The Duality of λ-Abstraction

VIKRAMAN CHOUDHURY, Università di Bologna, Italy and Inria Sophia Antipolis, France
SIMON J. GAY, University of Glasgow, UK

In this paper, we develop and study the following perspective – just as higher-order functions give exponentials, higher-order continuations give coexponentials. From this, we design a language that combines exponentials and coexponentials, producing a duality of lambda abstraction.

We formalise this language by giving an extension of a call-by-value simply-typed lambda-calculus with covalues, coabstraction, and coapplication. We develop the semantics of this language using the axiomatic structure of continuations, which we use to produce an equational theory, that gives a complete axiomatisation of control effects. We give a computational interpretation to this language using speculative execution and backtracking, and use this to derive the classical control operators and computational interpretation of classical logic, and encode common patterns of control flow using continuations. By dualising functional completeness, we further develop duals of first-order arrow languages using coexponentials. Finally, we discuss the implementation of this duality as control operators in programming, and develop some applications.

CCS Concepts: • **Theory of computation** → **Type theory**; **Logic and verification**; **Categorical semantics**; **Denotational semantics**; **Control primitives**; **Functional constructs**; • **Software and its engineering** → **Syntax**; **Semantics**.

Additional Key Words and Phrases: type theory, category theory, duality, continuations, control effects, control operators, classical logic, lambda-calculus, curry-howard, denotational semantics, equational theory

ACM Reference Format:
Vikraman Choudhury and Simon J. Gay. 2025. The Duality of λ-Abstraction. *Proc. ACM Program. Lang.* 9, POPL, Article 12 (January 2025), 30 pages. https://doi.org/10.1145/3704848

1 Introduction

There are several well-known dualities of computation: (1) values and continuations [Filinski 1989; Parigot 1992], (2) call-by-value and call-by-name [Selinger 2001; Wadler 2003], (3) programs and evaluation contexts [Curien and Herbelin 2000], (4) producers and consumers [Girard 1991], (5) clients and servers in session types [Honda 1993], (6) strict and lazy evaluation [Ong 1988; Abramsky and Ong 1993], (7) product and sum types [Burstall 1980; Burstall et al. 1980], (8) effects (monads) and coeffects (comonads) [Petricek et al. 2014].

This paper presents and develops a different perspective: a duality of abstraction – of currying and cocurrying. Abstraction is at the heart of functional programming – it gives us higher-order functions that we build by lambda abstraction, and we apply them to arguments using function application. This is well understood using the currying/uncurrying isomorphism:

$$(C \times A) \to B \quad \cong \quad C \to (A \Rightarrow B) \tag{1}$$

Authors' Contact Information: Vikraman Choudhury, vikraman.choudhury@unibo.it, Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna, Bologna, Italy and Inria Sophia Antipolis, Valbonne, France; Simon J. Gay, simon.gay@glasgow.ac.uk, School of Computing Science, University of Glasgow, Glasgow, UK.

ACM 2475-1421/2025/1-ART12
https://doi.org/10.1145/3704848

The forwards direction is currying, which gives lambda abstraction. In an environment C with a free variable of type A, if we can produce a value of type B, we can lambda-abstract and get a function $A \Rightarrow B$ in the environment C. The function type $A \Rightarrow B$ is an exponential object, which comes with a (universal) evaluation map $\mathrm{eval}_{A,B} \colon (A \Rightarrow B) \times A \to B$ given by uncurrying, allowing us to apply a function to an argument.

Products can be dualised to coproducts (sums) – can we dualise currying? Formally, this is a matter of reversing the arrows, turning the products into coproducts, and turning the function type $\Rightarrow$ into a $\Leftarrow$ type:

$$(A \Leftarrow B) \to C \quad \cong \quad B \to (C + A) \tag{2}$$

Continuing the analogy with currying, the dual type $A \Leftarrow B$ (or coexponential object) should come with a (universal) coevaluation map $\mathrm{coeval}_{A,B} \colon B \to A + (A \Leftarrow B)$. Programming languages have both products and sums, why should they not enjoy both $\Rightarrow$ and $\Leftarrow$?

Loch Ness Mystery. For good reasons, this mysterious $A \Leftarrow B$ type is not found in conventional programming languages. The categorically-minded reader will recognise these two natural isomorphisms as coming from the adjunctions of cartesian closure, and cocartesian coclosure:

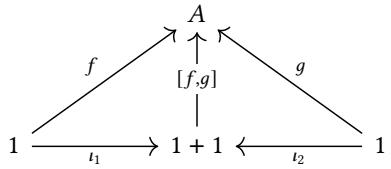
$$(-) \times A \dashv (-)^A \qquad\qquad {}^A(-) \dashv A + (-) \tag{3}$$

Exponential objects $(-)^A$ give right adjoints to product functors $(-) \times A$, and coexponential objects ${}^A(-)$ give left adjoints to coproduct functors $A + (-)$. If $\mathcal{C}$ is a cartesian closed category (a model for the simply-typed lambda-calculus), then by formal duality, $\mathcal{C}^{\mathrm{op}}$ becomes a cocartesian coclosed category. But, combining cartesian closure and cocartesian coclosure in the same category leads to a degeneracy – this is well-known as Joyal's lemma, and is explained in various forms by several authors in the literature, see for example, Lambek and Scott 1988, p.67; Girard 2011, § 7.A.4; Crolard 2001, Theorem 1.14; Abramsky 2012, § 2.1; Eades III and Bellin 2017, Theorem 20.

If $(-) \times A$ has a right adjoint, it must preserve the initial object, and if $A + (-)$ has a left adjoint, it must preserve the terminal object, giving these isomorphisms:

$$0 \times A \cong 0 \qquad A + 1 \cong 1 \tag{4}$$

By a Curry-Howard reading, the first isomorphism is the tautology $\bot \wedge A \leftrightarrow \bot$, and the second isomorphism is $\top \vee A \leftrightarrow \top$, which are well-known in their logical interpretation. However, seen as a programming language, this means the booleans would have no computational content, since $\mathrm{Bool} \cong 1 + 1 \cong 1$, leading to a degenerate language.



Since $1 + 1 \cong 1$, it is a terminal object, making $\iota_1, \iota_2 : 1 \to 1 + 1$ equal. If $f, g : 1 \to A$ are any two closed programs, then $f = [f, g] \circ \iota_1 = [f, g] \circ \iota_2 = g$. This makes the language degenerate – all closed programs of the same type are equal! This remark of Girard from *The Blind Spot* [2011, § 7.A.5, page 155] is worth quoting:

> **Digression: Loch Ness categories.** A certain number of "solutions" to the degeneracy (inconsistency at layer -2) circulate. All those I have seen being faulty, I will not indulge in a teratology, especially since some people devote an incredible amount of energy in the production of new erroneous solutions. A few remarks:

- If there is a category-theoretic solution, one is liable to provide a legible category. And not to formulate the adjunction rules – say – of a professed «subtraction» – the typical connective of the category-theoretic *bricoleurs* – supposedly acting like implication, but on the left. Hence, one must provide a *concrete* category, or at least a translation into a system already having a non-degenerate category-theoretic interpretation. What the experts in «subtraction» carefully avoid doing…with good reasons.

This degeneracy is often used to motivate linear logic, weakening the strict universal properties of limits and colimits, or that "we must separate the two worlds" [Eades III and Bellin 2017], leading to mixed linear-non-linear logics. These arguments are also important to the foundations of quantum theory [Abramsky 2012], which require no cloning and duplication, fitting nicely with linear logic. In this work, we refute this conventional wisdom – we produce a programming language with a computational interpretation, that has both currying and cocurrying! Of course, we cannot avoid mathematical degeneracies, so the trick works by playing on the meaning of the term "programming language".

Continuations and Classical Logic. Continuations are fundamental to many of the dualities of computation – they are dual to values (as in Parigot [1992]'s $\lambda\mu$), and they are fundamental to the duality of call-by-value and call-by-name (Selinger 2001; Wadler 2003), and the duality of programs and evaluation contexts (Curien and Herbelin [2000]'s $\mu\tilde{\mu}$). Continuations give a computational interpretation to classical logic, as discovered by Griffin [1989]. The logical nature of the isomorphism in Eq. (4) suggests that we should think about them using continuations. The duality in this paper also exploits continuations!

The ambitious reader might want to stop at this point, and try to implement the $\Leftarrow$ type and the isomorphism in Eq. (2), using their favorite control operators. This is the main insight on which this paper builds. The paper is written in such a way that different sections can be read independently, depending on the reader's inclination. Semantically-minded readers might want to skip ahead to § 3 to see what the trick is about. Programming-language enthusiasts might want to skip to § 6 to see how to program with the duality.

Outline and Contributions. This work is inspired by Filinski [1989]'s symmetric λ-calculus, and various dual calculi for values and continuations [Parigot 1992; Curien and Herbelin 2000]. The semantics uses the well-understood semantics of continuations, developed in Hofmann [1995], Thielecke [1997], Streicher and Reus [1998], and Hofmann and Streicher [2002], and in particular, Selinger [2001]. Compared to other dual calculi, we only restrict ourselves to a call-by-value language. The duality is a semantic one – of cartesian closure and cocartesian coclosure – which produces a syntactic duality of λ and $\tilde{\lambda}$.

- We present a $\lambda\tilde{\lambda}$ calculus, which exhibits two dual abstraction mechanisms: λ and $\tilde{\lambda}$. They bind values and covalues, respectively, and we call them functions and cofunctions, respectively, which exhibit currying and cocurrying. Functions have a function type, and cofunctions have (surprisingly) a sum type – the interaction of usual sums and cofunctions allows values and covalues to interact. We give a formal presentation of this language in § 2.
- We develop the semantics of $\lambda\tilde{\lambda}$ in two different ways, in § 3. First, we use continuations for covalues, and give a CPS semantics in § 3.1, essentially by pulling it out of a hat. Second, we perform a micrological study of continuations in § 3.2, to understand their axiomatic categorical structure, and how it produces exponentials and coexponentials. We interpret our language using this categorical semantics in § 3.3, and show that it matches the CPS semantics.
- Using our denotational semantics, we develop an equational theory for our language, in § 4. The equational theory is designed in stages, first giving the axiomatic equations for currying and

cocurrying, and then adding equations for control effects, which are validated by our semantics. We discuss the soundness, completeness, and axiomatics of these equations.
- Just as λ calculi can be split into first-order fragments, we split $\tilde{\lambda}$ into first-order arrow calculi, by dualising functional completeness, in § 5. These languages are understood operationally using continuations as handlers.
- We demonstrate the programming features of our language in § 6, by writing programs in a hypothetical language with functions and cofunctions. We show how our dual calculus can be easily implemented, and retrofit into existing programming languages, by turning $\tilde{\lambda}$ into a control operator, and implementing it in SML and Haskell.

We include our implementation and formalization as supplementary material [Choudhury 2024]. Some details are skipped in the main text, and included in the supplementary appendices, in the extended version of this paper.

2 Typing

We present a formal calculus called $\lambda\tilde{\lambda}$ which exhibits abstraction and coabstraction. The syntax and typing of $\lambda\tilde{\lambda}$ is presented in Fig. 1. It is a simply-typed lambda calculus with products, functions, and sums, extended with covalue types, coabstraction, and coapplication.

In Fig. 1a, we have the usual type constructors for unit, products, coproducts, and function types. Additionally, we have a dual type constructor $\tilde{A}$, which is the type of covalues. Expressions in our language are the usual ones, but additionally we have colambdas and coapplications, which are indicated by a tilde (that can be simply read as "bar", following French tradition) over the lambda and application symbols. Lambda and colambda are binding forms – lambdas can bind variables of any type, but colambdas only bind variables of dual types. Similar to application, coapplication coapplies the second argument to the first. Values are a subset of expressions, and substitution is restricted to values.

Raw terms are meaningless, and the meaningful terms are the well-typed ones deduced by the typing judgement, generated by the typing rules in Fig. 1c. To empahsize the symmetry in the rules, we note the polarity of the connectives in the presentation – positive/negative in the sense of [Girard 2001], or leftist/rightist in the sense of [Levy 2006, 2021], respectively, which means they have *reversible* derivations for introduction rules. Unit and products have the usual *rightist* typing rules (though they could also be presented in a *leftist* style). Sums have the usual *leftist* typing rules, with two injections, and a case construct. As is standard, functions are *rightist* – they are introduced by lambda abstraction, binding a value $x : A$ in the body $e : B$, producing a term of type $A \Longrightarrow B$. Application eliminates a function, which applies a function $A \Longrightarrow B$ to an A, producing a B.

Now we add two more rules which look completely symmetric – colambda binds a covalue $x : \tilde{A}$ in the body $e : B$, producing a term of type $A + B$, which we've been calling cofunctions. To eliminate a cofunction, we have coapplication, which applies a cofunction $A + B$ to a term of type $\tilde{A}$, cancelling out the A and producing a B. This is a *rightist* rule for sums. The use of the same $+$ type for sums and cofunctions is an *important* design choice, and is justified by the semantics developed in § 3. Sums have *bipartisan* status – we will see that the interaction of the leftist case and rightist $\tilde{\lambda}$ constructs leads to control effects!

