : T \mid \Delta}\ (\text{R-}\mu) \qquad \frac{c : (x : T, \Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : T \vdash \Delta}\ (\text{L-}\mu) \qquad \frac{\Gamma \vdash e : T \mid \Delta \quad \Gamma \mid f : T \vdash \Delta}{\langle e \mid\mid f \rangle : \Gamma \vdash \Delta}\ (\text{CUT})$$

Fig. 2. Core language

represent outer nodes of the type structure. In general, introduction rules yield terms nesting outside-in because the introduced type appears in the conclusion of the rule, while elimination rules induce an inside-out nesting as the eliminated type occurs in the premise of the rule.

When we use elimination forms on the left to introduce a type on the right or vice-versa we therefore reverse the nesting structure of the program and thereby also alter its modularity properties.

Consider the programs in Table 2 again. In the right calculus row, we destruct x inside-out with right elimination rules and then construct a producer with the type required by α outside-in using right introduction rules. The situation is reversed in the left calculus row. Here we destruct the continuation α inside-out with left elimination rules and construct a continuation with the type required by x outside-in using left introduction rules. Similarly, we can get any other combination of nesting orders by choosing one of the other calculi.

To summarize, having a choice of all four kinds of rules makes it easier for the programmer to choose a program structure that has the desired modularity and extensibility properties and maximizes the usage of implicit producers/consumers to avoid the naming of intermediate results and the associated CPS-like program structure.

3 INTRODUCTION VERSUS ELIMINATION, LEFT VERSUS RIGHT

In this and the next section, we present our language framework, divided into logical steps and parts. As the first step, we present a core language of inputs, outputs, and interactions, without any logical connectives[4] (Figure 2).

As usual in presentations of computational sequent calculi, we have three kinds of sequents: $\Gamma \vdash e : T \mid \Delta$ describes a producer term e that produces an output of type T in variable context Γ and covariable context Δ. Symmetrically, $\Gamma \mid f : T \vdash \Delta$ describes a consumer term f that consumes

[4]The core is similar to the presentation in Sec. 4 of Downen and Ariola [2018]

an input of type T. Finally, a command $c : (\Gamma \vdash \Delta)$ describes a complete and executable program. The core language only contains cut commands $\langle e \mid\mid f \rangle$.

With regard to reduction, we opt to use the standard call-by-value evaluation strategy, which involves prioritizing producers before consumers in cuts [Downen and Ariola 2018]. Alternatively, we could have chosen call-by-name evaluation in the standard way by reversing this priority [Wadler 2003]. The purpose of the focusing contexts $\mathcal{E}$ and $\mathcal{F}$ (which are empty in the core language) and the $\triangleright_\varsigma$ reduction rules are to push pending computations embedded in subterms to the top-level. This is also standard [Downen and Ariola 2018; Wadler 2003].

In Figure 3 we extend the language with three connectives: $\to$ (functions), $\oplus$ (positive sums), and $\otimes$ (positive products)[5]. The adjective "positive" comes from polarized type theory [Andreoli 1992; Zeilberger 2009] and denotes data types that are defined via constructors, as opposed to negative types, which are codata types that are defined via destructors. Positive types are evaluated eagerly, when constructed, whereas negative types are evaluated on demand, when they are destructed. In this paper, we use blue for positive and red for negative connectives.

All connectives come with all four kinds of rules: introduction and elimination, both left and right. Like in linear logic, we make a clear distinction between positive and negative connectives (the latter will be shown later); it turns out that for our bi-expressibility property (introduced in Section 4) it is essential that rules do not duplicate or destroy information. That property also necessitates a few generalizations of standard rules.

For instance, the syntax of λ abstraction is $\lambda(x \cdot \alpha).c$ rather than $\lambda x.e$; the latter can be encoded as $\lambda x.e := \lambda(x \cdot \alpha).\langle e \mid\mid \alpha \rangle$. The left introduction rule for $\to$ consists of a function argument and a consumer for the function's returned result. The left elimination rule allows us to inspect both components of that pair, yielding a command.

The rule R-$\oplus$-Intro$_i$ is standard. R-$\oplus$-Elim uses again commands in its branches and is in fact itself a command. An encoding for the more standard **Case** e $\{\overline{\mathbf{In}_i\ x_i \mapsto e_i}\}$ can be given as $\mu\alpha.(\mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto \langle e_i \mid\mid \alpha \rangle}\})$. The reason for the deviation from the standard is symmetry with the L-$\oplus$-Intro rule, which has the same structure except that the value to be pattern-matched on is implicit. L-$\oplus$-Elim$_i$ has also been designed to be symmetric to R-$\oplus$-Intro$_i$.

The rules for $\otimes$ follow the same design principles. Of particular note are the L-$\otimes$-Elim$_i$ rules. Their names **Handle**$_i$ are motivated by $\parr$ (the dual of $\otimes$), whose right elimination rules are reminiscent of error handlers as shown in Figure 1.

The reduction rules cover only the introduction forms of the constructs. We will see in the next section why that is sufficient.

This language and all its extensions we are about to present, has four subcalculi identified by considering the languages having only introduction rules (the Intro calculus), only elimination rules (the Elim calculus), only right rules (the Right calculus), or only left rules (the Left calculus).

We have proven (mechanized in Coq) standard type safety theorems for this language, but before we can talk about the details we need to introduce bi-expressibility.

[5]In the formal calculus, we use the constructor names $\mathbf{In}_1$/$\mathbf{In}_2$ and $\mathbf{Out}_1$/$\mathbf{Out}_2$ instead of **Left**/**Right** and **Fst**/**Snd** for both the $\oplus$ and the & type to facilitate our exposition of duality in Section 5. A version of the examples from Section 2 in formal calculus syntax can be found in Appendix A.

Syntax

$$
\begin{array}{lcll}
T & ::= & \ldots & \mid T \to T \mid T \oplus T \mid T \otimes T \\
e & ::= & \ldots & \mid \lambda(x \cdot \alpha).c \mid e\ e \mid \mathbf{In}_i\ e \mid [e, e] \\
f & ::= & \ldots & \mid e \cdot f \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \mid \mathbf{Out}_i\ f \mid \mathbf{Match}\ \{[x,x] \mapsto c\} \mid \mathbf{Handle}_i\ f\ \mathbf{with}\ e \\
c & ::= & \ldots & \mid \mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\} \mid \mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \mid \mathbf{Case}\ e\ \{[x,x] \mapsto c\} \\
v & ::= & \ldots & \mid \lambda(x \cdot \alpha).c \mid \mathbf{In}_i\ v \mid [v, v] \\
\mathcal{E}[\,] & ::= & \ldots & \mid [\Box, e] \mid [v, \Box] \mid \mathbf{In}_i\ \Box \\
\mathcal{F}[\,] & ::= & \ldots & \mid \Box \cdot f
\end{array}
$$

Typing

$$\frac{\Gamma\ \vdash e : T_1 \mid \Delta \qquad \Gamma \mid f : T_2 \vdash \Delta}{\Gamma \mid e \cdot f : T_1 \to T_2 \vdash \Delta}\ (\text{L-}\to\text{-Intro}) \qquad \frac{c : (x : T_1, \Gamma \vdash \alpha : T_2, \Delta)}{\Gamma \vdash \lambda(x \cdot \alpha).c : T_1 \to T_2 \mid \Delta}\ (\text{R-}\to\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \to T_2 \vdash \Delta \qquad c : (\Gamma, x : T_1\ \vdash \alpha : T_2, \Delta)}{\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\} : (\Gamma\ \vdash \Delta)}\ (\text{L-}\to\text{-Elim}) \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \mid \Delta \qquad \Gamma \vdash e_2 : T_1 \mid \Delta}{\Gamma \vdash e_1\ e_2 : T_2 \mid \Delta}\ (\text{R-}\to\text{-Elim})$$

$$\frac{\forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\Gamma \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : T_1 \oplus T_2 \vdash \Delta}\ (\text{L-}\oplus\text{-Intro}) \qquad \frac{\Gamma \vdash e : T_i \mid \Delta}{\Gamma \vdash \mathbf{In}_i\ e : T_1 \oplus T_2 \mid \Delta}\ (\text{R-}\oplus\text{-Intro}_i)$$

$$\frac{\Gamma \mid f : T_1 \oplus T_2 \vdash \Delta}{\Gamma \mid \mathbf{Out}_i\ f : T_i \vdash \Delta}\ (\text{L-}\oplus\text{-Elim}_i) \qquad \frac{\Gamma \vdash e : T_1 \oplus T_2 \mid \Delta \qquad \forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\mathbf{Case}\ e\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : (\Gamma \vdash \Delta)}\ (\text{R-}\oplus\text{-Elim})$$

$$\frac{c : (x : T_1, y : T_2, \Gamma \vdash \Delta)}{\Gamma \mid \mathbf{Match}\ \{[x,y] \mapsto c\} : T_1 \otimes T_2 \vdash \Delta}\ (\text{L-}\otimes\text{-Intro}) \qquad \frac{\Gamma \vdash e_1 : T_1 \mid \Delta \qquad \Gamma \vdash e_2 : T_2 \mid \Delta}{\Gamma \vdash [e_1, e_2] : T_1 \otimes T_2 \mid \Delta}\ (\text{R-}\otimes\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \otimes T_2 \vdash \Delta \qquad \Gamma \vdash e : T_i \mid \Delta}{\Gamma \mid \mathbf{Handle}_i\ f\ \mathbf{with}\ e : T_{2-i+1} \vdash \Delta}\ (\text{L-}\otimes\text{-Elim}_i) \qquad \frac{\Gamma \vdash e : T_1 \otimes T_2 \mid \Delta \qquad c : (x : T_1, y : T_2, \Gamma\ \vdash \Delta)}{\mathbf{Case}\ e\ \{[x,y] \mapsto c\} : (\Gamma\ \vdash \Delta)}\ (\text{R-}\otimes\text{-Elim})$$

Reduction

$$
\begin{array}{lll}
\langle \lambda(x \cdot \alpha).c \mid\mid v \cdot f \rangle & \triangleright_\beta & c\{x := v, \alpha := f\} \\
\langle\ \mathbf{In}_j\ v \mid\mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\}\ \rangle & \triangleright_\beta & c_j\{x_j := v\} \\
\langle [v_1, v_2] \mid\mid \mathbf{Match}\ \{[x,y] \mapsto c\} \rangle & \triangleright_\beta & c\{x := v_1, y := v_2\}
\end{array}
$$

Fig. 3. Syntax, typing and reduction for functions, positive sums, and positive products

4 BI-EXPRESSIBILITY AND SOUNDNESS

Our language has been designed in such a way that all four sub-calculi are in a sense equally powerful. This is made precise in Figure 4, which shows that each typing rule is derivable using only core constructs and the diagonally opposing rule (left intro is diagonally opposing to right elim and vice versa left elim to right intro):