The slogan for the typing rule for functions is: "binding a value produces a function". Dually, the slogan for the typing rule for cofunctions is: "binding a covalue produces a choice". The typing rule for application says "a function consumes a value", and the typing rule for coapplication says "a cofunction consumes a covalue". When a function binds a value A, it can use the bound variable $x : A$ in its body in any way that satisfies typing constraints, and similarly, when a cofunction

$$
\begin{array}{lrcl}
\textsc{Types} & A, B & ::= & \mathbf{1} \mid A \times B \mid A + B \mid A \Rightarrow B \mid \tilde{A} \\
\textsc{Terms} & e & ::= & x \mid * \mid (e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
 & & \mid & \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}(e_1, x.\, e_2, y.\, e_3) \\
 & & \mid & \lambda(x : A).\, e \mid e_1\ e_2 \mid \widetilde{\lambda}(x : \tilde{A}).\, e \mid \widetilde{e_1\ e_2} \\
\textsc{Values} & v & ::= & x \mid * \mid (v_1, v_2) \mid \mathsf{fst}(v) \mid \mathsf{snd}(v) \\
 & & \mid & \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid \mathsf{case}(v_1, x.\, v_2, y.\, v_3) \\
 & & \mid & \lambda(x : A).\, e \\
\textsc{Contexts} & \Gamma, \Delta, \Psi & ::= & \cdot \mid \Gamma, x : A \\
\textsc{Substitutions} & \theta, \phi & ::= & \langle\,\rangle \mid \langle \theta, {}^{v}/_{x} \rangle
\end{array}
$$

(a) Grammar for $\lambda\tilde{\lambda}$

$x : A \in \Gamma$	x is a variable of type A in context Γ
$\Gamma \supseteq \Delta$	Γ is a weakening of Δ
$\Gamma \vdash \theta : \Delta$	θ is a substitution from Γ to Δ
$\Gamma \vdash e : A$	e is an expression of type A in context Γ
$\Gamma \vdash e_1 \approx e_2 : A$	e_1 and e_2 are equal expressions of type A in context Γ

(b) Judgements for $\lambda\tilde{\lambda}$

$$
\frac{}{\Gamma \vdash * : \mathbf{1}}\ \mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\ \times\mathrm{I} \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}(e) : A}\ \times\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}(e) : B}\ \times\mathrm{E}_2
$$

$$
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}(e) : A + B}\ +\mathrm{I}_1 \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}(e) : A + B}\ +\mathrm{I}_2
$$

$$
\frac{\Gamma \vdash e_1 : A + B \quad \Gamma, x : A \vdash e_2 : C \quad \Gamma, y : B \vdash e_3 : C}{\Gamma \vdash \mathsf{case}(e_1, x.\, e_2, y.\, e_3) : C}\ +\mathrm{E}
$$

$$
\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).\, e : A \Rightarrow B}\ \Rightarrow\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \Rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ \Rightarrow\mathrm{E}
$$

$$
\frac{\Gamma, x : \tilde{A} \vdash e : B}{\Gamma \vdash \widetilde{\lambda}(x : \tilde{A}).\, e : A + B}\ \widetilde{\Rightarrow}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A + B \quad \Gamma \vdash e_2 : \tilde{A}}{\Gamma \vdash \widetilde{e_1\ e_2} : B}\ \widetilde{\Rightarrow}\mathrm{E}
$$

(c) Typing rules for $\lambda\tilde{\lambda}$

Fig. 1. Syntax and typing for $\lambda\tilde{\lambda}$

binds a covalue $\tilde{A}$, it can use the bound variable $x : \tilde{A}$ in its body in any way that satisfies typing constraints.

Weakening and Substitution. Since we have introduced new binders in our language, we need to show that weakening and substitution are still admissible. Unlike mixed substitution in CPS calculi, we only need call-by-value substitution. We describe the weakening and substitution rules for $\lambda\tilde{\lambda}$ in Fig. 2, and define substitution on raw terms in Definitions 2 and 3. This is standard, and when

$$\frac{}{x : A \in (\Gamma, x : A)}\ \in\text{-ID} \qquad \frac{x : A \in \Gamma \quad (x \neq y)}{x : A \in (\Gamma, y : B)}\ \in\text{-EX}$$

(a) Context Membership Rules

$$\frac{}{\cdot \supseteq \cdot}\ \supseteq\text{-ID} \qquad \frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta, x : A}\ \supseteq\text{-CONG} \qquad \frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta}\ \supseteq\text{-WK}$$

(b) Weakening Rules

$$\frac{}{\Gamma \vdash \langle\,\rangle : \cdot}\ \text{SUB-ID} \qquad \frac{\Gamma \vdash \theta : \Delta \quad \Gamma \vdash v : A}{\Gamma \vdash \langle \theta, {}^{v}/_{x} \rangle : \Delta, x : A}\ \text{SUB-VAL}$$

(c) Substitution Rules

Fig. 2. Membership, Weakening, and Substitution Rules

substituting under a binder, we do a renaming of the bound variable by extending the substitution. Finally, we prove admissibility of weakening and substitution.

Theorem 1 (Weakening and Substitution).

- *If* $\Gamma \supseteq \Delta$ *and* $\Delta \vdash e : A$, *then* $\Gamma \vdash e : A$.
- *If* $\Gamma \vdash \theta : \Delta$ *and* $\Delta \vdash e : A$, *then* $\Gamma \vdash \theta(e) : A$.

Definition 2 (Substitution on variables).

$$\theta[x] \triangleq \begin{cases} \lightning & \theta = \langle\,\rangle \\ e & \theta = \langle \phi, {}^{e}/_{x} \rangle \\ \phi[x] & \theta = \langle \phi, {}^{e}/_{y} \rangle, x \neq y \end{cases}$$

Definition 3 (Substitution on raw terms).

$$\begin{aligned}
\theta(*) &\triangleq * & \theta((e_1, e_2)) &\triangleq (\theta(e_1), \theta(e_2)) \\
\theta(\mathsf{fst}(e)) &\triangleq \mathsf{fst}(\theta(e)) & \theta(\mathsf{snd}(e)) &\triangleq \mathsf{snd}(\theta(e)) \\
\theta(x) &\triangleq \theta[x] & \theta(\mathsf{case}(e, {}^{x.\, e_1}_{y.\, e_2})) &\triangleq \mathsf{case}(\theta(e), {}^{z.\, \langle \theta, {}^{z}/_{x} \rangle(e_1)}_{z.\, \langle \theta, {}^{z}/_{y} \rangle(e_2)}) \\
\theta(\mathsf{inl}(e)) &\triangleq \mathsf{inl}(\theta(e)) & \theta(\mathsf{inr}(e)) &\triangleq \mathsf{inr}(\theta(e)) \\
\theta(\lambda x.\, e) &\triangleq \lambda y.\, \langle \theta, {}^{y}/_{x} \rangle(e) & \theta(e_1\, e_2) &\triangleq \theta(e_1)\, \theta(e_2) \\
\theta(\tilde{\lambda} x.\, e) &\triangleq \tilde{\lambda} y.\, \langle \theta, {}^{y}/_{x} \rangle(e) & \theta(\widetilde{e_1\, e_2}) &\triangleq \widetilde{\theta(e_1)\, \theta(e_2)}
\end{aligned}$$

Without formally giving a computational interpretation to $\lambda\tilde{\lambda}$ programs, we can informally think of colambdas as producing a right-biased speculative choice, bargaining for a covalue for the left side of the choice. Then, a covalue opens up a "channel" for the left side of the sum, which can be used by the body of the cofunction to "escape" to the left, despite having previously made a preference for the right side of the sum. This escape mechanism is a way for values and covalues to interact, or using our analogy, a way to send a value on the channel opened up by the covalue. To develop these ideas, we first need to give semantics to our language.

$$
\begin{aligned}
(\!|*|\!)^{\gamma}_{\mathbf{1}} &\triangleq \lambda k.\, k\ * \\
(\!|(e_1, e_2)|\!)^{\gamma}_{A\times B} &\triangleq \lambda k.\, (\!|e_2|\!)^{\gamma}_{B}(\lambda b.\, (\!|e_1|\!)^{\gamma}_{A}(\lambda a.\, k\ (a, b))) \\
(\!|\mathsf{fst}(e)|\!)^{\gamma}_{A} &\triangleq \lambda k.\, (\!|e|\!)^{\gamma}_{A\times B}(\lambda p.\, k\ \mathsf{fst}(p)) \\
(\!|\mathsf{snd}(e)|\!)^{\gamma}_{B} &\triangleq \lambda k.\, (\!|e|\!)^{\gamma}_{A\times B}(\lambda p.\, k\ \mathsf{snd}(p)) \\
(\!|\mathsf{inl}(e)|\!)^{\gamma}_{A+B} &\triangleq \lambda k.\, (\!|e|\!)^{\gamma}_{A}(\lambda a.\, k\ \mathsf{inl}(a)) \\
(\!|\mathsf{inr}(e)|\!)^{\gamma}_{A+B} &\triangleq \lambda k.\, (\!|e|\!)^{\gamma}_{B}(\lambda b.\, k\ \mathsf{inr}(b)) \\
(\!|\mathsf{case}(e, x.\, e_1, y.\, e_2)|\!)^{\gamma}_{C} &\triangleq \lambda k.\, (\!|e|\!)^{\gamma}_{A+B}(\lambda \begin{cases} \mathsf{inl}(a).\ (\!|e_1|\!)^{\gamma,a}_{C}\ k \\ \mathsf{inr}(b).\ (\!|e_2|\!)^{\gamma,b}_{C}\ k \end{cases}) \\
(\!|x|\!)^{\gamma}_{A} &\triangleq \lambda k.\, k(\gamma(x)) \\
(\!|\lambda x.\, e|\!)^{\gamma}_{A\Rightarrow B} &\triangleq \lambda k.\, k(\lambda a.\, \lambda k_B.\, (\!|e|\!)^{\gamma,a}_{B}\ k_B) \\
(\!|e_1\ e_2|\!)^{\gamma}_{B} &\triangleq \lambda k_B.\, (\!|e_2|\!)^{\gamma}_{A}(\lambda a.\, (\!|e_1|\!)^{\gamma}_{A\Rightarrow B}(\lambda f.\, f\ a\ k_B)) \\
(\!|\widetilde{\lambda} x.\, e|\!)^{\gamma}_{A+B} &\triangleq \lambda k.\ \mathit{let} \begin{cases} k_A \triangleq \lambda a.\, k\ \mathsf{inl}(a) \\ k_B \triangleq \lambda b.\, k\ \mathsf{inr}(b) \end{cases} \\
&\qquad \mathit{in}\ (\!|e|\!)^{\gamma,k_A}_{B}(k_B) \\
(\!|\widetilde{e_1\ e_2}|\!)^{\gamma}_{B} &\triangleq \lambda k_B.\, (\!|e_2|\!)^{\gamma}_{\tilde{A}}(\lambda k_A.\, (\!|e_1|\!)^{\gamma}_{A+B}(\lambda \begin{cases} \mathsf{inl}(a).\ k_A\ a \\ \mathsf{inr}(b).\ k_B\ b \end{cases}))
\end{aligned}
$$

Fig. 3. Continuation semantics for $\lambda\tilde{\lambda}$

3 Semantics

3.1 Continuation Semantics

Those familiar with continuations will recognize that these covalues look like continuations! Indeed, we can interpret this language using continuation semantics, by giving a CPS translation into a target language, shown in Fig. 3, which is familiar in the continuations literature (e.g., see [Streicher and Reus 1998]). It is given by a family of semantic functions indexed by types and contexts: $(\!|-|\!)^{\Gamma}_{A} : \Gamma \vdash e : A \to (\!|\Gamma|\!) \to ((\!|A|\!) \to R) \to R$. The target language is assumed to have functions, products, and sums, with a fixed answer (or response) type R, and the usual constructs follow the standard call-by-value semantics, fixing a *right-to-left* evaluation order. On types, the $\tilde{A}$ type is translated as $(\!|A|\!) \to R$, and the function type $A \Rightarrow B$ as $(\!|A|\!) \to ((\!|B|\!) \to R) \to R$. The function type is sometimes translated as $(\!|A|\!) \times ((\!|B|\!) \to R) \to R$, which is equivalent upto currying – we have chosen to write it this way to match the Kleisli interpretation. The context γ is often omitted in CPS translations – we include it here to precisely show how context lookup and extension works. A formalized, type-correct-by-construction CPS translation is given in the supplementary material.

The semantics of a lambda $\lambda x.\, e : A \Rightarrow B$ grabs a continuation $k : ((\!|A|\!) \to ((\!|B|\!) \to R) \to R) \to R$, which is applied to a function that binds a value $a : (\!|A|\!)$, a continuation $k_B : (\!|B|\!) \to R$, and evaluates the body e in the extended context (γ, a), using the continuation k_B. When translating an application $e_1\ e_2 : B$, we grab a continuation k_B, first evaluating the argument e_2, which requires a continuation $(\!|A|\!) \to R$. We pass a continuation that binds the value $a : (\!|A|\!)$, then evaluates the function e_1 in the same context, passing a and k_B as arguments.

Dual to functions, the semantics of a colambda $\widetilde{\lambda} x.\, e : A+B$ grabs a continuation $k : ((\!|A|\!) + (\!|B|\!)) \to R$, which we (crucially) split into two continuations $k_A : (\!|A|\!) \to R$ and $k_B : (\!|B|\!) \to R$. We pass the

continuation k_A into the environment γ, and evaluate the body e in this extended environment, continuing with k_B. To translate a coapplication $\widetilde{e_1\ e_2} : B$, we grab a continuation k_B, then evaluate the argument e_2, which requires a continuation $(\llparenthesis A \rrparenthesis \to R) \to R$. We pass a continuation which binds the covalue (or continuation) $k_A : \llparenthesis A \rrparenthesis \to R$, then evaluates the body e_1 in the same context. This requires a continuation $(\llparenthesis A \rrparenthesis + \llparenthesis B \rrparenthesis) \to R$, which we define by cases. When it receives an $\mathsf{inl}(a)$, the computation continues as k_A applied to a – when it receives an $\mathsf{inr}(b)$, the computation continues as k_B applied to b. From this semantics, we see that coabstraction and coapplication act as binding operators for continuations – managing the two continuations for a sum type, justifying the phrase "higher-order continuations".

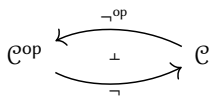
3.2 Loch Ness Semantics

The continuation semantics in the previous section makes it seem as if cofunctions were pulled out of a hat, and does not explain the conceptual reason, or beauty behind the duality in this language. The right way to understand this language is to understand the abstract structure of the continuation semantics using category theory. This is also necessary to further develop the metatheory of our language. The ideas here are well-known in the semantics of continuations – we choose to develop them in a way which matches the design of the language. There are different approaches to axiomatizing the categorical semantics of continuations [Levy 2001, § 8.8]:

- axiomatizing the type of continuations $\neg A$ or $\tilde{A}$ directly, following Thielecke [1997], or
- axiomatizing non-returning functions using an exponentiating object, following Hofmann [1995], Streicher and Reus [1998], and Hofmann and Streicher [2002].

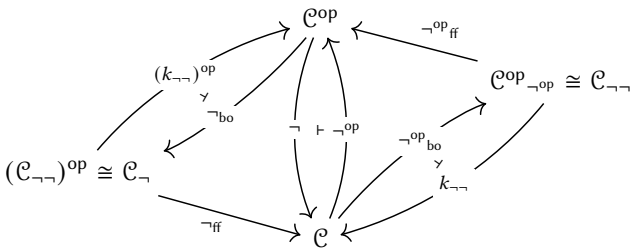
We develop the first point of view abstractly, then instantiate it with the second point of view.

A Micrological Study of Continuations. Let $\mathcal{C}$ be a (locally small) category with a functor $\neg : \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$ that is self-adjoint on the right [Thielecke 1997]. This means, for any objects $A, B \in \mathcal{C}$, we have the hom-set isomorphism: $\mathcal{C}^{\mathrm{op}}(\neg^{\mathrm{op}} A, B) \cong \mathcal{C}(A, \neg B)$, or equivalently, $\mathcal{C}(B, \neg A) \cong \mathcal{C}(A, \neg B)$, natural in A and B (the $^{\mathrm{op}}$ is dropped from $\neg^{\mathrm{op}}$ on objects).



Any functor can be split into a bijective-on-objects functor followed by a fully-faithful functor, factoring through the full image, which is called the (bo,ff) factorisation. We will do this to both the negation functors, $\neg$ and $\neg^{\mathrm{op}}$. The full image of $\neg : \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$, written $\overline{\mathsf{im}}(\neg) \equiv \mathcal{C}_{\neg}$, upto isomorphism of categories, is determined by $\mathcal{C}_{\neg}(A, B) \triangleq \mathcal{C}(\neg A, \neg B)$. The functor $\neg_{\mathsf{bo}} : \mathcal{C}^{\mathrm{op}} \to \mathcal{C}_{\neg}$ is, in particular, identity on objects (ioo), and negates morphisms, while the functor $\neg_{\mathsf{ff}} : \mathcal{C}_{\neg} \to \mathcal{C}$ negates objects, and is identity on morphisms. Dually, the full image of $\neg^{\mathrm{op}} : \mathcal{C} \to \mathcal{C}^{\mathrm{op}}$ is $\overline{\mathsf{im}}(\neg^{\mathrm{op}}) \equiv \mathcal{C}^{\mathrm{op}}_{\neg}$, with the corresponding splittings.

The self-dual adjunction gives a $\neg\neg$ monad on $\mathcal{C}$ (and a comonad on $\mathcal{C}^{\mathrm{op}}$). We write $\mathcal{C}_{\neg\neg}$ for the Kleisli category of this monad, and its Kleisli resolution is the adjunction $v_{\neg\neg} \dashv k_{\neg\neg} : \mathcal{C}_{\neg\neg} \to \mathcal{C}$. Further, by the self-adjointness of $\neg$, the dual of the Kleisli category coincides with the full image of $\neg$, that is, $(\mathcal{C}_{\neg\neg})^{\mathrm{op}} \cong \mathcal{C}_{\neg}$ (and dually for $\neg^{\mathrm{op}}$). Explicitly writing out all the adjunctions, a picture emerges.



Remarkably, all the facts about the duality of values and continuations can be derived from looking at this picture, and applying formal categorical reasoning to understand the abstract properties of each functor, giving various CPS translations (left as an exercise for the interested reader).

Proposition 4.

(1) $\neg\colon \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$ *preserves limits, and* $\neg^{\mathrm{op}}\colon \mathcal{C} \to \mathcal{C}^{\mathrm{op}}$ *preserves colimits.*
(2) $\neg_{\mathrm{bo}}$ *is a right adjoint, and preserves limits. Dually,* $\neg^{\mathrm{op}}_{\mathrm{bo}}$ *is a left adjoint, and preserves colimits.*
(3) $\mathcal{C}_{\neg}$ *is equivalent to the opposite of the Kleisli category of the* $\neg\neg$ *monad on* $\mathcal{C}$*, and* $\mathcal{C}^{\mathrm{op}}_{\neg}$ *is equivalent to the Kleisli category,* $\mathcal{C}_{\neg\neg}$*.*

Proof. Part (1) follows from right-adjoints preserving limits, and left-adjoints preserving colimits (LAPC/RAPL). For part (2), we observe that $\mathcal{C}^{\mathrm{op}}(\neg\neg A, B) \equiv \mathcal{C}(B, \neg\neg A) \cong \mathcal{C}(\neg A, \neg B) \equiv \mathcal{C}_{\neg}(A, B) \equiv \mathcal{C}_{\neg}(A, \neg_{\mathrm{bo}} B)$, making $\neg_{\mathrm{bo}}$ a right adjoint. Finally, part (3) follows from $\mathcal{C}_{\neg}(A, B) = \mathcal{C}(\neg A, \neg B) \cong \mathcal{C}(B, \neg\neg A) \equiv \mathcal{C}^{\mathrm{op}}_{\neg\neg}(A, B)$. □

In this paper, we are only interested in a subset of this picture – the Kleisli adjunction relating the categories $\mathcal{C}$ and $\mathcal{C}_{\neg\neg}$, and starting from a bicartesian closed category $\mathcal{C}$, with a negation functor given by a fixed exponentiating object $\neg = R^{(-)}$, which will allow us to exploit Moggi [1989]'s framework for the computational lambda calculus. We calculate the structure of $\mathcal{C}_{\neg}$ and $\mathcal{C}_{\neg\neg}$.

Proposition 5. *If* $\mathcal{C}$ *is bicartesian, we have*

(1) $\neg 0 \cong 1$ *and* $\neg(A + B) \cong \neg A \times \neg B$*.*
(2) $\mathcal{C}_{\neg}$ *is cartesian, with products given by coproducts in* $\mathcal{C}$*.*
(3) $\neg_{\mathrm{ff}}$ *preserves products, and reflects exponentials.*

In a closed category, taking exponentials with a fixed response object gives a $\neg$ functor.

Proposition 6. *If* $\mathcal{C}$ *is bicartesian closed with a fixed object* R *(the object of responses),*

(1) $\neg \triangleq R^{(-)}$ *is a* self-adjoint on the right *negation functor.*
(2) $\neg\neg$ *is a strong monad on* $\mathcal{C}$*, and has Kleisli exponentials.*
(3) $\mathcal{C}_{\neg}$ *is cartesian closed, with exponentials* $B \Rightarrow C$ *given by* $C \times R^B$*.*
(4) $\neg_{\mathrm{ff}}$ *is a cartesian closed functor.*
(5) The Kleisli category of $\neg\neg$ *is cocartesian coclosed, and premonoidal.*

The observation that the Kleisli category of the $\neg\neg$ (strong) monad is cocartesian coclosed is the key to our entire development. We have two different categories, a cartesian closed category of values, and a cocartesian coclosed category of computations, but we can still put them together in a *call-by-value programming language* using the ideas of Moggi [1989]! There is no mathematical trickery here – we're simply exploiting well-known mathematical structure and *dressing it up*. This is only a matter of appearances – a common trait of good magic tricks, and programming language design. Following Taylor [2002], if values are X, we think of covalues/continuations as

$$\llbracket \tilde{A} \rrbracket \triangleq \neg \llbracket A \rrbracket \qquad \llbracket A \Rightarrow B \rrbracket \triangleq (K\llbracket B \rrbracket)^{\llbracket A \rrbracket}$$

$$\llbracket A \times B \rrbracket \triangleq \llbracket A \rrbracket \times \llbracket B \rrbracket \qquad \llbracket A + B \rrbracket \triangleq \llbracket A \rrbracket + \llbracket B \rrbracket$$

(a) $\llbracket A \rrbracket$: Obj

$$\llbracket \cdot \rrbracket \triangleq 1 \qquad \llbracket \Gamma, x : A \rrbracket \triangleq \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket$$

(b) $\llbracket \Gamma \rrbracket$: Obj

Fig. 4. Interpretation of types and contexts

observations R^X, and computations R^{R^X} as meta-observations. Explicitly, there are two adjunctions (where $A \Rightarrow B$ is $A \to \neg\neg B$):

$$\mathcal{C}_{\neg\neg}(C \times A, B) \cong \mathcal{C}(C, A \Rightarrow B) \qquad \text{and} \qquad \mathcal{C}_{\neg\neg}(C \times \neg A, B) \cong \mathcal{C}_{\neg\neg}(C, A + B) \ .$$

The existence of the coclosure is a surprising mathematical fact, so we have derived it from an abstract point of view, which gives a conceptual understanding of where it *comes from*. However, one could simply give an explicit calculation of this isomorphism (see Appendix A). Note, in particular, that only the response object is required to be exponentiating. The abstract framework can also be instantiated with different structure on $\mathcal{C}$ (such as symmetric monoidal closure with coproducts, fixpoints, etc.), leading to other dual calculi.

In terms of Selinger [2001, Remark 1.1], $\mathcal{C}_\neg$ is a control category (the interpretation of call-by-name), and the Kleisli category $\mathcal{C}_{\neg\neg}$ is a co-control category (the interpretation of call-by-value). Using all the structure developed, we can now give a categorical and denotational semantics to $\lambda\tilde{\lambda}$.

To work with exponentials and co-exponentials, we adopt the musical notation of adjuncts (flats on the left, sharps on the right). Currying/uncurrying is the right/left adjunct operation in $(-) \times X \dashv (-)^X$. Co-currying/co-uncurrying is the left/right adjunct operation in ${}^X(-) \dashv X + (-)$.

Definition 7 (Exponentials and Coexponentials).

- The exponential of B by A is written as B^A.
- Given $f : C \times A \to B$, the currying of f is $f^\sharp : C \to B^A$.
- Given $g : C \to B^A$, the uncurrying of g is $g^\flat : C \times A \to B$.
- Evaluation is $\mathsf{ev}_{A,B} : B^A \times A \to B \triangleq 1^\flat_{B^A}$.
- The co-exponential of B by A is written as ${}^A B$.
- Given $f : B \to A + C$, the co-currying of f is $f^\flat : {}^A B \to C$.
- Given $g : {}^A B \to C$, the co-uncurrying of g is $g^\sharp : B \to A + C$.
- Co-evaluation is $\mathsf{coev}_{A,B} : B \to A + {}^A B \triangleq 1^\sharp_{{}^A B}$.

3.3 Interpretation

We now give an interpretation for $\lambda\tilde{\lambda}$ using the categorical structure we've defined.

Types and Contexts. Types and contexts are interpreted as objects in $\mathcal{C}$, as shown in Fig. 4.

Expressions. Expressions are interpreted as Kleisli arrows, that is, morphisms in $\mathcal{C}_K$ (for $K = \neg\neg$). The full interpretation is given in Fig. 5. This is standard call-by-value semantics following Moggi [1989], extended with sums and cofunctions. Sums use the cocartesian structure, and distributivity for case. Cofunctions are interpreted using the coexponential adjunction. We also interpret weakening and substitution, and prove appropriate coherence lemmas in § 3.3.