Theorem 4.1 (Bi-Expressibility). *Every typing rule of a connective can be encoded using the diagonally opposing rule and core constructs only.*

Proof. Simple inspection and type-checking of the encoding rules in Figure 4. □

Corollary 4.2 (Subcalculus Restriction). *Given any well-typed command* $c : (\Gamma \vdash \Delta)$*, producer* $\Gamma \vdash e : T \mid \Delta$*, or consumer* $\Gamma \mid f : T \vdash \Delta$ *and any subcalculus* $\mathfrak{C} \in \{\mathit{Intro}, \mathit{Elim}, \mathit{Left}, \mathit{Right}\}$*, there exists a translation result* $c' : (\Gamma \vdash \Delta)$*,* $\Gamma \vdash e' : T \mid \Delta$*, or* $\Gamma \mid f' : T \vdash \Delta$ *which uses only the syntax available in subcalculus* $\mathfrak{C}$*.*

$$\frac{\Gamma \vdash e : T_1 \mid \Delta \quad \Gamma \mid f : T_2 \vdash \Delta}{\Gamma \mid \tilde{\mu}x.\langle x\ e \mid\mid f\rangle : T_1 \to T_2 \vdash \Delta}\ (\text{L-}\to\text{-Intro})$$

$$\frac{c : (x : T_1, \Gamma \vdash \alpha : T_2, \Delta)}{\Gamma \vdash \mu\beta.\mathbf{Case}\ \beta\ \{x \cdot \alpha \mapsto c\} : T_1 \to T_2 \mid \Delta}\ (\text{R-}\to\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \to T_2 \vdash \Delta \quad c : (\Gamma, x : T_1 \vdash \alpha : T_2, \Delta)}{\langle \lambda(x \cdot \alpha).c \mid\mid f\rangle : (\Gamma \vdash \Delta)}\ (\text{L-}\to\text{-Elim})$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \mid \Delta \quad \Gamma \vdash e_2 : T_1 \mid \Delta}{\Gamma \vdash \mu\alpha.\langle e_1 \mid\mid e_2 \cdot \alpha\rangle : T_2 \mid \Delta}\ (\text{R-}\to\text{-Elim})$$

$$\frac{\forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.\ \mathbf{Case}\ x\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} : T_1 \oplus T_2 \vdash \Delta}\ (\text{L-}\oplus\text{-Intro})$$

$$\frac{\Gamma \vdash e : T_i \mid \Delta}{\Gamma \vdash \mu\alpha.\ \langle\ e \mid\mid \mathbf{Out}_i\ \alpha\ \rangle : T_1 \oplus T_2 \mid \Delta}\ (\text{R-}\oplus\text{-Intro}_i)$$

$$\frac{\Gamma \mid f : T_1 \oplus T_2 \vdash \Delta}{\Gamma \mid \tilde{\mu}x.\ \langle\ \mathbf{In}_i\ x \mid\mid f\ \rangle : T_i \vdash \Delta}\ (\text{L-}\oplus\text{-Elim}_i)$$

$$\frac{\Gamma \vdash e : T_1 \oplus T_2 \mid \Delta \quad \forall i, c_i : (\Gamma, x_i : T_i \vdash \Delta)}{\langle\ e \mid\mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\}\ \rangle : (\Gamma \vdash \Delta)}\ (\text{R-}\oplus\text{-Elim})$$

$$\frac{c : (x : T_1, y : T_2, \Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}z.\mathbf{Case}\ z\ \{[x, y] \mapsto c\} : T_1 \otimes T_2 \vdash \Delta}\ (\text{L-}\otimes\text{-Intro})$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid \Delta \quad \Gamma \vdash e_2 : T_2 \mid \Delta}{\Gamma \vdash \mu\alpha.\langle e_1 \mid\mid \mathbf{Handle}_2\ \alpha\ \mathbf{with}\ e_2\ \rangle : T_1 \otimes T_2 \mid \Delta}\ (\text{R-}\otimes\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \otimes T_2 \vdash \Delta \quad \Gamma \vdash e : T_1 \mid \Delta}{\Gamma \mid \tilde{\mu}x.\langle [e, x] \mid\mid f\rangle : T_2 \vdash \Delta}\ (\text{L-}\otimes\text{-Elim}_1)$$

$$\frac{\Gamma \mid f : T_1 \otimes T_2 \vdash \Delta \quad \Gamma \vdash e : T_2 \mid \Delta}{\Gamma \mid \tilde{\mu}x.\langle [x, e] \mid\mid f\rangle : T_1 \vdash \Delta}\ (\text{L-}\otimes\text{-Elim}_2)$$

$$\frac{\Gamma \vdash e : T_1 \otimes T_2 \mid \Delta \quad c : (x : T_1, y : T_2, \Gamma \vdash \Delta)}{\langle e \mid\mid \mathbf{Match}\{[x, y] \mapsto c\}\rangle : (\Gamma \vdash \Delta)}\ (\text{R-}\otimes\text{-Elim})$$

Fig. 4. Bi-Expressibility: Diagonal encodings. New (co)variable names are always assumed to be fresh.

Proof. We give a translation function (formalized in Coq) that restricts the command, producer or consumer to the subcalculus $\mathfrak{C}$ based on Theorem 4.1 and show that it preserves type soundness. □

One thing to note about the encodings is that, when applied to a full program, they will introduce administrative redexes. For instance, when applying the encodings to the examples in Section 2 (we now use the examples in formal calculus syntax from Appendix A), then the encoding of the right elimination construct by left introduction gives us directly the second line, whereas applying the encoding of right introduction by left elimination gives as an administrative redex of the form $\langle \mu\beta.\langle x \mid\mid \mathbf{Out}_2\ \beta\rangle \mid\mid \alpha\rangle$, which requires a reduction step to $\langle x \mid\mid \mathbf{Out}_2\ \alpha\rangle$ to yield the third line.

Another thing to note is that the notion of value depends on the subcalculus. Since we only define reduction for the Intro calculus, our value definition is tailored for it. A term like a λ-abstraction is a value in the Intro calculus; when it gets encoded into the Left or Elim calculus by the encoding rule, it turns into a μ-abstraction - which looks superficially as if we would turn a value into a non-value. But this is misleading, since each calculus would have its own definition of value, if we would define them completely separately. Instead, we use bi-expressibility to give reduction rules for the introduction constructs only: We assume that the elimination rules are "desugared" using the encodings in Figure 4.

For the language where the elimination forms are encoded as in Figure 4, we have also proven (in Coq) standard progress and preservation theorems.

Theorem 4.3 (Preservation). *For all commands c and all typing contexts Γ and Δ, if $c : (\Gamma \vdash \Delta)$ and $c \triangleright c'$, then $c' : \Gamma \vdash \Delta$.*

For the statement of the progress theorem, it is convenient to have a **Done** command with typing axiom $\mathbf{Done} : \Gamma \vdash \Delta$, such that there are non-trivial closed commands:

Theorem 4.4 (Progress). *For all closed commands c and all typing contexts Γ and Δ, if $c : (\Gamma \vdash \Delta)$, then either $c = \mathbf{Done}$ or there exists c' such that $c \triangleright c'$.*

Furthermore, reduction is deterministic:

Theorem 4.5 (Deterministic reduction). *For all commands c, c', c'' and all typing contexts Γ and Δ, if $c : (\Gamma \vdash \Delta)$, $c \triangleright c'$, and $c \triangleright c''$, then $c' = c''$.*

Let us briefly illustrate why bi-expressibility requires some generalizations of introduction or elimination forms of standard constructs; in particular, why we have replaced expressions by commands in some places. As an example, the standard right introduction rule for $\to$ is $\lambda x.e$, whereas we use $\lambda(x \cdot \alpha).c$. If we look at R-$\to$-Intro in Figure 4, then $\lambda x.e$ could be encoded as $\mu\beta.\mathbf{Case}\ \beta\ \{x \cdot \alpha \mapsto \langle e \mid\mid \alpha\rangle\}$. However, L-$\to$-Elim would then be stronger than R-$\to$-Intro and the L-$\to$-Elim encoding in Figure 4 would no longer work; if we wanted to reverse the transformation, we cannot guarantee that e does not have α as free variable. Weakening L-$\to$-Elim to $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto f\}$ instead of $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto c\}$ would not help; both constructs would be weaker, but they would not be bi-expressible. $\mathbf{Case}\ f\ \{x \cdot \alpha \mapsto e\}$, on the other hand, would be rather useless in the left calculus, since the only valid (non-μ) producer expression is a variable.

5 PROOF/REFUTATION DUALITY

Our setting with all four kinds of rules lets us use the proof/refutation duality [Tranchini 2012]. The proof/refutation duality is a duality between proofs of a statement and refutations of the dual statement. Tranchini [2012] defined a natural deduction calculus of refutation which is completely isomorphic to the standard natural deduction calculus of proofs. For instance, the introduction and elimination rules for refutations of disjunctions (written $\vee_R$), are identical to the ones for proofs of conjunctions — another facet of the duality between disjunction and conjunction:

$$\frac{\Gamma \vdash T_1 \qquad \Gamma \vdash T_2}{\Gamma \vdash T_1 \vee_R T_2}\ (\vee_R\text{-Intro}) \qquad\qquad \frac{\Gamma \vdash T_1 \vee_R T_2}{\Gamma \vdash T_i}\ (\vee_R\text{-Elim}_i)$$

The introduction rule should be read as "if T_1 is false and T_2 is false, then $T_1 \vee T_2$ is false", the elimination rules as "if $T_1 \vee T_2$ is false, then T_i is false". Tranchini has shown that a full (intuitionistic) refutation calculus containing all standard propositional connectives can be defined in such a way that the rules are mirrors of the respective dual connectives in the proof calculus. The duality between disjunction and conjunction as well as between the logical constants $\top$ and $\bot$ are standard; dualizing implication requires the less common notion of co-implication [Tranchini 2012], also known as subtraction [Crolard 2004] or difference [Curien and Herbelin 2000] (sometimes with reversed order of arguments).

Tranchini's work is purely in the logic domain and does not discuss programming, but from the perspective of programming, a refutation calculus can be seen as a language of consumers or continuations. The close symmetry between the corresponding proof and refutation calculi suggests that the term language of the proof calculus can also be used as a term language of the refutation calculus. In other words, we can have different interpretations of the same term: once as a producer and once as a consumer.