Weakening and Substitution. Membership and weakening are interpreted using projections of contexts, as shown in Fig. 6. To interpret substitutions, we need a value interpretation. The

$$[\![\frac{}{\Gamma \vdash * : \mathbf{1}}]\!] \triangleq !_\Gamma \fatsemi \eta_1$$

$$[\![\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}]\!] \triangleq \begin{array}{l} let \begin{cases} f \triangleq [\![\Gamma \vdash e_1 : A]\!] \\ g \triangleq [\![\Gamma \vdash e_2 : B]\!] \end{cases} \\ in \langle f , g \rangle \fatsemi \beta_{A,B} \end{array}$$

$$[\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}(e) : A}]\!] \triangleq [\![\Gamma \vdash e_1 : A]\!] \fatsemi K\pi_1 \quad [\![\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}(e) : B}]\!] \triangleq [\![\Gamma \vdash e_2 : B]\!] \fatsemi K\pi_2$$

$$[\![\frac{x : A \in \Gamma}{\Gamma \vdash x : A}]\!] \triangleq [\![x : A \in \Gamma]\!] \fatsemi \eta_A$$

$$[\![\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).\, e : A \Rightarrow B}]\!] \triangleq \begin{array}{l} let \quad f \triangleq [\![\Gamma, x : A \vdash e : B]\!] \\ in \ f^\sharp \fatsemi \eta_{A \to KB} \end{array}$$

$$[\![\frac{\Gamma \vdash e_1 : A \Rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}]\!] \triangleq \begin{array}{l} let \begin{cases} f \triangleq [\![\Gamma \vdash e_1 : A \Rightarrow B]\!] \\ g \triangleq [\![\Gamma \vdash e_2 : A]\!] \end{cases} \\ in \langle f , g \rangle \fatsemi \beta_{A \to KB, A} \fatsemi K\mathsf{ev}_{A,KB} \fatsemi \mu_B \end{array}$$

$$[\![\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}(e) : A + B}]\!] \triangleq [\![\Gamma \vdash e : A]\!] \fatsemi K\iota_1 \quad [\![\frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}(e) : A + B}]\!] \triangleq [\![\Gamma \vdash e : B]\!] \fatsemi K\iota_2$$

$$[\![\frac{\Gamma \vdash e_1 : A + B \quad \Gamma, x : A \vdash e_2 : C \quad \Gamma, y : B \vdash e_3 : C}{\Gamma \vdash \mathsf{case}(e_1, x.\, e_2, y.\, e_3) : C}]\!] \triangleq \begin{array}{l} let \begin{cases} f \triangleq [\![\Gamma \vdash e_1 : A + B]\!] \\ g_1 \triangleq [\![\Gamma, x : A \vdash e_2 : C]\!] \\ g_2 \triangleq [\![\Gamma, y : B \vdash e_3 : C]\!] \end{cases} \\ in \langle 1_\Gamma , f \rangle \fatsemi \sigma_{\Gamma,A} \fatsemi K\delta_{\Gamma,A,B} \fatsemi K[g_1 , g_2] \fatsemi \mu_C \end{array}$$

$$[\![\frac{\Gamma, x : \tilde{A} \vdash e : B}{\Gamma \vdash \widetilde{\lambda}(x : \tilde{A}).\, e : A + B}]\!] \triangleq \begin{array}{l} let \quad f \triangleq [\![\Gamma, x : \tilde{A} \vdash e : B]\!] \\ in \ f^\sharp \end{array}$$

$$[\![\frac{\Gamma \vdash e_1 : A + B \quad \Gamma \vdash e_2 : \tilde{A}}{\Gamma \vdash \widetilde{e_1\ e_2} : B}]\!] \triangleq \begin{array}{l} let \begin{cases} f \triangleq [\![\Gamma \vdash e_1 : A + B]\!] \\ g \triangleq [\![\Gamma \vdash e_2 : \tilde{A}]\!] \end{cases} \\ in \langle f , g \rangle \fatsemi \tau_{K(A+B), R^A} \fatsemi K1^\flat_{K(A+B)} \fatsemi \mu_B \end{array}$$

Fig. 5. Interpretation of expressions, $[\![\Gamma \vdash e : A]\!] : \mathsf{Hom}([\![\Gamma]\!], K[\![A]\!])$

interpretation of values and substitutions is shown in Fig. 7. The value interpretation is coherent with the expression interpretation, which we use to prove semantic weakening and substitution.

Theorem 8 (Semantic Weakening and Substitution).

- *If* $\Gamma \vdash v : A$, *then* $[\![\Gamma \vdash v : A]\!] = [\![\Gamma \vdash v : A]\!]_v \fatsemi \eta_A$.
- *If* $\Gamma \supseteq \Delta$ *and* $\Delta \vdash e : A$, *then* $[\![\Gamma \vdash e : A]\!] = \mathsf{Wk}(\Gamma \supseteq \Delta) \fatsemi [\![\Delta \vdash e : A]\!]$.
- *If* $\Gamma \vdash \theta : \Delta$ *and* $\Delta \vdash e : A$, *then* $[\![\Gamma \vdash \theta(e) : A]\!] = [\![\Gamma \vdash \theta : \Delta]\!] \fatsemi [\![\Delta \vdash e : A]\!]$.

Soundness. Using § 3.2, and unpacking the definition of the continuation monad $R^{R^{(-)}}$, we show that this agrees with the CPS translation in § 3.1.

Theorem 9. *If* $\Gamma \vdash e : A$, *then* $(\!|e|\!)^\gamma_A(k) = [\![\Gamma \vdash e : A]\!](\gamma, k)$, *for any* $\gamma \colon [\![\Gamma]\!]$ *and* $k \colon [\![A]\!] \to R$.

$$[\![\frac{}{\cdot \supseteq \cdot}]\!] \triangleq 1_1$$

$$[\![\frac{}{x : A \in (\Gamma, x : A)}]\!] \triangleq \pi_2 \qquad [\![\frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta}]\!] \triangleq \pi_1 \,;\, [\![\Gamma \supseteq \Delta]\!]$$

$$[\![\frac{x : A \in \Gamma \quad (x \neq y)}{x : A \in (\Gamma, y : B)}]\!] \triangleq \pi_1 \,;\, [\![x : A \in \Gamma]\!] \qquad [\![\frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta, x : A}]\!] \triangleq [[\![\Gamma \supseteq \Delta]\!] \times 1_A]$$

(a) $[\![x : A \in \Gamma]\!] : \mathsf{Hom}([\![\Gamma]\!], [\![A]\!])$ (b) $\mathsf{Wk}(\Gamma \supseteq \Delta) \triangleq [\![\Gamma \supseteq \Delta]\!] : \mathsf{Hom}([\![\Gamma]\!], [\![\Delta]\!])$

Fig. 6. Interpretation of Membership and Weakening

$$[\![\frac{}{\Gamma \vdash * : \mathbf{1}}]\!]_v \triangleq !_\Gamma$$

$$[\![\frac{x : A \in \Gamma}{\Gamma \vdash x : A}]\!]_v \triangleq [\![x : A \in \Gamma]\!]$$

$$[\![\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).\, e : A \Rightarrow B}]\!]_v \triangleq [\![\Gamma, x : A \vdash e : B]\!]^\sharp$$

$$[\![\frac{\Gamma \vdash v_1 : A \quad \Gamma \vdash v_2 : B}{\Gamma \vdash (v_1, v_2) : A \times B}]\!]_v \triangleq \langle [\![\Gamma \vdash v_1 : A]\!]_v \,,\, [\![\Gamma \vdash v_2 : B]\!]_v \rangle$$

$$[\![\frac{\Gamma \vdash v_1 : A + B \quad \Gamma, x : A \vdash v_2 : C \quad \Gamma, y : B \vdash v_3 : C}{\Gamma \vdash \mathsf{case}(v_1, x.\, v_2, y.\, v_3) : C}]\!]_v \triangleq \mathit{let} \begin{cases} f & \triangleq & [\![\Gamma \vdash v_1 : A + B]\!]_v \\ g_1 & \triangleq & [\![\Gamma, x : A \vdash v_2 : C]\!]_v \\ g_2 & \triangleq & [\![\Gamma, y : B \vdash v_3 : C]\!]_v \end{cases} \mathit{in}\ \langle 1_\Gamma \,,\, f \rangle \,;\, \delta_{\Gamma,A,B} \,;\, [g_1 \,,\, g_2]$$

$$[\![\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathsf{inl}(v) : A + B}]\!]_v \triangleq [\![\Gamma \vdash v : A]\!]_v \,;\, \iota_1 \qquad [\![\frac{\Gamma \vdash v : A \times B}{\Gamma \vdash \mathsf{fst}(v) : A}]\!]_v \triangleq [\![\Gamma \vdash v : A \times B]\!]_v \,;\, \pi_1$$

$$[\![\frac{\Gamma \vdash v : B}{\Gamma \vdash \mathsf{inr}(v) : A + B}]\!]_v \triangleq [\![\Gamma \vdash v : B]\!]_v \,;\, \iota_2 \qquad [\![\frac{\Gamma \vdash v : A \times B}{\Gamma \vdash \mathsf{snd}(v) : B}]\!]_v \triangleq [\![\Gamma \vdash v : A \times B]\!]_v \,;\, \pi_2$$

(a) $[\![\Gamma \vdash v : A]\!]_v : \mathsf{Hom}([\![\Gamma]\!], [\![A]\!])$

$$[\![\frac{}{\Gamma \vdash \langle\rangle : \cdot}]\!] \triangleq !_\Gamma$$

$$[\![\frac{\Gamma \vdash \theta : \Delta \quad \Gamma \vdash v : A}{\Gamma \vdash \langle \theta, v/x \rangle : \Delta, x : A}]\!] \triangleq \langle [\![\Gamma \vdash \theta : \Delta]\!] \,,\, [\![\Gamma \vdash v : A]\!]_v \rangle$$

(b) $[\![\Gamma \vdash \theta : \Delta]\!] : \mathsf{Hom}([\![\Gamma]\!], [\![\Delta]\!])$

Fig. 7. Interpretation of values and substitution

$$\frac{\Gamma, x : A \vdash e : B \quad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A).\, e)\ v \approx [v/x]e : B}\ \lambda\beta \qquad \frac{\Gamma \vdash v : A \Rightarrow B}{\Gamma \vdash (\lambda(x : A).\, v\ x) \approx v : A \Rightarrow B}\ \lambda\eta$$

$$\frac{\Gamma, x : \tilde{A} \vdash e : B \quad \Gamma \vdash v : \tilde{A}}{\Gamma \vdash \widetilde{(\tilde{\lambda}(x : \tilde{A}).\, e)}\ v \approx [v/x]e : B}\ \tilde{\lambda}\beta \qquad \frac{\Gamma \vdash e : A + B}{\Gamma \vdash \widetilde{(\tilde{\lambda}(x : \tilde{A}).\, \widetilde{e\ x})} \approx e : A + B}\ \tilde{\lambda}\eta$$

(a) Conversion rules for the equational theory of $\lambda\tilde{\lambda}$

Fig. 8. Equational theory of $\lambda\tilde{\lambda}$

$$\begin{aligned}\mathcal{E} \quad ::= \ & \cdot \mid \mathsf{fst}(\mathcal{E}) \mid \mathsf{snd}(\mathcal{E}) \mid (e, \mathcal{E}) \mid (\mathcal{E}, v) \\ & \mid \mathsf{inl}(\mathcal{E}) \mid \mathsf{inr}(\mathcal{E}) \mid \mathsf{case}(\mathcal{E}, x.\, e_1, y.\, e_2) \\ & \mid e\ \mathcal{E} \mid \mathcal{E}\ v \mid \widetilde{\lambda x}.\, \mathcal{E} \mid \widetilde{e\ \mathcal{E}} \mid \widetilde{\mathcal{E}\ v}\end{aligned}$$

Fig. 9. Evaluation contexts

4 Equational Theory

The purpose of giving a categorical semantics is to produce an equational theory for the language – which is to be understood as an axiomatic theory generated by an operational semantics. On top of the axiomatic equational theory, we add control effects, validated by our semantics.

4.1 Conversion Rules

The equivalence and congruence rules are standard – we give some important conversion rules in Fig. 8 (and the remaining rules are in Fig. 12 in the appendix). These rules are the standard ones for call-by-value – extended with sums and co-exponentials.

The $\beta\eta$-rules come from the two adjunctions for closure and co-closure. Beta laws for both functions and cofunctions are restricted to values, since substitution holds for values. Functions satisfy eta laws upto values, because these are Kleisli exponentials. But the coexponential adjunction lives in the computation category, so cofunctions satisfy full eta laws for expressions. These equations don't perform any control effects – so far they're only exploiting the two adjunctions to validate binding of values and covalues.

4.2 Control Effects

The real test for an equational theory of continuations is in the axiomatics of control operators [Hyland et al. 2007]. In our calculus, the role of control operators is played by $\tilde{\lambda}$, case, and their interaction with sums.

We design equations for control effects in $\lambda\tilde{\lambda}$ in Fig. 10. These are inspired by Hofmann [1995] and Hofmann and Streicher [2002]'s equations, and Selinger [2001]'s equations for cbv $\lambda\mu$. We use evaluation contexts instead of commuting conversions, given in Fig. 9. In particular, we have an additional evaluation context for cofunctions, $\widetilde{\lambda x}.\ \mathcal{E}$ (as well as $\widetilde{e\ \mathcal{E}}$ and $\widetilde{\mathcal{E}\ v}$), which allows reductions to happen inside the body of a cofunction (unlike lambda abstractions). Well-typed evaluation contexts $\mathcal{E} : A \Longmapsto B$ are interpreted in $\mathcal{C}_K$, as mapping Kleisli arrows (expressions being plugged, of type A) to Kleisli arrows (plugged expressions of type B).

$\widetilde{\lambda}$-CONST is the constant (or garbage collection) rule – when the covalue is not used, it simply becomes a right-biased sum. The next two rules are the interaction of normal sums and cofunctions.

$$\frac{\Gamma \vdash e : B}{\Gamma \vdash (\widetilde{\lambda}(x : \tilde{A}).\ e) \approx \mathsf{inr}_A(e) : A + B}\ \widetilde{\lambda}\text{-CONST}$$

$$\frac{\Gamma \vdash e : B \quad \Gamma \vdash \mathcal{E} : B \Rightarrow C}{\Gamma \vdash (\widetilde{\lambda}(x : \tilde{A}).\ \mathcal{E}\langle\!\langle \widetilde{\mathsf{inr}_A(e)}\ x \rangle\!\rangle) \approx \mathsf{inr}_A(\mathcal{E}\langle\!\langle e \rangle\!\rangle) : A + C}\ \widetilde{\lambda}\text{-inr-PASS}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash \mathcal{E} : B \Rightarrow C}{\Gamma \vdash (\widetilde{\lambda}(x : \tilde{A}).\ \mathcal{E}\langle\!\langle \widetilde{\mathsf{inl}_B(e)}\ x \rangle\!\rangle) \approx \mathsf{inl}_C(e) : A + C}\ \widetilde{\lambda}\text{-inl-JUMP}$$

$$\frac{\Gamma, x : \tilde{A} \vdash v : B \quad \Gamma, y : A \vdash e_1 : C \quad \Gamma, z : B \vdash e_2 : C}{\Gamma \vdash \mathsf{case}(\widetilde{\lambda}(x : \tilde{A}).\ v, y.\ e_1, z.\ e_2) \approx \mathsf{case}(\widetilde{\lambda}(x : \tilde{A}).\ [v/z]e_2, y.\ e_1, z.\ z) : C}\ \text{CASE-}\widetilde{\lambda}\text{-}\beta$$

$$\frac{\Gamma \vdash e : A + B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C \quad \Gamma \vdash \mathcal{E} : C \Rightarrow D}{\Gamma \vdash \mathcal{E}\langle\!\langle \mathsf{case}(e, x.\ e_1, y.\ e_2) \rangle\!\rangle \approx \mathsf{case}(e, x.\ \mathcal{E}\langle\!\langle e_1 \rangle\!\rangle, y.\ \mathcal{E}\langle\!\langle e_2 \rangle\!\rangle) : D}\ \text{CASE-}\zeta$$

Fig. 10. Control effects in $\lambda\tilde{\lambda}$

$\widetilde{\lambda}$-inr-PASS is like transparent passthrough, where inr produces normal return – if the covalue was created by $\tilde{\lambda}$ but then coapplied to an inr, it might as well have never been created. $\widetilde{\lambda}$-inl-JUMP is the interesting control effect, it is a non-local "jump with argument" (as in Thielecke [1999] and Levy [2003]), where we copply to an inl, throwing to the left. This behaves like Felleisen's $\mathcal{A}$ (with Hofmann [1995]'s $\mathcal{A} - ABS$ equation), *backtracking* to where the *speculative choice* was created, resuming with the value of the argument. The evaluation context gets discarded – to an observer this speculative computation never ran.

The next control effect is the interaction of case and cofunctions. $\tilde{\lambda}$ makes a *speculative choice*, and case is trying to observe this choice. $\tilde{\lambda}$ evaluates its body to a value – then *speculatively* offers the right side of the choice to the observer. The interesting behaviour happens when the observer uses the bound covalue to do another effect! Finally, CASE-ζ is a strong extensionality rule, which duplicates the outer evaluation context of a case expression to both its branches.

To understand these equations better, we give them a workout. The well-known operational semantics of TND [Wadler 2003, Devil's Choice] can be checked by running these two programs, that try to observe $\widetilde{\lambda}x.\ x$ with case.

$$\begin{array}{ll} & \mathsf{case}(\widetilde{\lambda}x.\ x, a.\ 0, k.\ 1) \\ \rightsquigarrow & \mathsf{case}(\widetilde{\lambda}x.\ 1, a.\ 0, y.\ y) \\ \rightsquigarrow & \mathsf{case}(\mathsf{inr}(1), a.\ 0, y.\ y) \\ \rightsquigarrow & 1 \end{array} \qquad \begin{array}{ll} & \mathsf{case}(\widetilde{\lambda}x.\ x, a.\ 0, k.\ \widetilde{\mathsf{inl}(1)\ k}) \\ \rightsquigarrow & \mathsf{case}(\widetilde{\lambda}x.\ \widetilde{\mathsf{inl}(1)}\ x, a.\ 0, y.\ y) \\ \rightsquigarrow & \mathsf{case}(\mathsf{inl}(1), a.\ 0, y.\ y) \\ \rightsquigarrow & 0 \end{array}$$

These equations validate Hofmann [1995] and Hofmann and Streicher [1997]'s axiomatics of call-by-value control operators, where the encodings of control operators are given in § 6. Since we don't have **0**, some of these equations have to be adjusted from Hofmann's versions.

PROPOSITION 10. *The equational theory validates these equations:*

$$\mathcal{C}_A(\hat{\mathcal{C}}(e)) = e \qquad \mathcal{C}\text{-}APP$$

$$\mathsf{callcc}_A(\lambda(k : \tilde{A}).\ \mathcal{E}\langle\!\langle \widetilde{\mathsf{inl}(e)\ k} \rangle\!\rangle) = e \qquad \mathsf{callcc}\text{-}APP$$

$$\mathrm{call/cc}_{A,B}(\lambda(k:A \to B).\ \mathcal{E}\langle\langle k\ e\rangle\rangle) = e \qquad \text{call/cc-}ABS$$
$$\mathrm{call/cc}_{A,B}(e) = \mathrm{call/cc}_{A,C}(\lambda(k:A \to C).\ e(\lambda(x:A).\ \mathcal{E}\langle\langle kx\rangle\rangle)) \qquad \text{call/cc-}APP$$
$$\mathcal{E}\langle\langle \mathrm{call/cc}_{A,B}\rangle\rangle = \mathrm{call/cc}_{C,B}(\lambda(k:C \to B).\ \mathcal{E}\langle\langle e(\lambda(x:A).\ k(\mathcal{E}\langle\langle x\rangle\rangle))\rangle\rangle) \qquad \text{call/cc-}NAT$$

The semantic understanding of these equations comes from the axiomatics of control operators in the Kleisli category $\mathcal{C}_K$, which is given in Appendix B. It can be checked that the semantics using a response object, $\neg = R^{(-)}$ and $K = \neg\neg$, satisfies these equations, and further establish the soundness of the equational theory with respect to this continuation semantics.

Theorem 11 (Soundness). *If* $\Gamma \vdash e_1 \approx e_2 : A$, *then* $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

Proof. These are checked using universal properties, and using the value and substitution lemmas. For the control effects, we give categorical control equations. □

From this, it also follows that the equational theory is sound with respect to the continuation semantics. The question of completeness is considered in Appendix B.1.

Corollary 12. *If* $\Gamma \vdash e_1 \approx e_2 : A$, *then for any* γ *and* k, $(\!|e_1|\!)^{\gamma}_{A}(k) = (\!|e_2|\!)^{\gamma}_{A}(k)$.

5 Applications of the Duality

The $\lambda\tilde{\lambda}$ calculus highlights the duality of closure and co-closure – this new perspective can be exploited in different ways, which is the focus of this section.

The λ-calculus is a theory of higher-order functions, and can be decomposed into first-order arrow calculi: κ/ζ [Hasegawa 1995], with value/variable arrows. Building on the theme of duality – we show the decomposition of $\tilde{\lambda}$, or higher-order cofunctions, into first-order $\widetilde{\kappa}/\widetilde{\zeta}$ calculi, with covariable/covalue (co)arrows.

5.1 Dualising Functional Completeness

We will perform this decomposition conceptually, starting with the essential idea of Lambek [1974]'s functional completeness – a consequence of cartesian closure, and dualising it formally. Informally, functional completeness says that, to define a higher-order function $f : A \to B$, it is enough to bind a variable $x : A$ and produce a B. We will formalise this fact using abstract properties of (co)monads and adjunctions, reconstruct Hasegawa's calculi, and dualise each step. Starting from an observation, originally due to Hermida [1993]:

Proposition 13 (Hermida [1993, Prop. 5.2.1]). *Given a comonad* $G\colon \mathcal{C} \to \mathcal{C}$ *and its Kleisli resolution* $v_G \dashv k_G\colon \mathcal{C} \to \mathcal{C}_G$, *the following are equivalent:*

(1) G *has a right adjoint* $G \dashv T\colon \mathcal{C} \to \mathcal{C}$.
(2) k_G *has a right adjoint* $k_G \dashv R\colon \mathcal{C}_G \to \mathcal{C}$.

Under either of the above equivalent hypotheses, $T(= R \circ k_G)$ *is the functor part of a monad, and the corresponding Kleisli category* $\mathcal{C}_T$ *is isomorphic to* $\mathcal{C}_G$.

Dually, if a monad $T\colon \mathcal{C} \to \mathcal{C}$ *has Kleisli resolution* $v_T \dashv k_T\colon \mathcal{C} \to \mathcal{C}_T$, *the following are equivalent:*

(1) T *has a left adjoint* $G \dashv T\colon \mathcal{C} \to \mathcal{C}$.
(2) v_T *has a left adjoint* $L \dashv v_T\colon \mathcal{C}_T \to \mathcal{C}$.

Then, $G(= L \circ v_T)$ *is the functor part of a comonad, and the Kleisli category* $\mathcal{C}_G$ *is isomorphic to* $\mathcal{C}_T$.

Proof. Starting from (1), the functor R is given by T on objects, and for $Ga \xrightarrow{f} b$, acts on morphisms as $Ta \xrightarrow{T\eta_a} TGTa \xrightarrow{T\delta_{Ta}} TGGTa \xrightarrow{TG\epsilon_a} TGa \xrightarrow{Tf} Tb$. This makes $T = R \circ k_G$ a monad. Hermida gives a direct calculation of the monad structure. □

The informal idea is that in the λ-calculus, $C \times (-)$ is a reader/coreader/environment comonad, with a free value $1 \rightsquigarrow C$, and its right adjoint $C \Rightarrow (-)$ is a reader monad, with a free value $1 \rightsquigarrow C$ injected into its environment. Dualising this, $C \rightsquigarrow 0$ is a free covalue, and we think of $C + (-)$ as an exception monad, with a free covalue $C \rightsquigarrow 0$ in its environment – an escape hatch to jump to C. With $\tilde{\lambda}$ (or with cocartesian coclosure), this has a left adjoint comonad ${}^{C}(-)$, which merits the name: exception/coexception/handler comonad. It has a free covalue $C \rightsquigarrow 0$ injected into its environment, or a handler/continuation for C.

Proposition 14 (Folklore). *In a ccc with $c \times (-) \dashv (-)^c \colon \mathcal{C} \to \mathcal{C}$:*

(1) $c \times (-) \colon \mathcal{C} \to \mathcal{C}$ is a comonad.
(2) $(-)^c \colon \mathcal{C} \to \mathcal{C}$ is a monad.
(3) Their Kleisli categories are equivalent: $\mathcal{C}(c \times a, b) \cong \mathcal{C}(a, b^c)$.
(4) Their Kleisli categories are cartesian closed.

Dually, we have the following for a cocartesian coclosed category:

Proposition 15 (Dual to folklore). *In a coccc with ${}^{c}(-) \dashv c + (-) \colon \mathcal{C} \to \mathcal{C}$:*

(1) $c + (-) \colon \mathcal{C} \to \mathcal{C}$ is a monad.
(2) ${}^{c}(-) \colon \mathcal{C} \to \mathcal{C}$ is a comonad.
(3) Their Kleisli categories are equivalent: $\mathcal{C}({}^{c}a, b) \cong \mathcal{C}(a, c + b)$.
(4) Their Kleisli categories are cocartesian coclosed.

The Kleisli category $\mathcal{C}_{c\times(-)}$, written $\mathcal{C}[c]$, has a generic element (value) $e_c : 1 \to c$, given by $c \times 1 \xrightarrow{\sim} c$. This is Hasegawa's "fullness condition": $\mathcal{C}[c](1, -) \cong \mathcal{C}(c, -)$. Dually, the Kleisli category $\mathcal{C}_{c+(-)}$, written $\mathcal{C}[\tilde{c}]$, has a generic (co)element (covalue) $e_{\tilde{c}} : c \to 0$, given by $c \xrightarrow{\sim} c + 0$. From this, we derive functional completeness and its dual.

Proposition 16 (Lambek [1974]'s Functional Completeness and its dual). *Let $\mathcal{C}$ and $\mathcal{D}$ be cartesian closed categories and $F \colon \mathcal{C} \to \mathcal{D}$ a ccc functor. Let $c \in \mathcal{C}$, and $t : F(1) \cong 1 \to F(c)$ be an element in $\mathcal{D}$. There is a unique (upto isomorphism) extension of F to a ccc functor $\tilde{F} \colon \mathcal{C}[c] \to \mathcal{D}$, such that $\tilde{F} \circ k_{c\times(-)} \cong F$, and $\tilde{F}(e_c) = t$.*

Dually, let $\mathcal{C}$ and $\mathcal{D}$ be cocartesian coclosed categories and $F \colon \mathcal{C} \to \mathcal{D}$ a coccc functor. Let $c \in \mathcal{C}$, and $t : F(c) \to F(0) \cong 0$ be an element in $\mathcal{D}$. There is a unique (upto isomorphism) extension of F to a coccc functor $\tilde{F} \colon \mathcal{C}[\tilde{c}] \to \mathcal{D}$, such that $\tilde{F} \circ k_{c+(-)} \cong F$, and $\tilde{F}(e_{\tilde{c}}) = t$.