Figure 5 illustrates the idea by defining typing rules for $\prec$, $\&$, and $⅋$, the duals of $\to$, $\oplus$, and $\otimes$, respectively. What is noteworthy about these rules is that they are completely determined by the typing rules of their respective duals. In fact, the rules in Figure 5 could be replaced by fusing the syntactic categories e and f,

$$\begin{array}{lcl} e & ::= & x \mid \mu x.c \mid \tilde{\mu} x.c \mid e \cdot e \mid \textbf{Match}\ \{\overline{\textbf{In}_i\ x_i \mapsto c_i}\} \mid \textbf{Out}_i\ e \mid \textbf{Match}\ \{[x,x] \mapsto c\} \\ & & \mid \textbf{Handle}_i\ e\ \textbf{with}\ e \mid \lambda(x \cdot z).c \mid e\ e \mid \textbf{In}_i\ e \mid [e,e] \\ x, \alpha & ::= & \textit{identifier} \end{array}$$

and adding these two rules

$$\frac{\Delta^\circ \mid e^\circ : T^\circ \vdash \Gamma^\circ}{\Gamma \vdash e : T \mid \Delta} \qquad\qquad \frac{\Delta^\circ \vdash e^\circ : T^\circ \mid \Gamma^\circ}{\Gamma \mid e : T \vdash \Delta}$$

where T° is defined as

$$\begin{array}{llllll} X^\circ & = & X & & & \\ (T_1 \to T_2)^\circ & = & T_1^\circ \prec T_2^\circ & (T_1 \prec T_2)^\circ & = & T_1^\circ \to T_2^\circ \\ (T_1 \otimes T_2)^\circ & = & T_1^\circ ⅋ T_2^\circ & (T_1 ⅋ T_2)^\circ & = & T_1^\circ \otimes T_2^\circ \\ (T_1 \,\&\, T_2)^\circ & = & T_1^\circ \oplus T_2^\circ & (T_1 \oplus T_2)^\circ & = & T_1^\circ \,\&\, T_2^\circ\,, \end{array}$$

Γ° is the obvious extension to typing contexts, and e° creates a copy of the expression that is identical except that $\langle e_1 \mid\mid e_2 \rangle^\circ = \langle e_2^\circ \mid\mid e_1^\circ \rangle$ where e contains a command.

A fully shared and symmetric term syntax between producers and consumers would enable an exciting new feature we call *consumer/producer polymorphism*. It becomes possible to have libraries of code that can be used as both producers and consumers.

For instance, λ-abstraction is the common way to abstract over certain code patterns, and that is true regardless of whether the code consumes or produces values. A generic function like the one for function composition, $\lambda f.\lambda g.\lambda x.f\ (g\ x)$, which desugars to

$$comp = \lambda(f \cdot \alpha).\langle \lambda(g \cdot \beta).\langle \lambda(x \cdot \gamma).\langle f\ (g\ x) \mid\mid \gamma \rangle \mid\mid \beta \rangle \mid\mid \alpha \rangle$$

can be used both as a producer of type $(Y \to Z) \to (X \to Y) \to X \to Z$ and, via

$$comp^\circ = \lambda(f \cdot \alpha).\langle \alpha \mid\mid \lambda(g \cdot \beta).\langle \beta \mid\mid \lambda(x \cdot \gamma).\langle \gamma \mid\mid f\ (g\ x) \rangle \rangle \rangle$$

Syntax

$$
\begin{array}{lcll}
T & ::= & \ldots & \mid T \prec T \mid T \,\&\, T \mid T \parr T \\
e & ::= & \ldots & \mid f \cdot e \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ x_i \mapsto c_i}\} \mid \mathbf{Out}_i\ e \mid \mathbf{Match}\ \{[x,x] \mapsto c\} \mid \mathbf{Handle}_i\ e\ \mathbf{with}\ f \\
f & ::= & \ldots & \mid \lambda(\alpha \cdot x).c \mid f\ f \mid \mathbf{In}_i\ f \mid [f,f] \\
c & ::= & \ldots & \mid \mathbf{Case}\ e\ \{\alpha \cdot x \mapsto c\} \mid \mathbf{Case}\ f\ \{\overline{\mathbf{In}_i\ \alpha_i \mapsto c_i}\} \mid \mathbf{Case}\ f\ \{[\alpha,\alpha] \mapsto c\} \\
v & ::= & \ldots & \mid f \cdot v \mid \mathbf{Match}\ \{\overline{\mathbf{In}_i\ \alpha_i \mapsto c_i}\} \mid \mathbf{Match}\ \{[\alpha,\alpha] \mapsto c\} \\
\mathcal{E}[\,] & ::= & \ldots & \mid f \cdot \square
\end{array}
$$

Typing

$$\frac{c : (x : T_2, \Gamma \vdash \alpha : T_1, \Delta)}{\Gamma \mid \lambda(\alpha \cdot x).c : T_1 \prec T_2 \vdash \Delta}\ (\text{L-}\prec\text{-Intro}) \qquad \frac{\Gamma \mid f : T_1 \vdash \Delta \quad \Gamma \vdash e : T_2 \mid \Delta}{\Gamma \vdash f \cdot e : T_1 \prec T_2 \mid \Delta}\ (\text{R-}\prec\text{-Intro})$$

$$\frac{\Gamma \mid f_1 : T_1 \prec T_2 \vdash \Delta \quad \Gamma \mid f_2 : T_1 \vdash \Delta}{\Gamma \mid f_1\ f_2 : T_2 \vdash \Delta}\ (\text{L-}\prec\text{-Elim}) \qquad \frac{\Gamma \vdash e : T_1 \prec T_2 \mid \Delta \quad c : (x : T_1, \Gamma \vdash \alpha : T_2, \Delta)}{\mathbf{Case}\ e\ \{\alpha \cdot x \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{R-}\prec\text{-Elim})$$

$$\frac{\Gamma \mid f : T_i \vdash \Delta}{\Gamma \mid \mathbf{In}_i\ f : T_1 \,\&\, T_2 \vdash \Delta}\ (\text{L-}\&\text{-Intro}_i) \qquad \frac{\forall i, c_i : (\Gamma \vdash \alpha_i : T_i, \Delta)}{\Gamma \vdash \mathbf{Match}\ \{\overline{\mathbf{In}_i\ \alpha_i \mapsto c_i}\} : T_1 \,\&\, T_2 \mid \Delta}\ (\text{R-}\&\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \,\&\, T_2 \vdash \Delta \quad \forall i, c_i : (\Gamma \vdash \alpha_i : T_i, \Delta)}{\mathbf{Case}\ f\ \{\overline{\mathbf{In}_i\ \alpha_i \mapsto c_i}\} : (\Gamma \vdash \Delta)}\ (\text{L-}\&\text{-Elim}) \qquad \frac{\Gamma \vdash e : T_1 \,\&\, T_2 \mid \Delta}{\Gamma \vdash \mathbf{Out}_i\ e : T_i \mid \Delta}\ (\text{R-}\&\text{-Elim}_i)$$

$$\frac{\Gamma \mid f_1 : T_1 \vdash \Delta \quad \Gamma \mid f_2 : T_2 \vdash \Delta}{\Gamma \mid [f_1, f_2] : T_1 \parr T_2 \vdash \Delta}\ (\text{L-}\parr\text{-Intro}) \qquad \frac{c : (\Gamma \vdash \alpha : T_1, \beta : T_2, \Delta)}{\Gamma \vdash \mathbf{Match}\ \{[\alpha,\beta] \mapsto c\} : T_1 \parr T_2 \mid \Delta}\ (\text{R-}\parr\text{-Intro})$$

$$\frac{\Gamma \mid f : T_1 \parr T_2 \vdash \Delta \quad c : (\Gamma \vdash \alpha : T_1, \beta : T_2, \Delta)}{\mathbf{Case}\ f\ \{[\alpha,\beta] \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{L-}\parr\text{-Elim}) \qquad \frac{\Gamma \vdash e : T_1 \parr T_2 \mid \Delta \quad \Gamma \mid f : T_i \vdash \Delta}{\Gamma \vdash \mathbf{Handle}_i\ e\ \mathbf{with}\ f : T_{2-i+1} \mid \Delta}\ (\text{R-}\parr\text{-Elim}_i)$$

Reduction

$$
\begin{array}{lcl}
\langle f \cdot v \parallel \lambda(\alpha \cdot x).c \rangle & \triangleright_\beta & c\{x := v, \alpha := f\} \\
\langle \mathbf{Match}\ \{\overline{\mathbf{In}_i\ \alpha_i \mapsto c_i}\} \parallel \mathbf{In}_j\ f \rangle & \triangleright_\beta & c_j\{\alpha_j := f\} \\
\langle \mathbf{Match}\ \{[\alpha,\beta] \mapsto c\} \parallel [f_1, f_2] \rangle & \triangleright_\beta & c\{\alpha := f_1, \alpha := f_2\}
\end{array}
$$

Fig. 5. Dual rules for $\prec$, $\&$, and $\parr$

as a consumer of type $(Y \prec Z) \prec (X \prec Y) \prec X \prec Z$. Similarly, a swap function of type $(X \otimes Y) \to (Y \otimes X)$ can also be used as a cofunction of type $(X \parr Y) \prec (Y \parr X)$. Both are equally useful. Every program can be used in two ways - an exciting avenue that we intend to explore more in future work.

In the design above, e° is not yet completely identical to e. In an earlier design, we phrased the calculus in such a way that the dualization operation on expressions is indeed the identity function (by having statements where the producer/consumer side switch depending on whether one abstracts over a μ or a $\tilde{\mu}$), but this made the presentation more complicated in other ways. But even without this, code could be reused in the form of "macro" transformations or with a dedicated language construct (say, a $\mathit{dual}(e)$ construct) where the interpreter takes care of switching the sides when necessary. We leave the elaboration of these ideas to future work.

To summarize, the proof/refutation duality allows us to mechanically derive the syntax, typing and reduction rules for the respective dual connective, and it opens a path towards the reuse of a term as both a producer of a type and a consumer of its dual type.

6 EXTENSIONS

In this section, we consider some standard extensions of the calculi in the new light of having all four rules and bi-expressibility.

The first extension we consider are the logical constants, which serve as units for products and sums. Since we have two differently polarized products and two differently polarized sums, we consequently need 4 different units whose rules are given in Figure 6. Following the standard notation from linear logic, the unit for $\otimes$ is written 1, the unit for $\oplus$ is 0, the unit for $\&$ is $\top$, and the unit for $\parr$ is $\bot$. 1 is dual to $\bot$ and 0 is dual to $\top$, hence both the syntax and the typing rules of the respective dual connective follows mechanically like described in the previous section.

One of the exciting insights of linear logic is that the function type $\to$ can be decomposed into a combination of a negative sum $\parr$ and negation type. The decomposition of both negative functions $\to$ and positive cofunctions $\prec$ requires two different kinds of negations; negative negations $\neg$ and positive negations $\sim$. The rules for both kinds of negations, which are dual to each other, are given in Figure 7. With negation in place, we can very directly see that the type $T_1 \to T_2$ is isomorphic to $\neg\, T_1 \parr T_2$:

$$
\begin{array}{lcl}
\lambda(x \cdot \beta).c & = & \textbf{Match}\ \{[\alpha, \beta] \mapsto \textbf{Case}\ \alpha\ \{\textbf{Not}\ x \mapsto c\}\} \\
e_1\ e_2 & = & \textbf{Handle}_1\ e_1\ \textbf{with}\ (\textbf{Not}\ e_2) \\
e \cdot f & = & [\textbf{Not}\ e, f] \\
\textbf{Case}\ f\ \{x \cdot \beta \mapsto c\} & = & \textbf{Case}\ f\ \{[\alpha, \beta] \mapsto \textbf{Case}\ \alpha\ \{\textbf{Not}\ x \mapsto c\}\}
\end{array}
$$

And, by duality, the same holds for $T_1 \prec T_2$ and $\sim T_1 \otimes T_2$. It is also not hard to see that $\textbf{Not}\ e : \neg\, T$ is isomorphic to $e \cdot \textbf{triv} : T \to \bot$ and $\textbf{Not}\ f : \sim T$ is isomorphic to $f \cdot \textbf{triv} : T \prec 0$.