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{k_{c\times(-)}} & \mathcal{C}[c] \\ & \searrow^{F} & \downarrow{\scriptstyle \tilde{F}} \\ & & \mathcal{D} \end{array} \qquad \begin{array}{ccc} \mathcal{C} & \xrightarrow{k_{c+(-)}} & \mathcal{C}[\tilde{c}] \\ & \searrow^{F} & \downarrow{\scriptstyle \tilde{F}} \\ & & \mathcal{D} \end{array}$$

Proof. $\tilde{F}$ is given by F on objects, and on morphisms calculated:

$$Fa \xrightarrow{\sim} 1 \times Fa \xrightarrow{\sim} F1 \times Fa \xrightarrow{t\times Fa} Fc \times Fa \xrightarrow{\sim} F(c \times a) \xrightarrow{Ff} Fb$$
$$Fa \xrightarrow{Ff} F(c + b) \xrightarrow{\sim} Fc + Fb \xrightarrow{t+Fb} F0 + Fb \xrightarrow{\sim} 0 + Fb \xrightarrow{\sim} Fb$$

□

The Kleisli resolutions of these monads/comonads produce Hasegawa's left/right adjoints to inclusion functors, giving κ/ζ abstraction, and their duals: $\widetilde{\kappa}/\widetilde{\zeta}$ abstraction.

$$v_{c\times(-)} \dashv k_{c\times(-)} \colon \mathcal{C} \to \mathcal{C}[c] \qquad k_{(-)^c} \dashv v_{(-)^c} \colon \mathcal{C}[c] \to \mathcal{C}$$
$$k_{c+(-)} \dashv v_{c+(-)} \colon \mathcal{C}[\tilde{c}] \to \mathcal{C} \qquad v_{{}^{c}(-)} \dashv k_{{}^{c}(-)} \colon \mathcal{C} \to \mathcal{C}[\tilde{c}]$$

$$\frac{\begin{array}{c}[x : \mathbf{1} \rightsquigarrow C] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\kappa x^C.f : (C \times A) \rightsquigarrow B}\ \times\mathrm{L} \qquad \frac{\begin{array}{c}[x : \mathbf{1} \rightsquigarrow C] \\ \vdots \\ f : A \rightsquigarrow B \quad c : \mathbf{1} \rightsquigarrow C\end{array}}{(\kappa x^C.f) \circ \mathsf{lift}_A(c) \equiv f[c/x] : A \rightsquigarrow B}\ \kappa^+$$

$$\frac{c : \mathbf{1} \rightsquigarrow C}{\mathsf{lift}_A(c) : A \rightsquigarrow (C \times A)}\ \times\mathrm{R} \qquad \frac{h : (C \times A) \rightsquigarrow B}{\kappa x^C.(h \circ \mathsf{lift}_A(x)) \equiv h : (C \times A) \rightsquigarrow B}\ \kappa^-$$

(a) κ calculus

$$\frac{c : \mathbf{1} \rightsquigarrow C}{\mathsf{pass}_B(c) : (C \Rightarrow B) \rightsquigarrow B}\ \Rightarrow\mathrm{L} \qquad \frac{h : A \rightsquigarrow (C \Rightarrow B)}{\zeta x^C.(\mathsf{pass}_B(x) \circ h) \equiv h : A \rightsquigarrow (C \Rightarrow B)}\ \zeta^-$$

$$\frac{\begin{array}{c}[x : \mathbf{1} \rightsquigarrow C] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\zeta x^C.f : A \rightsquigarrow (C \Rightarrow B)}\ \Rightarrow\mathrm{R} \qquad \frac{\begin{array}{c}[x : \mathbf{1} \rightsquigarrow C] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\mathsf{pass}_B(c) \circ (\zeta x^C.f) \equiv f[c/x] : A \rightsquigarrow B}\ \zeta^+$$

(b) ζ calculus

Fig. 11. κ and ζ calculi

5.2 Dualising κ/ζ Calculi

From this analysis, we extract a presentation of the dual $\tilde{\kappa}/\tilde{\zeta}$ calculi, with substitution and equations, given in Figs. 11 and 12. We finally see the subtraction type $(-)$ type as a first-class type constructor, in Fig. 12. Referring back to Girard's comments in § 1, we argue that the subtraction connective is not a good fit for a traditional type-theoretic presentation, but is a good fit for this first-order presentation.

Just as κ and ζ can be understood as first-order languages for understanding functions, $\tilde{\kappa}$ and $\tilde{\zeta}$ can be understood as first-order languages for understanding exceptions and handlers. By interpreting in our running cocc category $\mathcal{C}_K$, a covalue $c : C \rightsquigarrow 0$ is a $C \to R^2 0 \cong C \to R$ – a continuation, and these first-order operations bind and apply covalues on arrows, changing control flow. The type $A - C$ is interpreted as a program of type A with a handler for C attached, and $C + B$ is a program of type B program that could throw a C. The operators themselves do not produce any control effects.

As an example, suppose we have a sequential program:

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D \xrightarrow{e} E$$

We decide to inspect the program at C, so we insert a code pointer (or breakpoint):

$$A \xrightarrow{f} B \xrightarrow{\widetilde{\zeta z^Z}.g} Z + C \xrightarrow{h^Z} Z + D \xrightarrow{\widetilde{\mathsf{pass}}_D(z)} D \xrightarrow{e} E$$

The program h^Z could then use the Z handler to do something interesting – inspect the program's state at that point, modify it, or return a Z value, skipping e and escaping. This mechanism can be used as a basis for debugging or checkpoints.

$$\frac{\begin{array}{c}[x : C \rightsquigarrow \mathbf{0}] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\widetilde{\kappa} x^C.f : (A - C) \rightsquigarrow B}\ -\mathrm{L} \qquad \frac{\begin{array}{c}[x : C \rightsquigarrow \mathbf{0}] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\widetilde{\kappa} x^C.f \circ \widetilde{\mathsf{lift}}_A(c) \equiv f[c/x] : A \rightsquigarrow B}\ \tilde{\kappa}^+$$

$$\frac{c : C \rightsquigarrow \mathbf{0}}{\widetilde{\mathsf{lift}}_A(c) : A \rightsquigarrow (A - C)}\ -\mathrm{R} \qquad \frac{h : (A - C) \rightsquigarrow B}{\widetilde{\kappa} x^C.(h \circ \widetilde{\mathsf{lift}}_A(x)) \equiv h : (A - C) \rightsquigarrow B}\ \kappa^-$$

(a) $\tilde{\kappa}$ calculus

$$\frac{c : C \rightsquigarrow \mathbf{0}}{\widetilde{\mathsf{pass}}_B(c) : (C + B) \rightsquigarrow B}\ +\mathrm{L} \qquad \frac{h : A \rightsquigarrow (C + B)}{\widetilde{\zeta} x^C.(\widetilde{\mathsf{pass}}_B(x) \circ h) \equiv h : A \rightsquigarrow (C + B)}\ \tilde{\zeta}^-$$

$$\frac{\begin{array}{c}[x : C \rightsquigarrow \mathbf{0}] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\widetilde{\zeta} x^C.f : A \rightsquigarrow (C + B)}\ +\mathrm{R} \qquad \frac{\begin{array}{c}[x : C \rightsquigarrow \mathbf{0}] \\ \vdots \\ f : A \rightsquigarrow B\end{array}}{\widetilde{\mathsf{pass}}_B(c) \circ (\widetilde{\zeta} x^C.f) \equiv f[c/x] : A \rightsquigarrow B}\ \tilde{\zeta}^+$$

(b) $\tilde{\zeta}$ calculus

Fig. 12. $\tilde{\kappa}$ and $\tilde{\zeta}$ calculi

5.3 Logical Aspects

The Curry-Howard correspondence with classical logic is established by the following theorem.

Theorem 17 (Curry-Howard correspondence). *The following are equivalent:*

(1) $\Gamma \vdash \Delta, A$ *is provable in Gentzen [1935]'s LK (without* $\bot$*).*
(2) $\Gamma \vdash \Delta, A$ *is provable in Crolard [2001]'s subtractive logic + TND.*
(3) There is a typable term $\Gamma \vdash e : \Delta + A$ *in* $\lambda\tilde{\lambda}$ *(where* Δ *is interpreted as sums of propositions).*

The use of the dual type of covalues (or continuations) can be seen as a capability – capturing a continuation variable allows the program to perform control effects, which means disallowing continuation variables in contexts would be a way to recover pure computation (or intuitionistic logic), using the framework of purity comonads [Choudhury and Krishnaswami 2020].

$$\begin{array}{lcl}(\Gamma, x : \tilde{A})^{\mathsf{p}} & \triangleq & \Gamma^{\mathsf{p}} \\ (\Gamma, x : A)^{\mathsf{p}} & \triangleq & \Gamma^{\mathsf{p}}, x : A\end{array}$$

A boxed sum $\Box(A + B)$ becomes a pure sum, and must be either $\Box A$ or $\Box B$.

6 Examples and Implementation

We illustrate the programming features of the $\lambda\tilde{\lambda}$ calculus by writing programs in a hypothetical language, whose syntax is similar to a typed functional programming language (like ML or Haskell), extended with cofunctions. These examples are all implemented in the supplementary material, and can be interactively followed along with the text.

Functions and Cofunctions. The language has values and covalues, and functions and cofunctions. Functions `fn` bind values, and cofunctions `cofn` bind covalues. Covalues have `co` types, and `cofn` binds a covalue, producing a cofunction which has (surprisingly!) a sum type, written as `a + b`. Coapplication is written as `f @ k`, which supplies a covalue to a sum type.

```
fun ex₁ (f : int → str) (g : int + str) : int → int + str =
  fn (x : int) ⇒
    cofn (k : co int) ⇒
      if x = 0 then g @ k else f x
```

The program `ex₁` take two arguments: a function `f : int → str`, and a sum (or cofunction) `g : int + str`, and returns something of type `int → int + str`. The body of the program first introduces a lambda using `fn (x : int)`, that binds a value `x : int` creating a function whose domain is an `int`. The body of the program needs to produce something of type `int + str`. This is introduced by a colambda `cofn (k : co int)` which binds a covalue `k : co int`, creating a cofunction. The value `x` is used in the body by applying the function `f`, and the covalue `k` is used in the body by applying the sum `g`. Here are two sample executions of the program `ex₁`:

```
     ex₁ Int.toStr (INR "0") 0
⇝    cofn (k : co int) ⇒
       if 0 = 0
         then (INR "0") @ k
         else Int.toStr 0
⇝    cofn (k : co int) ⇒ (INR "0") @ k
⇝    INR "0"
```

```
     ex₁ Int.toStr (INL 1) 1
⇝    cofn (k : co int) ⇒
       if 1 = 0
         then (INL 1) @ k
         else Int.toStr 1
⇝    cofn (k : co int) ⇒ Int.toStr 1
⇝    INR "1"
```

The standard rules for capture-avoiding substitutions apply to both functions and cofunctions, which we perform implicitly. On the left, the body of the inner cofunction reduces by following the left branch of the conditional, which produces the cofunction `cofn (k : co int) ⇒ (INR "0") @ k`. This reduces by eta-conversion to the value `INR "0"`. Dually, on the right, the body of the inner cofunction reduces by following the right branch of the conditional, which produces the cofunction `cofn (k : co int) ⇒ Int.toStr 1`. The body of this cofunction doesn't use the covalue `k`, causing it to collapse, producing `INR "1"`.

This example could've been written without any of this technology, just using standard sums:

```
fun ex₁ (f : int → str) (g : int + str) : int → int + str =
  fn (x : int) ⇒
    if x = 0 then g else INR (f x)
```

What is then the point of cofunctions? `INL`/`INR` produce ordinary sums, but cofunctions produce "Faustian" sums [Wadler 2003], which have control effects.

Exceptional Cofunctions. Consider a simple program which multiplies the elements in a list (taken from [Harper et al. 1993]):

```
fun mult (l : list int) : int =
  let fun loop [] = 1
        | loop (h :: t) = h * loop t
  in loop l
end
```

If `l` contains a `0` anywhere in the list, the program will always return `0`, but performing several vacuous multiplications along the way.

```
mult [1, 2, 0, 3, 4]  ⇝  1 * (2 * (0 * (3 * (4 * 1))))
                      ⇝  1 * (2 * (0 * (3 * 4)))
                      ⇝  1 * (2 * (0 * 12))
                      ⇝  1 * (2 * 0)
                      ⇝  1 * 0
                      ⇝  0
```

A naive way to avoid vacuous multiplications is to stop computing as soon as we see a `0`.

```
fun mult (l : list int) : int =
  let fun loop [] = 1
        | loop (0 :: _) = 0
        | loop (h :: t) = h * loop t
  in loop l
end
```

This proceeds as:

```
mult [1, 2, 0, 3, 4]  ⇝  1 * (2 * 0)
                      ⇝  1 * 0
                      ⇝  0
```

This avoids traversing the list once it notices a `0`, but still vacuously multiplies by `0` as it finishes the rest of the computation. Ideally, we want to avoid multiplying once we see a `0`, and treat it as an exceptional value.