With regard to universal and existential types, whose rules are given in Figure 8, it is pleasing to see that the left elimination rule for universal types is basically the usual right elimination rule for "opening an existential package", and the left elimination rule of existential types corresponds to the usual type application right elimination rule for universal types. Of course, $\forall$ is dual to $\exists$, and the duality is again reflected directly in the typing rules. Bi-expressibility follows the same structure as bi-expressibility for functions.

Syntax

$$
\begin{array}{lcl}
T & ::= & \ldots \mid 1 \mid 0 \mid \top \mid \bot \\
e & ::= & \ldots \mid \mathbf{triv} \mid \mathbf{Case}\ e\ \{\} \mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} \mid \mathbf{Match}\ \{\} \\
f & ::= & \ldots \mid \mathbf{triv} \mid \mathbf{Case}\ f\ \{\} \mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} \mid \mathbf{Match}\ \{\} \\
c & ::= & \ldots \mid \mathbf{UnTriv}\ f \mid \mathbf{Case}\ e\ \{\mathbf{triv} \mapsto c\} \mid \mathbf{UnTriv}\ e \mid \mathbf{Case}\ f\ \{\mathbf{triv} \mapsto c\} \\
v & ::= & \ldots \mid \mathbf{triv} \mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\}
\end{array}
$$

Typing

$$\frac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} : 1 \vdash \Delta}\ (\text{L-1-Intro}) \qquad \frac{}{\Gamma \vdash \mathbf{triv} : 1 \mid \Delta}\ (\text{R-1-Intro})$$

$$\frac{\Gamma \mid f : 1 \vdash \Delta}{\mathbf{UnTriv}\ f : (\Gamma \vdash \Delta)}\ (\text{L-1-Elim}) \qquad \frac{\Gamma \vdash e : 1 \mid \Delta \quad c : (\Gamma \vdash \Delta)}{\mathbf{Case}\ e\ \{\mathbf{triv} \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{R-1-Elim})$$

$$\frac{}{\Gamma \mid \mathbf{Match}\ \{\} : 0 \vdash \Delta}\ (\text{L-0-Intro}) \qquad \textit{no right introduction rule for } 0$$

no left elimination rule for 0 $\qquad \dfrac{\Gamma \vdash e : 0 \mid \Delta}{\mathbf{Case}\ e\ \{\} : (\Gamma \vdash \Delta)}\ (\text{R-0-Elim})$

no left introduction rule for $\top$ $\qquad \dfrac{}{\Gamma \vdash \mathbf{Match}\ \{\} : \top \mid \Delta}\ (\text{R-}\top\text{-Intro})$

$$\frac{\Gamma \mid f : \top \vdash \Delta}{\mathbf{Case}\ f\ \{\} : (\Gamma \vdash \Delta)}\ (\text{L-}\top\text{-Elim}) \qquad \textit{no right elimination rule for } \top$$

$$\frac{}{\Gamma \mid \mathbf{triv} : \bot \vdash \Delta}\ (\text{L-}\bot\text{-Intro}) \qquad \frac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} : \bot \mid \Delta}\ (\text{R-}\bot\text{-Intro})$$

$$\frac{\Gamma \mid f : \bot \vdash \Delta \quad c : (\Gamma \vdash \Delta)}{\mathbf{Case}\ f\ \{\mathbf{triv} \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{L-}\bot\text{-Elim}) \qquad \frac{\Gamma \vdash e : \bot \mid \Delta}{\mathbf{UnTriv}\ e : (\Gamma \vdash \Delta)}\ (\text{R-}\bot\text{-Elim})$$

Reduction (*no rules for* $\top$ *and* 0)

$$\langle \mathbf{triv} \mid\mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} \rangle \triangleright_\beta c \qquad \langle \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} \mid\mid \mathbf{triv} \rangle \triangleright_\beta c$$

Bi-Expressibility (*rules for* $\top$ *and* $\bot$ *are analogous to those for* 1 *and* 0)

$$\frac{c : (\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.\mathbf{Case}\ x\ \{\mathbf{triv} \mapsto c\} : 1 \vdash \Delta}\ (\text{L-1-Intro}) \qquad \frac{}{\Gamma \vdash \mu\alpha.\mathbf{UnTriv}\ \alpha : 1 \mid \Delta}\ (\text{R-1-Intro})$$

$$\frac{\Gamma \mid f : 1 \vdash \Delta}{\langle \mathbf{triv} \mid\mid f \rangle : (\Gamma \vdash \Delta)}\ (\text{L-1-Elim}) \qquad \frac{\Gamma \vdash e : 1 \mid \Delta \quad c : (\Gamma \vdash \Delta)}{\langle e \mid\mid \mathbf{Match}\ \{\mathbf{triv} \mapsto c\} \rangle : (\Gamma \vdash \Delta)}\ (\text{R-1-Elim})$$

$$\frac{}{\Gamma \mid \tilde{\mu}x.\mathbf{Case}\ x\ \{\} : 0 \vdash \Delta}\ (\text{L-0-Intro}) \qquad \textit{no right introduction rule for } 0$$

no left elimination rule for 0 $\qquad \dfrac{\Gamma \vdash e : 0 \mid \Delta}{\langle e \mid\mid \mathbf{Match}\ \{\} \rangle : (\Gamma \vdash \Delta)}\ (\text{R-0-Elim})$

Fig. 6. Positive units 1, 0 and negative units $\top$, $\bot$

Syntax

$$
\begin{array}{lll}
T & ::= & \ldots \mid \neg T \mid {\sim} T \\
e & ::= & \ldots \mid \mathbf{Not}\ f \mid \mathbf{Throw}\ f\ f \mid \mathbf{Match}\ \{\mathbf{Not}\ x \mapsto c\} \mid \mathbf{Case}\ e\ \{\mathbf{Not}\ \alpha \mapsto c\} \\
f & ::= & \ldots \mid \mathbf{Not}\ e \mid \mathbf{Throw}\ e\ e \mid \mathbf{Match}\ \{\mathbf{Not}\ \alpha \mapsto c\} \mid \mathbf{Case}\ f\ \{\mathbf{Not}\ x \mapsto c\} \\
v & ::= & \ldots \mid \mathbf{Not}\ f \\
\mathcal{F}[\,] & ::= & \ldots \mid \mathbf{Not}\ \square
\end{array}
$$

Typing

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \mid \mathbf{Match}\ \{\mathbf{Not}\ \alpha \mapsto c\} : {\sim} T \vdash \Delta}\ (\text{L-}{\sim}\text{-Intro}) \qquad \frac{\Gamma \mid f : T \vdash \Delta}{\Gamma \vdash \mathbf{Not}\ f : {\sim} T \mid \Delta}\ (\text{R-}{\sim}\text{-Intro})$$

$$\frac{\Gamma \mid f_1 : {\sim} T \vdash \Delta \quad \Gamma \mid f_2 : T \vdash \Delta}{\mathbf{Throw}\ f_1\ f_2 : (\Gamma \vdash \Delta)}\ (\text{L-}{\sim}\text{-Elim}) \qquad \frac{\Gamma \vdash e : {\sim} T \mid \Delta \quad c : (\Gamma \vdash \alpha : T, \Delta)}{\mathbf{Case}\ e\ \{\mathbf{Not}\ \alpha \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{R-}{\sim}\text{-Elim})$$

$$\frac{\Gamma \vdash e : T \mid \Delta}{\Gamma \mid \mathbf{Not}\ e : \neg T \vdash \Delta}\ (\text{L-}\neg\text{-Intro}) \qquad \frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \vdash \mathbf{Match}\ \{\mathbf{Not}\ x \mapsto c\} : \neg T \mid \Delta}\ (\text{R-}\neg\text{-Intro})$$

$$\frac{\Gamma \mid f : \neg T \vdash \Delta \quad c : (\Gamma, x : T \vdash \Delta)}{\mathbf{Case}\ f\ \{\mathbf{Not}\ x \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{L-}\neg\text{-Elim}) \qquad \frac{\Gamma \vdash e_1 : \neg T \mid \Delta \quad \Gamma \vdash e_2 : T \mid \Delta}{\mathbf{Throw}\ e_1\ e_2 : (\Gamma \vdash \Delta)}\ (\text{R-}\neg\text{-Elim})$$

Reduction

$$
\begin{array}{lll}
\langle \mathbf{Not}\ f \mid\mid \mathbf{Match}\ \{\mathbf{Not}\ \alpha \mapsto c\} \rangle & \triangleright_\beta & c\{\alpha := f\} \\
\langle \mathbf{Match}\ \{\mathbf{Not}\ x \mapsto c\} \mid\mid \mathbf{Not}\ v \rangle & \triangleright_\beta & c\{x := v\}
\end{array}
$$

Bi-Expressibility

$$\frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \mid \mu x.\mathbf{Case}\ x\ \{\mathbf{Not}\ \alpha \mapsto c\} : {\sim} T \vdash \Delta}\ (\text{L-}{\sim}\text{-Intro}) \qquad \frac{\Gamma \mid f : T \vdash \Delta}{\Gamma \vdash \tilde{\mu}\alpha.\mathbf{Throw}\ \alpha\ f : {\sim} T \mid \Delta}\ (\text{R-}{\sim}\text{-Intro})$$

$$\frac{\Gamma \mid f_1 : {\sim} T \vdash \Delta \quad \Gamma \mid f_2 : T \vdash \Delta}{\langle \mathbf{Not}\ f_2 \mid\mid f_1 \rangle : (\Gamma \vdash \Delta)}\ (\text{L-}{\sim}\text{-Elim}) \qquad \frac{\Gamma \vdash e : {\sim} T \mid \Delta \quad c : (\Gamma \vdash \alpha : T, \Delta)}{\langle e \mid\mid \mathbf{Match}\ \{\mathbf{Not}\ \alpha \mapsto c\} \rangle : (\Gamma \vdash \Delta)}\ (\text{R-}{\sim}\text{-Elim})$$

$$\frac{\Gamma \vdash e : T \mid \Delta}{\Gamma \mid \mu x.\mathbf{Throw}\ x\ e : \neg T \vdash \Delta}\ (\text{L-}\neg\text{-Intro}) \qquad \frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \vdash \tilde{\mu}\alpha.\mathbf{Case}\ \alpha\ \{\mathbf{Not}\ x \mapsto c\} : \neg T \mid \Delta}\ (\text{R-}\neg\text{-Intro})$$

$$\frac{\Gamma \mid f : \neg T \vdash \Delta \quad c : (\Gamma, x : T \vdash \Delta)}{\langle \mathbf{Match}\ \{\mathbf{Not}\ x \mapsto c\} \mid\mid f \rangle : (\Gamma \vdash \Delta)}\ (\text{L-}\neg\text{-Elim}) \qquad \frac{\Gamma \vdash e_1 : \neg T \mid \Delta \quad \Gamma \vdash e_2 : T \mid \Delta}{\langle e_1 \mid\mid \mathbf{Not}\ e_2 \rangle : (\Gamma \vdash \Delta)}\ (\text{R-}\neg\text{-Elim})$$

Fig. 7. Extension with negation

Syntax

$$
\begin{array}{llll}
T & ::= & \ldots \mid \forall X.T \mid \exists X.T & \\
e & ::= & \ldots \mid \Lambda\{X,\alpha\}.c \mid e\ [T] \mid \{T, e\} & \qquad v ::= \ldots \mid \Lambda\{X,\alpha\}.c \mid \{T, v\} \\
f & ::= & \ldots \mid \Lambda\{X,x\}.c \mid f\ [T] \mid \{T, f\} & \qquad \mathcal{E}[\,] ::= \ldots \mid \{T, \Box\} \\
c & ::= & \ldots \mid \textbf{Case}\ f\ \{X,\alpha \mapsto c\} \mid \textbf{Case}\ e\ \{X,x \mapsto c\} &
\end{array}
$$

Typing

$$\frac{\Gamma \mid f : T_2\{X := T_1\} \vdash \Delta}{\Gamma \mid \{T_1, f\} : \forall X.T_2 \vdash \Delta}\ (\text{L-}\forall\text{-Intro}) \qquad \frac{c : X, \Gamma \vdash \alpha : T, \Delta \quad X \notin FV(\Gamma \cup \Delta)}{\Gamma \vdash \Lambda\{X,\alpha\}.c : \forall X.T \mid \Delta}\ (\text{R-}\forall\text{-Intro})$$

$$\frac{\Gamma \mid f : \forall X.T \vdash \Delta \quad c : (X, \Gamma \vdash \alpha : T, \Delta)}{\textbf{Case}\ f\ \{X,\alpha \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{L-}\forall\text{-Elim}) \qquad \frac{\Gamma \vdash e : \forall X.T \mid \Delta}{\Gamma \vdash e\ [T_1] : T\{X := T_1\} \mid \Delta}\ (\text{R-}\forall\text{-Elim})$$

$$\frac{c : (x : T, \Gamma \vdash X, \Delta) \quad X \notin FV(\Gamma \cup \Delta)}{\Gamma \mid \Lambda\{X,x\}.c : \exists X.T \vdash \Delta}\ (\text{L-}\exists\text{-Intro}) \qquad \frac{\Gamma \mid e : T_2\{X := T_1\} \vdash \Delta}{\Gamma \vdash \{T_1, e\} : \exists X.T_2 \mid \Delta}\ (\text{R-}\exists\text{-Intro})$$

$$\frac{\Gamma \mid f : \exists X.T \vdash \Delta}{\Gamma \mid f\ [T_1] : T\{X := T_1\} \vdash \Delta}\ (\text{L-}\exists\text{-Elim}) \qquad \frac{\Gamma \vdash e : \exists X.T \mid \Delta \quad c : (x : T, \Gamma \vdash X, \Delta)}{\textbf{Case}\ e\ \{X,x \mapsto c\} : (\Gamma \vdash \Delta)}\ (\text{R-}\exists\text{-Elim})$$

Reduction

$$\langle \Lambda\{X,\alpha\}.c \mid\mid \{T, f\}\rangle \triangleright_\beta c\{X := T, \alpha := f\} \qquad \langle \{T, v\} \mid\mid \Lambda\{X,x\}.c\rangle \triangleright_\beta c\{X := T, x := v\}$$

Bi-Expressibility (*Rules for* $\exists$ *are analogous to those for* $\forall$)

$$\frac{\Gamma \mid f : T_2\{X := T_1\} \vdash \Delta}{\Gamma \mid \tilde{\mu}x.\langle x\ [T_1] \mid\mid f\rangle : \forall X.T_2 \vdash \Delta}\ (\text{L-}\forall\text{-Intro}) \qquad \frac{c : X, \Gamma \vdash \alpha : T, \Delta \quad X \notin FV(\Gamma \cup \Delta)}{\Gamma \vdash \mu\beta.\textbf{Case}\ \beta\ \{X,\alpha \mapsto c\} : \forall X.T \mid \Delta}\ (\text{R-}\forall\text{-Intro})$$

$$\frac{\Gamma \mid f : \forall X.T_1 \vdash \Delta \quad c : (X, \Gamma \vdash \alpha : T_1, \Delta)}{\langle \Lambda\{X,\alpha\}.c \mid\mid f\rangle : (\Gamma \vdash \Delta)}\ (\text{L-}\forall\text{-Elim}) \qquad \frac{\Gamma \vdash e : \forall X.T \mid \Delta}{\Gamma \vdash \mu\alpha.\langle e \mid\mid \{T_1, \alpha\}\rangle : T\{X := T_1\} \mid \Delta}\ (\text{R-}\forall\text{-Elim})$$

Fig. 8. Extension with universal and existential types.

7 PROGRAMMING WITH ALL RULES

In this section, we present a few examples that illustrate the utility of polarized connectives and the flexibility of having all rules. We also revisit the filter example from Section 1.

7.1 Error Handling

The first example serves to illustrate both the value of having first-class consumers, and the naturality of programs using rules from all calculi. Consider the familiar problem of error-handling. Suppose we have three functions $f : A \longrightarrow B \vee E$, $g : B \longrightarrow C \vee E$ and $h : C \longrightarrow D \vee E$. These functions take an argument (of type A, B or C) and either return a result (of type B, C or D) or return an error E. The problem is how to compose these functions in such a way that $h \circ g \circ f$ is a function of type $A \longrightarrow D \vee E$.

Solving this problem crucially depends on how the type $\vee$ is represented. In our language, we have two choices: Positive ("data type", evaluated when constructed) disjunction $\oplus$ or negative ("codata type", evaluated when destructed) disjunction $\parr$. We will now discuss both alternatives in turn.[6]

Error Handling Using a Positive Type. When we choose to model $\vee$ as a positive type $\oplus$, we have constructors $\mathbf{In}_1$ and $\mathbf{In}_2$, and can destruct a term of type $A \oplus B$ by pattern matching on it. The composition of the three functions f, g and h can thus be written as follows:

$$
\begin{array}{l}
h \circ g \circ f : A \to D \oplus E \\
h \circ g \circ f = \ \lambda(x \cdot \alpha).\mathbf{Case}\ (f\ x)\ \{ \\
\qquad\qquad \mathbf{In}_1\ y \mapsto \mathbf{Case}\ (g\ y)\ \{ \\
\qquad\qquad\quad \mathbf{In}_1\ z \mapsto \langle h\ z \mid\mid \alpha \rangle \\
\qquad\qquad\quad \mathbf{In}_2\ e \mapsto \langle \mathbf{In}_2\ e \mid\mid \alpha \rangle \} \\
\qquad\qquad \mathbf{In}_2\ e \mapsto \langle \mathbf{In}_2\ e \mid\mid \alpha \rangle \}
\end{array}
$$

This style of error handling is familiar in conventional functional programming languages with algebraic types. The verbosity of this implementation can, of course, be greatly reduced by syntactic sugar or monadic do-notation. But what we really want to point out is how un-noteworthy the implementation is. This illustrates our point that we do not lose the naturality of natural deduction when writing programs using these calculi.

Error Handling Using a Negative Type. Implementing the same example using the negative type $\parr$ is less familiar:

$$
\begin{array}{l}
h \circ g \circ f : A \to D \parr E \\
h \circ g \circ f = \ \lambda(x \cdot \alpha). \\
\qquad\qquad \langle \mathbf{Match}\ \{[res, err] \mapsto \\
\qquad\qquad\quad \langle\ \mathbf{Handle}_2 \\
\qquad\qquad\qquad h\ (\mathbf{Handle}_2 \\
\qquad\qquad\qquad\quad g\ (\mathbf{Handle}_2\ f\ x\ \mathbf{with}\ err) \\
\qquad\qquad\qquad\quad \mathbf{with}\ err) \\
\qquad\qquad\qquad \mathbf{with}\ err) \mid\mid res\ \rangle\ \} \\
\qquad\qquad \mid\mid \alpha \rangle
\end{array}
$$

After introducing the variables x and α by a lambda abstraction, we encounter the $\parr$ introduction form $\mathbf{Match}\ \{[res, err] \mapsto \langle \ldots \mid\mid res \rangle\}$, bringing two continuations into scope: the continuation res which we can use to return a result of type D and the continuation err which we can use to return an error of type E. Since we want to write our program following the happy path, we return directly to the result continuation, so we have to fill the hole with a term of type D. If t is a term of type $A \parr E$, then the elimination form $\mathbf{Handle}_2\ t\ \mathbf{with}\ e$ returns the first option A along the implicit happy path, and the possibility of an error case of type E is handled by the continuation e explicitly in the handler.

Comparison. Programming with both polarities feels natural, leaving the choice of which construct to use up to the programmer. In conventional (functional) programming languages, exceptions are usually modelled using the positive type $\oplus$. Since these languages are based on natural deduction, there is no natural way to model exceptions using the negative type $\parr$, apart from rewriting the

[6] This example is inspired by Spiwack [2014], who discusses this example in the context of the $\overline{\lambda}\mu\tilde{\mu}$-calculus, where the sequent calculus syntax leads to terms that are less natural from a programmer's point of view.

program into continuation-passing or callback style. On the other hand, (checked) exceptions correspond more closely to the way we modelled exceptions with ⅋. The difference is that instead of representing the additional exception path in the types, languages using checked exceptions usually model this continuation using the `throws` keyword, and usually scope the exception handler dynamically instead of lexically.