As type theorists, we know about sum types, and we can use them to model two branches of computation – a value on the right is a normal value, and a value on the left is an exceptional value. We do multiplications on the right, but we return a `0` on the left. To understand the behavior of the program, we additionally print a trace as we're computing, which shows the evaluation context at each step. The `trace` function takes a computation as an argument, evaluates it (following call-by-value right-to-left convention), prints a trace, and returns the result.

```
fun trace (s : string) (x : a) : a = print (s ^ "\n") ; x

fun mult (l : list int) : int + int =
  let fun loop [] = INR 1
        | loop (0 :: _) = INL 0
        | loop (h :: t) = trace ("at " ^ Int.toStr h)
                                (mapRight (fn x ⇒ h * x) (loop t))
  in loop l
end
```

This computes as:

```
      mult [1, 2, 0, 3, 4]
⇝  mapRight (fn x ⇒ 1 * x) (mapRight (fn x ⇒ 2 * x) (loop [0, 3, 4]))
⇝  mapRight (fn x ⇒ 1 * x) (mapRight (fn x ⇒ 2 * x) (INL 0))
⇝  mapRight (fn x ⇒ 1 * x) (INL 0)
⇝  INL 0
```

printing the trace: `"at 2"`, then `"at 1"`.

This encoding using sums is almost the behavior we want, which avoids vacuous multiplications, but still traverses up to the top of the list once it hits a `0`, printing the trace. What we really want is to short-circuit the computation, abandoning the computation in the right branch, and jumping to the left branch with a `0`. We can do this using the jumping behaviour of cofunctions.

```
fun mult (l : list int) : int + int =
```

```
  cofn (k : co int) ⇒
    let fun loop [] = 1
          | loop (0 :: _) = (INL 0) @ k
          | loop (h :: t) = trace ("at " ^ Int.toStr h) (h * loop t)
    in loop l
  end
```

`cofn` binds a covalue `k : co int`, and (speculatively) executes its body, assuming that it is computing the right branch of the sum. The bound covalue `k` allows one to backtrack and "jump with an (`int`) argument" to the left branch. The `loop` function is the same as before, except when it hits a `0`, it coapplies `INL 0` to the bound covalue `k`, jumping to the left branch and exiting the program. This computes as:

```
mult [1, 2, 0, 3, 4]  ⇝  cofn (k : co int) ⇒ loop [1, 2, 0, 3, 4]
                      ⇝  cofn (k : co int) ⇒ 1 * loop [2, 0, 3, 4]
                      ⇝  cofn (k : co int) ⇒ 1 * (2 * loop [0, 3, 4])
                      ⇝  cofn (k : co int) ⇒ 1 * (2 * (INL 0) @ k)
                      ⇝  INL 0
```

and prints no trace!

Algebra of Cofunctions. Constant functions, like `fn (x : int) ⇒ 0`, when applied to any argument, will always return the value `0`. Similarly, we have constant cofunctions, like `cofn (k : co int) ⇒ 0`, which are right-biased sums – but unlike functions (lambdas), they aren't frozen thunks – for example, constant cofunctions collapse and reduce to an ordinary (right) sum.

```
cofn (k : co a) ⇒ b   ⇝   INR b
```

The identity cofunction, which returns the covalue it binds, produces a choice between a value and a covalue – this is Tertium Non Datur (or the Law of the Excluded Third).

```
fun tnd () : a + co a = cofn (k : co a) ⇒ k
```

The `tnd` cofunction doesn't reduce on its own – not until you observe it by pattern matching (see § 4.2). The «subtraction» type `a - b` is the "proper" dual of the function type `a → b`, and is defined as a product of a value and a covalue:

```
type a - b = a * co b
```

Note that the subtraction type is derived from covalues and not included as a primitive type in our language. The duality is witnessed by currying and cocurrying – our motivating example, which can be implemented using our cofunction operations (also see `ftoc` and `ctof`):

```
fun curry (f : (c * a) → b) : c → (a → b)   = fn x ⇒ fn y ⇒ f (x, y)
fun uncurry (f : c → (a → b)) : (c * a) → b = fn (x, y) ⇒ f x y
fun cocurry (f : c → a + b) : (c - a) → b   = fn (c, k) ⇒ (f c) @ k
fun councurry (f : (c - a) → b) : c → a + b = fn c ⇒ cofn k ⇒ f (c, k)
```

Subtraction and sums interplay in the following way:

```
fun coeval (x : a) : b + (a - b)   = cofn k ⇒ (x, k)
fun couneval (f : (b + a) - b) : a = (#1 f) @ (#2 f)
```

The type of `coeval` can be seen as a generalised version of `tnd`. Since cofunctions are just sums, they have the familiar case construct for pattern matching. Using case, we can do operations on both sides of the sum, for example, we can define a cocomposition of subtractive types:

```
fun cocompose (f : a - c) : (a - b) + (b - c) =
```

```
  case coeval (#1 f) of
      INL b ⇒ INR (b, #2 f)
    | INR (a, k) ⇒ INL (a, k)
```

Value-Covalue Interaction. We could produce a choice between a value and a covalue out of nothing, on both sides of the sum – but what if we had access to a value and covalue at the same time, i.e., `a - a`? We can `throw` – lifting the value to the left, then coapplying the covalue, producing `b` out of nothing.

```
fun throw (p : a - a) : b = (INL (#1 p)) @ (#2 p)
```

Dually, the sum type `a + a` can be collapsed to an `a`:

```
fun codiag (s : a + a) : a = case s of INL a ⇒ a | INR a ⇒ a
```

Note also that sums are symmetric, and can be swapped:

```
fun swap (s : a + b) : b + a = case s of INL a ⇒ INR a | INR b ⇒ INL b
```

To a continuations afficionado, these control operators will look familiar. `callcc` is SML's native control operator, which reifies the evaluation context as a continuation, and `call/cc` is Peirce's law.

```
fun callcc (f : co a → a) : a =
  codiag (cofn (k : co a) ⇒ f k)
```

```
fun call/cc (f : (a → b) → a) : a =
  codiag (cofn (k : co a) ⇒
    f (fn a ⇒ throw (a, k)))
```

This definition highlights the duplicating nature of `callcc` (pointed out by Filinski [1989]), which collapses both sides of the sum using `codiag` – the use of sums and `case` makes this explicit.

To show the computational interpretation of classical logic, we use the `co` type as the primitive negation type (instead of a function `a → 0`). Exploiting `tnd` (at types `co a` and `a` respectively), and `case`, we can perform double negation introduction and elimination:

```
fun dni (x : a) : co (co a) =
  case tnd () of
      INL k_a ⇒ throw (x, k_a)
    | INR k_a² ⇒ k_a²
```

```
fun dne (k_a² : co (co a)) : a =
  case tnd () of
      INL a ⇒ a
    | INR k_a ⇒ throw (k_a, k_a²)
```

The self-adjointness of negation is fundamental, and is captured by the symmetry of sums:

```
fun adj (f : co a → b) : co b → a = fn (k_b : co b) ⇒
  (swap (cofn (k_a : co a) ⇒ f k_a)) @ k_b
```

Using this, we get the contravariance of negation:

```
fun contramap (f : a → b) : co b → co a = adj (fn k_a² ⇒ f (dne k_a²))
```

The classical encoding of functions as material implication is obtained as follows:

```
fun lam (f : a → b) : co a + b = cofn k_a² ⇒ f (dne k_a²)
fun app (e : co a + b) : a → b = fn a ⇒ e @ dni a
```

de Morgan's laws are obtained as follows:

```
fun deMorgan₁ (k : co (a + b)) : (co a * co b) = (contramap INL k, contramap INR k)
fun deMorgan₂ ((k_a , k_b) : co a * co b) : co (a + b) = contramap (fn e ⇒ e @ k_a) k_b
```

```
fun deMorgan₃ (kₚ : co (a * b)) : co a + co b =
  adj (fn kₛ ⇒ let val (k²ₐ, k²ᵦ) = deMorgan₁ kₛ in (dne k²ₐ, dne k²ᵦ) end) kₚ
fun deMorgan₄ (kₛ : co a + co b) : co (a * b) =
  case tnd () of
    INL (a, b) ⇒ (case kₛ of INL kₐ ⇒ throw (a, kₐ) | INR kᵦ ⇒ throw (b, kᵦ))
  | INR kₚ ⇒ kₚ
```

The subtraction type is the formal dual of the function type. We show this by defining functions `ftoc` and `ctof` (named after the operators in [Führmann and Thielecke 2004, § 3]).

```
fun ftoc (f : a → b) : co (a - b) =
  case lam f of
    INL kₐ ⇒ deMorgan₄ (INL kₐ)
  | INR b ⇒ deMorgan₄ (INR (dni b))
```

```
fun ctof (k : co (a - b)) : a → b =
  fn a ⇒ case deMorgan₃ k of
            INL kₐ ⇒ throw (a, kₐ)
          | INR k²ᵦ ⇒ dne k²ᵦ
```

Dummett's linearity axiom is derived as follows:

```
fun dummett () : (a → b) + (b → a) =
  case tnd () of
    INL b ⇒ INL (fn a ⇒ b)
  | INR kᵦ ⇒ INR (fn b ⇒ throw (b, kᵦ))
```

Implementing $\tilde{\lambda}$. Rather than programming in a hypothetical language, we show how $\lambda\tilde{\lambda}$ can be implemented and retrofit into an existing programming language. Unlike other dual calculi (like $\lambda\mu$ or $\mu\tilde{\mu}$) which change the judgemental structure of the language, our dual calculus readily adapts to an existing functional language, because $\tilde{\lambda}$ can be thought of as a control operator, and used as an interface to a native continuation type.
Conor McBride once said:

> Moggi [1989] taught the dog how to bark, Wadler [1993] made us bark ourselves.

If we have native continuations and control operators (like in SML), we can use them to implement $\tilde{\lambda}$, or, if we have a continuation monad (like in Haskell), we can program in the monad using co-exponential combinators. We describe both implementations, and these are further developed in the supplementary material.

SML. The native continuation type comes with control operators `callcc`/`throw`. Using them, we encode the generalised control operators `colam`/`coapp`.

```
signature COEXP =
sig
  type 'a dual
  val colam : ('a dual → 'b) → ('a, 'b) either
  val coapp : ('a, 'b) either → 'a dual → 'b
end
structure Coexp: COEXP =
struct
  type 'a dual = 'a cont
  fun colam (f : 'a dual → 'b) : ('a, 'b) either =
    callcc (fn (k : ('a, 'b) either dual) ⇒
      let val a = callcc (fn (kₐ : 'a dual) ⇒ throw k (INR (f kₐ)))
```

```
      in throw k (INL a)
      end)
  fun coapp (e : ('a, 'b) either) (k : 'a dual) : 'b =
    case e of
      INL a ⇒ throw k a
    | INR b ⇒ b
end
```

Note, encoding `colam` requires *two* uses of `callcc` to grab both continuations. This programming pattern is commonly used in backtracking with multiple continuations, and `colam` leads to more readable code than using `callcc` directly. From `colam`/`coapp`, we recover `callcc`/`throw`, which satisfy the right equations (see Proposition 10).

```
fun codiag (e : ('a, 'a) either) : 'a =
  case e of
    INL a ⇒ a
  | INR a ⇒ a
fun callcc (f : 'a cont → 'a) : 'a = codiag (colam f)
fun throw (a : 'a) (k : 'a cont) : 'b = coapp (INL a) k
```

It can be verified, assuming that the native `callcc`/`throw` satisfy the right laws (as in [Sabry and Felleisen 1993]), that the derived operators satisfy the right equational theory.

Haskell. We program directly in the continuation monad `Cont r`, for a polymorphic result type `r`.

```
colam :: ((a → r) → Cont r b) → Cont r (a + b)
colam f = cont $ \k →
  let kₐ = k . Left
      k_b = k . Right
   in runCont (f kₐ) k_b

coapp :: (a + b) → (a → r) → Cont r b
coapp e kₐ = cont $ \k_b →
  case e of
    Left a → kₐ a
    Right b → k_b b
```

Backtracking Combinators. These co-exponential combinators are useful for programming with two continuations (double-barrelled cps), which is a common style for backtracking, with a success and a failure continuation. The (classical) `assumeRight` combinator is like the (intuitionistic) `Right` constructor, but with an additional assumption for the left continuation. The opposite of `assumeRight` is `resolveRight`, which coapplies the left continuation to the cofunction.

```
assumeRight :: ((a → r) → Cont r b) → Cont r (a + b)
assumeRight = colam

resolveRight :: (a + b) → (a → r) → Cont r b
resolveRight = coapp
```

The symmetric cofunction type allows us to swap choices, which is equivalent to going back and making a different choice, producing the dual `assumeLeft`/`resolveLeft` combinators.

```
swap :: a + b → b + a
```

```
swap = either Right Left

assumeLeft :: ((b → r) → Cont r a) → Cont r (a + b)
assumeLeft = fmap swap . colam

resolveLeft :: (a + b) → (b → r) → Cont r a
resolveLeft = coapp . swap
```

We need two more combinators, which allow us to use the result type, by running both continuations. This allows us to manipulate the backtracking state.

```
assumeBoth :: ((a → r) → (b → r) → r) → Cont r (a + b)
assumeBoth = assumeRight . (cont .)

resolveBoth :: Cont r (a + b) → (a → r) → (b → r) → r
resolveBoth m k = runCont (m >>= (`resolveRight` k))
```

Using these combinators, we program a backtracking SAT solver (and a backtracking tree search, in the supplementary material).