7.2 *Parsimonious* Filter

Let us now return to the parsimonious filter example from Figure 1 in Section 1. How does that definition, based on equality between commands with nested (co)patterns, relate to the formal calculus that we have seen so far? To begin, consider just the left-hand sides of each definition in Figure 1 written by matching on the structures in a command (since the right-hand sides of each equality will be the same in each step, we will omit them for now):

$$\begin{aligned}
&\langle x && \mid\mid \alpha \circ \mathbf{Not}\ f && \rangle = \ldots \\
&\langle \mathit{filter}\ p\ xs && \mid\mid \mathit{start} && \rangle = \ldots \\
&\langle \mathit{filter/pass}\ p\ [] && \mid\mid [\mathit{diff}, \mathit{same}] && \rangle = \ldots \\
&\langle \mathit{filter/pass}\ p\ (x :: xs) && \mid\mid [\mathit{diff}, \mathit{same}] && \rangle = \ldots
\end{aligned}$$

The first step to desugaring this syntax is to replace each (left or right) elimination rule with the corresponding (left or right) introduction rule, according to the notion of bi-expressibility given here. For example, given a definition clause $\mathit{filter}\ p\ xs = \ldots$ familiar to functional programmers, we replace the application forms (corresponding to right function elimination) with call stacks (corresponding to left function introduction) around the starting continuation $p \cdot xs \cdot \mathit{start}$. Dually, the composition operator $\circ$ combining a (negated) function with a continuation defines a consumer instead of a producer. The infix application form $\alpha \circ (\mathbf{Not}\ f)$ can be rewritten in prefix notation as $(\circ)\ \alpha\ (\mathbf{Not}\ f)$ (corresponding to left subtraction elimination). This is replaced with a stack (corresponding to right subtraction introduction) around the starting value: $\alpha \cdot \mathbf{Not}\ f \cdot x$. Replacing each (left and right) eliminations by its equivalent introduction leads to these definition clauses:

$$\begin{aligned}
&\langle \alpha \cdot \mathbf{Not}\ f \cdot x && \mid\mid (\circ) && \rangle = \ldots \\
&\langle \mathit{filter} && \mid\mid p \cdot xs \cdot \mathit{start} && \rangle = \ldots \\
&\langle \mathit{filter/pass} && \mid\mid p \cdot [] && \cdot [\mathit{diff}, \mathit{same}] \rangle = \ldots \\
&\langle \mathit{filter/pass} && \mid\mid p \cdot (x :: xs) && \cdot [\mathit{diff}, \mathit{same}] \rangle = \ldots
\end{aligned}$$

The next step is to combine the (multi-)clause definitions on commands into the definition of a single consumer or producer which matches on its input or output, respectively. This step can be performed uniformly with a **Match** introduction written with nested (co)patterns like so:

$$(\circ) = \mathbf{Match}\ \{\ \alpha \cdot \mathbf{Not}\ f \cdot x \mapsto \ldots\ \}$$

$$\mathit{filter} = \mathbf{Match}\ \{\ p \cdot xs \cdot \mathit{start} \mapsto \ldots\ \}$$

$$\begin{aligned}
\mathit{filter/pass} = \mathbf{Match}\ \{\ & p \cdot [] && \cdot [\mathit{diff}, \mathit{same}] \mapsto \ldots \\
& p \cdot (x :: xs) && \cdot [\mathit{diff}, \mathit{same}] \mapsto \ldots\ \}
\end{aligned}$$

The final step to desugaring is to flatten out the **Match** with nested (co)patterns into their single-step counterparts for each individual type. Fundamentally, flattening combined patterns and

```
∘ : Z ≺ (¬(Y ⟶ Z) ≺ Y)
∘ = λ(α : Z · y : ¬(Y ⟶ Z) ≺ Y).
      Case y { β : ¬(Y ⟶ Z) · x : Y ↦
        Case β { Not (f : Y ⟶ Z) ↦ ⟨f x || α⟩ } }

filter : (X ⟶ Bool) ⟶ List X ⟶ List X
filter = λ(p : X ⟶ Bool · α : List X ⟶ List X).
           Case α { xs : List X · start : List X ↦
             ⟨Handle₂ (filter/pass p) xs with (start ∘ (Not (const xs)))
             || start⟩ }

filter/pass : (X ⟶ Bool) ⟶ List X ⟶ (List X ⅋ ⊤)
filter/pass =
  λ(p : X ⟶ Bool · α : List X ⟶ List X ⅋ ⊤).
    Case α { ys : List X · β : List X ⅋ ⊤ ↦
      Case ys {
        [] ↦ Case β { [diff : ListX, same : ⊤] ↦ ⟨Match {} || same⟩}
        (x : X :: xs : List X) ↦
          Case β { [diff : ListX, same : ⊤] ↦
            Case (p x) {
              True  ↦ ⟨filter/pass p xs || [diff ∘ (Not (x ::)), same]⟩
              False ↦ ⟨filter/pass p xs || [diff, diff ∘ (Not (const xs))]⟩ } } } }
```

Fig. 9. Parsimonious filter function reloaded

copatterns is not that different from the procedure of flattening ordinary nested patterns, involving tuples and sum types and other algebraic data types, typically done in conventional functional programming languages. The flattening of our parsimonious filter function looks like this:

$$(\circ) = \lambda(\alpha \cdot y).\ \textbf{Case}\ y\ \{\ \beta \cdot x \mapsto \textbf{Case}\ \beta\ \{\ \textbf{Not}\ f \mapsto \ldots\ \}\ \}$$

$$\mathit{filter} = \lambda(p \cdot \alpha).\ \textbf{Case}\ \alpha\ \{\ \mathit{xs} \cdot \mathit{start} \mapsto \ldots\ \}$$

$$\begin{aligned}\mathit{filter/pass} = \lambda(p \cdot \alpha).\ \textbf{Case}\ \alpha\ \{\ \mathit{ys} \cdot \beta \mapsto \textbf{Case}\ \mathit{ys}\ \{\ []\ &\mapsto \textbf{Case}\ \beta\ \{\ [\mathit{diff}, \mathit{same}] \mapsto \ldots\ \}\\ x :: \mathit{xs}\ &\mapsto \textbf{Case}\ \beta\ \{\ [\mathit{diff}, \mathit{same}] \mapsto \ldots\ \}\ \}\ \}\ \}\end{aligned}$$

The final, completely desugared and type-annotated version of Figure 1 into the core calculus syntax (with some trivial extensions, such as Booleans or recursion) is shown in Figure 9.

The desugared `filter/pass` function takes a predicate p on X as well as a list ys. Using ⅋, it returns a List X if at least one of the elements of the input do not fulfil p and a unit value otherwise. This function uses direct style to analyze the list and split into one of three cases:

- The list is empty. In this case, we send the unit **Match** {} to the continuation *same*, which signals that the list contains no filtered elements.

- The list is non-empty and its head satisfies p. In this case, we call filter recursively on the tail. If that call signals that nothing is filtered in the tail, then nothing is filtered in the whole list (*same*). If something is filtered in the tail, we start constructing the new tail with the curried constructor $(x ::)$ and send the whole tail (constructed from this call to $(x ::)$ and the result of the recursive call) to *diff*.
- The list is non-empty and its head does not satisfy p. In this case we keep the default continuation as-is and reset the shared tail to the current tail, since we cannot use the previous one which would have contained x. We never invoke *same* and implicitly discard it before the recursive call.

We can then use this function with the `filter` wrapper, which captures the current continuation *start* with a λ and uses $\mathbf{Handle}_2$ to discharge the "same" case with a continuation that passes the list to *start*.

The infix $\circ$ utility function is interesting in that it uses both the $\rightarrow$ and the $\neg$ connective. It demonstrates the utility of having λ and application forms on the consumer side, and of using negation to pass producers into a consumer context.

This example demonstrates the benefits of having all rules available. Compared to sequent calculus, all program parts can be written in direct style, using appropriate elimination forms. The usage and choice of polarized connectives leads to a more structured and readable program than low-level usages of `call/cc` and similar control operators.

In what sense is this implementation of `filter` "efficient"? Consider the expression $\langle \texttt{filter}(> 100)[0 \ldots 10^6] || \mathit{start} \rangle$. It will reduce to

$$\langle \texttt{filter/pass}(> 100)\ [] \ ||| [\mathit{start} \circ \mathit{Not}(101 ::) \circ \ldots \circ \mathbf{Not}\ (10^6 ::), \mathit{start} \circ \mathbf{Not}\ (\texttt{const}[101 \ldots 10^6]) \rangle$$

which can immediately return to the second continuation with $\langle () \ || \ \mathit{start} \circ \mathbf{Not}\ (\mathit{const}[101 \ldots 10^6]) \rangle$ and thus avoid unwinding the stack built up in the left-hand *diff* continuation. In this sense, the filter/pass function is "partially" tail-recursive: the recursive stack frame for the pass case can be optimized away in this program, but not the return pointer for when an element is removed. In a real implementation of our calculus, this would allow the compiler to use sharing on the $[101 \ldots 10^6]$ tail of the input list.

To summarize, in a real implementation of the language, the code in Figure 9 would combine these three properties:

(1) Tail-call optimization of recursive calls when the head is removed.
(2) Sharing the common list suffix rather than reallocating it.
(3) Immediately jumping to the starting caller when there is no more prefix to append to a modified tail.

8 RELATED AND FUTURE WORK

Computational Sequent Calculus. Our work is obviously related to previous sequent calculus-based languages, in particular the $\overline{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [2000] and the dual calculus of Wadler [2003]. Being directly inspired by the sequent calculus, each of these calculi define *distinct* syntactic categories for producers and consumers — for syntactically representing the sequent calculus' left and right rules logical rules — and do not feature elimination forms. The syntactic categories of $\overline{\lambda}\mu\tilde{\mu}$ with only function types are similar but not fully isomorphic; this is why Curien and Herbelin [2000] extend $\overline{\lambda}\mu\tilde{\mu}$ with a subtraction type, corresponding to the $\prec$ connective discussed here, which completes the duality with function types. In contrast, Wadler [2003]'s dual calculus eschews functions altogether, instead focusing on only conjunction, disjunction, and negation, presented in a non-polarized style (that is, the single conjunction type is produced by the

pair (x, y) and consumed by the projections $\mathrm{fst}[\alpha]$ and $\mathrm{snd}[\beta]$). These connectives are given two dual interpretations — one following a call-by-value semantics and one following call-by-name — which turns out to reveal the hidden polarities of the connectives: under call-by-value conjunction and disjunction correspond equationally to the positive $\otimes$ and $\oplus$ and under call-by-name they correspond to the negative $\&$ and $\parr$ discussed here, but not vice versa [Downen and Ariola 2014]. Our calculus also resembles the presentation of the calculus of *classical natural deduction* in Lovas and Crary [2006], based on work by Nanevski. That calculus contains products and sums, but does not differentiate between different polarizations. Instead, it uses the left introduction rules of the corresponding positive and right introduction rules of the corresponding negative connectives. They also don't have elimination rules in the core system but instead encode the right elimination rules in a manner similar to our diagonal encodings.

$\lambda\mu$ Calculus. The $\lambda\mu$ calculus [Parigot 1992b] is a natural deduction style language that corresponds to classical logic. We conjecture that it can be very straightforwardly embedded into our calculus with the following compositional transformation:

$$\begin{array}{lcll} [\![x]\!] & = & x & \\ [\![\lambda x.e]\!] & = & \lambda(x \cdot \alpha).\langle [\![e]\!] \mid\mid \alpha \rangle & \alpha \textit{ fresh} \\ [\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] & \\ [\![\mu\beta.([\alpha]\ e)]\!] & = & \mu\beta.\langle [\![e]\!] \mid\mid \alpha \rangle & \end{array}$$

Wadler [2005] describes a translation from $\lambda\mu$ to his aforementioned dual calculus (and back); due to the absence of elimination forms, the translation is considerably more complicated.

Translating Natural Deduction to Sequent Calculus. Gentzen [1935] describes a translation of derivations in the intuitionistic deduction system NJ into the intuitionistic sequent calculus LJ. In this translation, elimination rules are transformed into usages of the corresponding left rule for the connective and then an invocation of the cut rule, which means that normal forms are in general not translated to normal forms. Prawitz [1965, p.92] discusses a translation from natural deduction to sequent calculus that preserves normal forms, but at the expense of compositionality: The translation extends the sequent calculus derivation at its bottom when translating an introduction rule but from the top when an elimination rule is translated. Curien and Herbelin [2000] present two term-level translations $^{\mathcal{N}}$ and $^{>}$ that correspond to Prawitz' and Gentzen's proposals, respectively. These translations are similar to the elimination rule encodings of bi-expressibility, and in terms of continuation-passing style $^{>}$ is analogous to Hofmann and Streicher [1997]'s call-by-name CPS transformation whereas $^{\mathcal{N}}$ corresponds to Plotkin [1975]'s colon transformation.

Subtractive Logic. A dual to implication called subtraction, pseudo-difference, or co-implication is well-known in the domain of (bi-intuitionstic) logic [Rauszer 1974; Tranchini 2012], but finding an intuitive operational interpretation turned out to be difficult. Crolard [2004] proposed a rather complicated operational interpretation as "coroutines". We suggest that the reason for the complication is that Crolard considered an asymmetrical language with no consumer language. We think that our completely symmetric rules for $\rightarrow$ and $\prec$, with all rules, together with their simple operational semantics, is an improvement over these works.

Linear Logic, Polarity, Data and Codata Types. An important step in the development of the proof theory of sequent calculus was the discovery of linear logic by Girard [1987]. Linear logic restricts the applicability of structural rules, like weakening and contraction, which makes it possible to use a resource reading of typing judgements [Wadler 1990]: variables bound in the context are resources which are consumed in the construction of terms, which explains why they cannot be freely duplicated or discarded. A consequence of this resource interpretation is that the ordinary

connectives of classical or intuitionistic logic have to be split into multiple connectives; e.g. $\wedge$ has to be split into $\otimes$ and $\&$, $\vee$ has to be split into $\oplus$ and $\parr$. Studying proof search for linear logic, Andreoli [1992] realized that these connectives fall into two classes, which he called synchronous and asynchronous; later this terminology changed to positive and negative polarity. In programming language terms, polarity corresponds to the distinction between data types which are defined via their constructors, and codata types [Downen et al. 2019; Hagino 1989] which are defined via their observations/destructors. For example, while both $\oplus$ and $\parr$ are disjunctions, $\oplus$ corresponds to a data type defined with the help of two injections constructors, while $\parr$ corresponds to a codata type defined with one destructor bringing two continuations into scope. As argued by Zeilberger [2009] and many others, this distinction is important even in a system which does not enforce the linear use of variables because it determines evaluation order. We have illustrated this with examples where the availability of polarized connectives was critical.

When considering user-defined data and codata types, as we plan to do in future work, one has to specify generic mechanisms to construct and destruct terms of those types. Copattern matching [Abel et al. 2013] was introduced as a generic mechanism for constructing inhabitants of codata types, dually to how pattern matching is used to destruct inhabitants of data types. For example, Zeilberger [2008] provided a calculus where pattern matching and copattern matching, constructors and destructors are the only term-level constructs. In the context of the sequent calculus, Downen and Ariola [2021] provide a variant of the $\mu/\tilde{\mu}$ calculus with user provided data and codata types, but consider only the left and right introduction forms of the sequent calculus, and no elimination forms. The calculus presented in this paper could be similarly presented with user-defined data and codata type declarations, with introduction and elimination forms, left and right, derived from these declarations. In fact, a lot of the sets of rules presented in this paper were considered by us in this more general form first, but we defer a full development of that idea to future work.

One-Sided versus Two-Sided. The calculus we present here is a *two sided* sequent calculus, in the sense that we use both sides of the sequent separated by $\vdash$: producers live on the right-hand side and consumers live on the left. This distinction can quickly be summarized by the cut rule used in our core calculus:

$$\frac{\Gamma \vdash e : T \mid \Delta \qquad \Gamma \mid f : T \vdash \Delta}{\langle e \mid\mid f \rangle : (\Gamma \vdash \Delta)}$$

Importantly, this rule promises any producer e of a type T can interact with any consumer f of the *same type* T. But this isn't the only way to arrange interaction in a sequent calculus. A popular variant in the setting of classical linear logic [Girard 1987] is a *one sided* sequent calculus, which only ever uses a single side of the sequent throughout. This "restriction" can be made without loss of expressivity because of involutive negation in classical (linear) logic, for every proposition A there is a dual proposition $A^\perp$ such that $A^{\perp\perp} = A$, such that having A on the left of $\vdash$ is the same as having $A^\perp$ on the right. This involutive negation corresponds to the duality of types, here written as T°, which can be used to formulate a one-sided language. Focusing again on the iconic cut rule, we have two more possibilities for arranging a one-sided version of the calculus as

$$\frac{\vdash e : T \mid \Delta \qquad \vdash f : T^\circ \mid \Delta'}{\langle e \mid\mid f \rangle : (\vdash \Delta, \Delta')} \qquad\qquad \frac{\Gamma \vdash e : T \qquad \Gamma' \vdash f : T^\circ}{\langle e \mid\mid f \rangle : (\Gamma, \Gamma' \vdash)}$$

by putting *all* types on the right of $\vdash$ [Munch-Maccagnoni 2009] (as is popular in linear logic) or *all* variables to the left of $\vdash$ and the expression to the right [Spiwack 2014] (more popular in the programming languages community). The important thing to notice about these one-sided cut rules is the promise that any producer of a type T can interact with any *other producer* of the *opposite type* T°.

The idea of connecting producers to other producers fits with our idea of consumer/producer polymorphism in Section 5. Essentially, the point is that if the duality is complete enough, and if consumers of T are completely interchangeable with producers of T°, then two producers (or two consumers) of opposite types should be able to interact directly with one another. In such a setting there is no difference between T consumers and T° producers, so Munch-Maccagnoni [2009] eliminates the distinction between the commands $\langle e \mid\mid f \rangle$ and $\langle f \mid\mid e \rangle$. We conjecture that a complete story of consumer/producer polymorphism should elaborate on this one-sided view of sequents.

Left calculus. Right calculi are abounds in the literature behind both the fields of logic and programming languages. Intro calculi frequently appear in the study of proof theory, and are growing in number to help understand and implement programming languages, too. However, examples of left calculi can scarcely be found in the literature.

The idea of left elimination rules has been presented before by Carraro et al. [2012], which give a calculus similar to $\overline{\lambda}\mu\tilde{\mu}$ with a left rule for introducing function call stacks, but instead of ordinary λ-abstractions, it contains projections that access the argument and the return-pointer in a call stack. These projections are similar to the left elimination rule for functions presented here, by the analogy that a **Case** extracting the two components of a pair of two things is similar to projections from that pair to each component individually.

Independently, Nakazawa and Nagai [2014] introduces the same combination of left introduction and eliminations for functions in the context of the $\Lambda\mu$-calculus [Groote 1994], a variant of Parigot's $\lambda\mu$-calculus which collapses the distinction between commands and producers. This collapse improves the completeness properties of the rewriting theory [Saurin 2005] and elevates μ to a much more expressive *delimited control operator* [Herbelin and Ghilezan 2008], and the use of left elimination rules were key to reconciling extensionality (expressed by the λ-calculus' η law) of delimited control in $\Lambda\mu$ with standard properties like confluence.

Elimination calculus. We know of two calculi which correspond to what we call an elimination calculus. Negri's uniform calculus for classical linear logic (cf. Negri [2002] and Negri and Von Plato [2001, p. 213ff.]) is a termfree logical calculus with only elimination rules. Negri uses the term "general introduction rule" for what we call left elimination rules. Natural deduction and sequent calculus can be obtained by instantiating major and minor premises of these elimination rules with an instance of the axiom rule. This is similar to our bi-expressibility rules, but since she doesn't have to consider term assignment, she also doesn't have to deal with the activation and deactivation of formulas.

Parigot's free deduction [Parigot 1992a], a precursor to his $\lambda\mu$-calculus [Parigot 1992b], also contains only elimination rules, and he also obtains natural deduction and sequent calculus by instantiating major and minor premises by the axiom rule. In distinction to Negri, he also provides a term system with constructs which can be seen as precursors to both the μ and $\tilde{\mu}$-abstractions of Curien and Herbelin [2000].

Communication calculi and session types. There is a well-known relationship between linear logic and session types, both in its intuitionistic [Caires and Pfenning 2010] and its classical [Wadler 2014] version. These session type systems are based on some form of communication calculus, like the π-calculus, and provide a type system for the communication channels. There are two relationships between session type systems and our work. First, there is a relationship between the duality operation T° on types and a similar operation on session types: The dual of a session type for sending some data is a session type for receiving some data of that type, the dual of a session type for choosing between various options is a session type for offering those choices, and so on.

The second relationship concerns the non-local character of reduction in a communication calculus: The term which wants to send some information on a channel might be at some distance from the term for receiving information on that channel, but they nevertheless interact in a reduction step. This can also be observed in the elimination calculus, if we would formalize the reduction rules for that system directly. For example Parigot [1992a], who considers reduction rules for a system based on elimination rules, uses the following version of the axiom rule (which is derivable in our system)

$$\frac{}{\langle x \mid\mid x^{\perp} \rangle : (x : A \vdash x^{\perp} : A)} \textsc{Axiom}$$

A cut in his system then consists between an elimination rule which uses x as the major premise, and an elimination rule which uses $x^{\perp}$ as the major premise, but they don't have to stand directly next to each other. The resulting reduction system is closely reminiscent of a communication step happening between two ends of a channel. We plan to investigate both these aspects in more detail in future work.

Relations between rules. Our bi-expressibility principle concerns the "diagonal" relation between right-intro/left-elim and left-intro/right-elim. There are other sanity principles for rule pairs, but they concern the relation between introduction and elimination rules and are of particular interest in proof-theoretic semantics, such as *invertibility* and *harmony* [Schroeder-Heister 2018].

Transformations between consumers and producers. Our approach to consumer-producer polymorphism is related to but very different from whole-program transformations (generalizations of defunctionalization and refunctionalization) that transform data types into codata types or vice versa [Binder et al. 2019; Rendel et al. 2015]. Consumer-producer polymorphism allows us to use the same program in two ways, once as a consumer and once as a producer. When viewed as a macro code generator, it is a compositional transformation. The aforementioned generalizations of de- and refunctionalization, on the other hand, transform a whole program in a way that can be viewed as a matrix transposition [Ostermann and Jabs 2018], while preserving its operational behavior. For instance, when defunctionalizing a codata type into a data type, all copattern-matches on a destructor in the whole program are turned into a single pattern match. These transformations could also be applied to programs in the language presented here, but they are not in the scope of this paper.

9 CONCLUSIONS

Programming abstractions based on sequent calculus have, despite their attractive symmetry and expressive power, seen only limited influence on functional language design. We have opened up the design space of natural deduction and sequent calculus by considering all four kinds of rules and the four natural subcalculi. We have analyzed the interdependency between program structure and rule choice and have argued that offering all rules to the programmer maximizes expressiveness and allows a natural and modular program structure. We have proposed a constructive sanity check for the rules, bi-expressibility, and have shown how the dualities between the available connectives can be deepened in the form of a uniform syntax and consumer/producer polymorphism.

DATA AVAILABILITY STATEMENT

The Coq formalization of the main results of this paper is available online [Ostermann et al. 2022].

ACKNOWLEDGEMENTS

David Binder, Klaus Ostermann, Ingo Skupin and Tim Süberkrüb were supported by the DFG project "Efficient Compilation of Control-Effects", DFG grant OS 293/5-1. David Binder was also supported by the DFG project "Constructive Semantics and the Completeness Problem", DFG grant PI 1174/1-1.

A EXAMPLE PROGRAMS IN FORMAL CALCULUS SYNTAX

The examples in Section 2 were written in a simplified version of the formal syntax. The fully formal version of Table 1 can be found in Table 3, the fully formal version of Table 2 in Table 4.

Table 3. Four different ways to swap the components of $z : X \oplus Y$ and send to consumer $\alpha : Y \oplus X$.

Calculus	Program
Right	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle \mathbf{In}_2\ x \mid\mid \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle \mathbf{In}_1\ y \mid\mid \alpha\rangle\}$
Intro	$\langle z \mid\mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle \mathbf{In}_2\ x \mid\mid \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle \mathbf{In}_1\ y \mid\mid \alpha\rangle\}\rangle$
Left	$\langle z \mid\mid \mathbf{Match}\ \{\mathbf{In}_1\ x \mapsto \langle x \mid\mid \mathbf{Out}_2\ \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle y \mid\mid \mathbf{Out}_1\ \alpha\rangle\}\rangle$
Elim	$\mathbf{Case}\ z\ \{\mathbf{In}_1\ x \mapsto \langle x \mid\mid \mathbf{Out}_2\ \alpha\rangle; \mathbf{In}_2\ y \mapsto \langle y \mid\mid \mathbf{Out}_1\ \alpha\rangle\}$

Table 4. Computation from $x : (\top \mathbin{\&} X) \mathbin{\&} \top$ to $\alpha : \bot \oplus ((X \oplus \bot) \oplus \bot)$.

Calculus	Program	Program Structure
Right	$\langle \mathbf{In}_2\ (\mathbf{In}_1\ (\mathbf{In}_1\ (\mathbf{Out}_2\ (\mathbf{Out}_1\ x)))) \mid\mid \alpha\rangle$	α outside-in, x inside-out
Intro	$\langle x \mid\mid \mathbf{In}_1\ (\mathbf{In}_2\ \tilde{\mu}x.\langle \mathbf{In}_2\ (\mathbf{In}_1\ (\mathbf{In}_1\ x)) \mid\mid \alpha\rangle)\rangle$	x outside-in, α outside-in
Left	$\langle x \mid\mid \mathbf{In}_1\ (\mathbf{In}_2\ (\mathbf{Out}_1\ (\mathbf{Out}_1\ (\mathbf{Out}_2\ \alpha))))\rangle$	x outside-in, α inside-out
Elim	$\langle \mathbf{Out}_2\ (\mathbf{Out}_1\ x) \mid\mid \mathbf{Out}_1\ (\mathbf{Out}_1\ (\mathbf{Out}_2\ \alpha))\rangle$	α inside-out, x inside-out

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2480359.2429075

Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347. Issue 3. https://doi.org/10.1093/logcom/2.3.297

David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2019. Decomposition Diversity with Symmetric Data and Codata. *Proc. ACM Program. Lang.* 4, POPL, Article 30 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371098

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory* (Paris, France) *(CONCUR'10)*. Springer-Verlag, Berlin, Heidelberg, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Alberto Carraro, Thomas Ehrhard, and Antonino Salibra. 2012. The stack calculus. In *Proceedings Seventh Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2012, Rio de Janeiro, Brazil, September 29-30, 2012 (EPTCS, Vol. 113)*. 93–108. https://doi.org/10.48550/arXiv.1303.7331

Tristan Crolard. 2004. A Formulae-as-Types Interpretation of Subtractive Logic. *Journal of Logic and Computation* 14 (2004), 529–570. Issue 4. https://doi.org/10.1093/logcom/14.4.529

Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/357766.351262

Paul Downen and Zena M. Ariola. 2014. The Duality of Construction. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410 (ESOP '14)*. Springer-Verlag, Berlin, Heidelberg, 249–269. https://doi.org/10.1007/978-3-642-54833-8_14

Paul Downen and Zena M. Ariola. 2018. A tutorial on computational classical logic and the sequent calculus. *Journal of Functional Programming* 28 (2018). https://doi.org/10.1017/S0956796818000023

Paul Downen and Zena M. Ariola. 2021. Duality in Action. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–32. https://doi.org/10.4230/LIPIcs.FSCD.2021.1

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *European Symposium on Programming (ESOP '19)*. Springer, 119–146. https://doi.org/10.1007/978-3-030-17184-1_5

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs. An Introduction to Computing and Programming.* The MIT Press, Cambridge, Massachusetts London, England.

Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift* 39 (1935), 176–210 and 405–431.

Jeremy Gibbons. 2021. How to design co-programs. *Journal of Functional Programming* 31 (2021). https://doi.org/10.1017/S0956796821000113

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Philippe de Groote. 1994. On the Relation between the $\lambda\mu$-Calculus and the Syntactic Theory of Sequential Control. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning (LPAR '94)*. Springer-Verlag, Berlin, Heidelberg, 31–43. https://doi.org/10.1007/3-540-58216-9_27

Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629–650. https://doi.org/10.1016/S0747-7171(89)80065-3

Hugo Herbelin and Silvia Ghilezan. 2008. An Approach to Call-by-Name Delimited Continuations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/1328438.1328484

Martin Hofmann and Thomas Streicher. 1997. Continuation models are universal for $\lambda\mu$-calculus. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 – July 2, 1997*. IEEE Computer Society, 387–395. https://doi.org/10.1109/LICS.1997.614964

William Lovas and Karl Crary. 2006. Structural Normalization for Classical Natural Deduction. *Draft* (2006).

Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability. In *Computer Science Logic: 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL* (Coimbra, Portugal) *(CSL '09)*, Erich Grädel and Reinhard Kahle (Eds.). Springer, Berlin, Heidelberg, 409–423. https://doi.org/10.1007/978-3-642-04027-6_30

Koji Nakazawa and Tomoharu Nagai. 2014. Reduction System for Extensional Lambda-mu Calculus. In *Rewriting and Typed Lambda Calculi*, Gilles Dowek (Ed.). Springer International Publishing, Cham, 349–363. https://doi.org/10.1007/978-3-319-08918-8_24

Sara Negri. 2002. Varieties of Linear Calculi. *J. Philos. Log.* 31, 6 (2002), 569–590. https://doi.org/10.1023/A:1021264102972

Sara Negri and Jan Von Plato. 2001. *Structural Proof Theory.* Cambridge University Press. https://doi.org/10.1017/CBO9780511527340

Klaus Ostermann, David Binder, Ingo Skupin, Tim Süberkrüb, and Paul Downen. 2022. *Introduction and Elimination, Left and Right - Coq Formalization.* https://doi.org/10.5281/zenodo.6685674

Klaus Ostermann and Julian Jabs. 2018. Dualizing Generalized Algebraic Data Types by Matrix Transposition. In *European Symposium on Programming*. Springer, 60–85. https://doi.org/10.1007/978-3-319-89884-1_3

Michel Parigot. 1992a. Free deduction: An analysis of "Computations" in classical logic. In *Logic Programming*, A. Voronkov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 361–380. https://doi.org/10.1007/3-540-55460-2_27

Michel Parigot. 1992b. $\lambda\mu$-Calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, Andrei Voronkov (Ed.). Springer, Berlin, Heidelberg, 190–201.

Gordon D. Plotkin. 1975. Call-By-Name, Call-By-Value and the λ-Calculus. *Theoretical Computer Science* 1 (1975), 125–159. Issue 2. https://doi.org/10.1016/0304-3975(75)90017-1

Dag Prawitz. 1965. *Natural Deduction: A Proof-Theoretical Study.* Almqvist & Wiksell, Stockholm. Reprinted Mineola NY: Dover Publications (2006).

Cecylia Rauszer. 1974. A formalization of the propositional calculus of H-B logic. *Studia Logica* 33 (1974), 23–34.

Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/2784731.2784763

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference* (Boston). Association for Computing Machinery, New York, 717–740. https://doi.org/10.1145/800194.805852

Alexis Saurin. 2005. Separation with Streams in the $\lambda\mu$-calculus. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS '05)*. IEEE Computer Society, USA, 356–365. https://doi.org/10.1109/LICS.2005.48

Peter Schroeder-Heister. 2018. Proof-Theoretic Semantics. In *The Stanford Encyclopedia of Philosophy* (Spring 2018 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University.

Olin Shivers and David Fisher. 2004. Multi-Return Function Call. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 79–89. https://doi.org/10.1145/1016848.1016864

Arnaud Spiwack. 2014. A Dissection of L. (2014). Unpublished draft.

Luca Tranchini. 2012. Natural Deduction for Dual-intuitionistic Logic. *Studia Logica* 100, 3 (2012), 631–648. https://doi.org/10.1007/s11225-012-9417-8

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North-Holland.

Philip Wadler. 2003. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. ACM, New York, NY, USA, 189–201. https://doi.org/10.1145/944705.944723

Philip Wadler. 2005. Call-by-Value Is Dual to Call-by-Name - Reloaded. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 185–203. https://doi.org/10.1007/978-3-540-32033-3_15

Philip Wadler. 2014. Propositions as sessions. *J. Funct. Program.* 24, 2-3 (2014), 384–418. https://doi.org/10.1145/2398856.2364568

Noam Zeilberger. 2008. On the unity of duality. *Annals of Pure and Applied Logic* 153, 1-3 (2008), 66–96. https://doi.org/10.1016/j.apal.2008.01.001

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph. D. Dissertation. Carnegie Mellon University, USA. Advisor(s) Pfenning, Frank and Lee, Peter.