SAT Solver. This example is based on Errington [2024]'s SAT solver. We fix a simple propositional logic, with atoms, conjunctions, disjunctions, and negations (cf. Hedges [2014]'s SAT solver, which directly uses Boolean functions). The state of the solver is either a success or a failure: a success state contains a current assignment of variables, a boolean result tracking the satisfiability of the current formula, and the result (toplevel continuation) of the rest of the computation, or, a failure state, which is empty. Note that the result type `r` needs to be used in the `Succ r` state to track the result of the computation.

```
data Prop = PAtom String | ⊥ | ⊤ | Prop ∨ Prop | Prop ∧ Prop | ¬ Prop

type Succ r = (Env Bool, Bool, () → r)
type Fail = ()
```

The programming pattern with `assume/resolve` combinators is to assume one side of the sum, getting access to the continuation for the other side, then do some new computation, but resolving it with a new continuation, which short-circuits the solver if an assignment is impossible. The simplest cases are $\bot$ and $\top$, where we can immediately choose left or right – the `assumeRight` combinator gives access to the toplevel continuation which is returned as part of the success state.

```
solve :: Env Bool → Prop → Cont r (Fail + Succ r)
solve env ⊥ = assumeLeft $ \_ → return ()
solve env ⊤ = assumeRight $ \kfail → return (env, True, kfail)
```

To solve for $\neg\ \varphi$, we assume left (failure), getting access to the success continuation. Then, we solve for the formula φ and resolve the computation with a new success continuation, which simply negates the boolean result.

```
solve env (¬ φ) = assumeLeft $ \ksucc → do
  s ← solve env φ
  resolveLeft s $ \(env, b, kfail) → ksucc (env, not b, kfail)
```

The case for disjunction starts by assuming left (failure), getting access to the success continuation. We solve for the first formula φ_1, but we resolve this computation by plugging in a new success continuation. This continuation tests the result of the solver – if a satisfying assignment is found,

we can short-circuit the computation by calling the success continuation, otherwise, we have to continue solving for the second formula φ_2. We resolve this computation on both sides – the left side (failure) plugs in the toplevel continuation, the right side (success) plugs in a new success continuation, updated with the new boolean result. The case for conjunction is symmetric.

```
solve env (φ₁ ∨ φ₂) = assumeLeft $ \kₛᵤᶜᶜ → do
  s₁ ← solve env φ₁
  resolveLeft s₁ $ \(env, b₁, kₛᵤᶜᶜ) →
    if b₁ then kₛᵤᶜᶜ (env, True, kₛᵤᶜᶜ)
          else resolveBoth (solve env φ₂) kₛᵤᶜᶜ $
                 \(env, b₂, kₛᵤᶜᶜ) → kₛᵤᶜᶜ (env, b₁ || b₂, kₛᵤᶜᶜ)
```

Finally, to solve for atoms, we look up the atom in the environment – if it is already assigned, we are done, otherwise, we assume continuations for both sides, and solve for both assignments by calling the success continuation with two new assignments.

```
solve env (PAtom x) =
  case lookupEnv x env of
    Just b → assumeRight $ \kₛᵤᶜᶜ → return (env, b, kₛᵤᶜᶜ)
    Nothing → assumeBoth $ \kₛᵤᶜᶜ kₛᵤᶜᶜ →
                kₛᵤᶜᶜ (insertEnv x True env, True, \() →
                  kₛᵤᶜᶜ (insertEnv x False env, False, kₛᵤᶜᶜ))
```

The full implementation is given in the supplementary material, and also benchmarks this solver against a brute force solver written using `shift`/`reset`.

Effect Handlers. Effect handlers are a natural example for managing stacks of continuations – the handler algebra `f r → r`, and the generator `a → r`, where `f` is the signature functor. Using a Church-encoded `Free` monad (or codensity monad), and using the co-exponential combinators, we can derive the universal property of the `Free` monad, which can be used to implement a library of effect handlers (also in the supplementary material).

```
newtype Free f a = Free {runFree :: ∀ r. (f r → r) → Cont r a}

colamFree :: Free f a → Cont r (f r + a)
colamFree f = colam $ \alg → cont $ \gen → runCont (runFree f alg) gen

foldFree :: Functor f ⇒ (f r → r) → (a → r) → Free f a → r
foldFree alg gen = reset0 . fmap (either gen alg) . colamFree
```

Note, both of these Haskell implementations use the result type `r` inside the program. This does not denote a term in the $\lambda\tilde{\lambda}$ calculus, but only in the Haskell encoding of the co-exponential combinators.

Interpreter. A bidirectional typechecker, and interpreter for $\lambda\tilde{\lambda}$, in the style of an abstract machine, is given in the supplementary material. It evaluates well-typed $\lambda\tilde{\lambda}$ terms by directing the beta equations and control effect equations, producing an operational semantics.

7 Discussion

The theme of this work is the duality of currying and cocurrying, which is used to produce a duality of abstraction – for values and covalues. This is a useful perspective showing that values and continuations are dual to each other and should have the same ontological status. Continuations

producing left adjoints to sums provides insights into the computational behavior of classical disjunction, and control flow.

The semantics of continuations are well-known and have been extensively studied, and our work builds it from a different viewpoint, starting from the primitive notion of cartesian closure and cocartesian coclosure. In more formal programming jargon, the $\lambda\tilde{\lambda}$ calculus is about the duality of cbv functions and cbn cofunctions. Filinski [1989] mentions coexponentials might be understood using the Kleisli category of the continuation monad – which we have exploited here. Griffin [1989] mentions the classical logic interpretation of control operations might have something to do with Filinski's duality – we have shown that this is indeed the case. Hofmann [1995] axiomatizes continuations using the empty type, and suggests finding another axiomatization using a native continuation type.

The purpose of this work is not simply to solve the puzzle of avoiding Joyal's lemma, but to also highlight the fact that a new axiomatization of the semantics sheds light into the nature of evaluation of programs and control operators – in this case, speculative execution and backtracking, and how this can be distilled into programming language design. The design of the language was semantically motivated, and we made a few stylistic choices in presenting it. Not including the $\mathbf{0}$ type, for example, is a stylistic choice, and $\tilde{\mathbf{1}}$ works instead (for typing). We prefer to abort using sums, by inl and coapplication, which is semantically equivalent. Another approach is to add a judgement for non-returning programs, like in Zeilberger [2009]'s cbv CPS calculus, or Levy [2003]'s JwA. We conflated value sums and computational sums to highlight the duality of abstraction and coabstraction, but this language could also be presented in a fine-grained call-by-value style.

The modern view of dualities of computation is in polarised adjunction calculi, and their categorical models [Curien, Fiore, et al. 2016]. The duality exploited here shows up in their cartesian polarised structure theorems (Ex. 27). The biclosed action models of Fiore [2013] generalise the exponentiating (response) object to copowers.

In general, dual calculi for values and covalues take both cbv and cbn points of view, but ours restricts to the cbv case, which makes it possible to have the dual abstraction mechanisms in the language. This can be understood by looking at the cbv variants of $\lambda\mu$ and $\mu\tilde{\mu}$. Selinger [2001, 2003] presents different extensions of cbv $\lambda\mu$ with μ binding two variables, or a ν binder, which uses the coexponential interpretation. Other presentations of $\lambda\mu$ in cbv, such as the one by Ong and Stewart [1997], do not directly exhibit this structure. The cbv translation of $\mu\tilde{\mu}$ in Curien and Herbelin [2000] uses the subtraction type, but they do not develop an equational theory. We have not studied a reduction semantics for our language – the $\lambda\mu$ calculus fails Böhm's separation theorem for cbn, and has to be extended with eta rules [Saurin 2008], but the status for cbv is unknown, and similarly for our calculus.

The crucial difference in our calculus is that we stick to a type-theoretic, one-sided sequent, natural deduction style presentation (multicategorical) – the closest cousins are Levy's JwA and Zeilberger's cbv cps calculus (which use non-returning judgements), or Harper et al. [1993]'s `callcc` in SML. Ariola et al. [2009] present a type theory with a subtraction type, but we do not add it as a primitive. Instead, we have subtraction in our dual arrow calculi.

This work was inspired by the semantic understanding of dualities in session types and classical linear logic, in particular, while trying to understand the axiomatics of $(-)^*$ in star-autonomous categories. The $(-)^*$ operator is an involution, but $R^{(-)}$ of continuations is not. But, they both exhibit a function/cofunction duality – star-autonomous categories have $\multimap$ and $\circ\!\!-$. This is developed in the work of Melliès [2017] on dialogue categories, studying the axiomatics of negation motivated by game semantics, which inspired our analysis. The composable continuations "monad" [Atkey 2009], which is not really a monad, has a similar adjunction: $C \times S^A \to R^{S^B} \cong C \to R^{S^{A+B}}$.

Data-Availability Statement

The accompanying implementation and a partial formalisation for the paper is available at the following GitHub repository: vikraman/popl25-duality-artifact [Choudhury 2024].

Acknowledgments

Vikraman is supported by the EU Marie-Skłodowska-Curie action ReGraDe-CS, grant № 101106046 (https://doi.org/10.3030/101106046). We thank the anonymous reviewers for their detailed comments and suggestions for improvements, which significantly improved the presentation of this work. Thanks are due to Nathanael Arkor, Dylan McDermott, Hugo Paquet, Philip Saville, Neel Krishnaswami, Bob Atkey, Sam Lindley, Ohad Kammar, Marcelo Fiore, Eugenio Moggi, Tim Griffin, Peter Selinger, for various comments and discussions. Several discussions with Amr Sabry and Borislav Agapiev were instrumental in the early stages of this work.

References

Gerhard Gentzen. Dec. 1, 1935. "Untersuchungen über das logische Schließen. I." *Mathematische Zeitschrift*, 39, 1, (Dec. 1, 1935), 176–210. DOI: https://doi.org/10.1007/BF01201353 (cit. on p. 18).

J. Lambek. Mar. 1, 1974. "Functional Completeness of Cartesian Categories." *Annals of Mathematical Logic*, 6, 3, (Mar. 1, 1974), 259–292. DOI: https://doi.org/10.1016/0003-4843(74)90003-5 (cit. on pp. 15, 16).

R. M. Burstall. 1980. "Electronic Category Theory." In: *Mathematical Foundations of Computer Science 1980*. Vol. 88. Ed. by P. Dembiński. Springer-Verlag, Berlin/Heidelberg, 22–39. ISBN: 978-3-540-10027-0. DOI: https://doi.org/10.1007/BFb0022493 (cit. on p. 1).

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1980. "HOPE: An Experimental Applicative Language." In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming - LFP '80*. The 1980 ACM Conference. ACM Press, Stanford University, California, United States, 136–143. DOI: https://doi.org/10.1145/800087.802799 (cit. on p. 1).

J. Lambek and P. J. Scott. Mar. 25, 1988. *Introduction to Higher-Order Categorical Logic*. Cambridge University Press, (Mar. 25, 1988). 308 pp. ISBN: 978-0-521-35653-4. Google Books: 6PY_emBeGjUC (cit. on p. 2).

C.-H. Luke Ong. 1988. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. University of London, Great Britain. 1 p. Retrieved Dec. 2, 2024 from http://hdl.handle.net/10044/1/47211 (cit. on p. 1).

Andrzej Filinski. 1989. "Declarative Continuations: An Investigation of Duality in Programming Language Semantics." In: *Category Theory and Computer Science*. Vol. 389. Ed. by David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné. Springer-Verlag, Berlin/Heidelberg, 224–249. ISBN: 978-3-540-51662-0. DOI: https://doi.org/10.1007/BFb0018355 (cit. on pp. 1, 3, 22, 27).

Timothy G. Griffin. Dec. 1, 1989. "A Formulae-as-Type Notion of Control." In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '90). Association for Computing Machinery, New York, NY, USA, (Dec. 1, 1989), 47–58. ISBN: 978-0-89791-343-0. DOI: https://doi.org/10.1145/96709.96714 (cit. on pp. 3, 27).

E. Moggi. June 1989. "Computational Lambda-Calculus and Monads." In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. (June 1989), 14–23. DOI: https://doi.org/10.1109/LICS.1989.39155 (cit. on pp. 9, 10, 23).

Jean-Yves Girard. Nov. 1991. "A New Constructive Logic: Classic Logic." *Mathematical Structures in Computer Science*, 1, 3, (Nov. 1991), 255–296. DOI: https://doi.org/10.1017/S0960129500001328 (cit. on p. 1).

Michel Parigot. 1992. "$\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction." In: *Logic Programming and Automated Reasoning* (Lecture Notes in Computer Science). Ed. by Andrei Voronkov. Springer, Berlin, Heidelberg, 190–201. ISBN: 978-3-540-47279-7. DOI: https://doi.org/10.1007/BFb0013061 (cit. on pp. 1, 3).

Samson Abramsky and C.-H. Luke Ong. Aug. 1993. "Full Abstraction in the Lazy Lambda Calculus." *Information and Computation*, 105, 2, (Aug. 1993), 159–267. DOI: https://doi.org/10.1006/inco.1993.1044 (cit. on p. 1).

Robert Harper, Bruce F. Duba, and David Macqueen. Oct. 1993. "Typing First-Class Continuations in ML." *Journal of Functional Programming*, 3, 4, (Oct. 1993), 465–484. DOI: https://doi.org/10.1017/S095679680000085X (cit. on pp. 19, 27).

Claudio Alberto Hermida. Nov. 1, 1993. "Fibrations, Logical Predicates and Indeterminates." *DAIMI Report Series*, 22, 462, (Nov. 1, 1993). DOI: https://doi.org/10.7146/dpb.v22i462.6935 (cit. on p. 15).

Kohei Honda. 1993. "Types for Dyadic Interaction." In: *CONCUR'93* (Lecture Notes in Computer Science). Ed. by Eike Best. Springer, Berlin, Heidelberg, 509–523. ISBN: 978-3-540-47968-0. DOI: https://doi.org/10.1007/3-540-57208-2_35 (cit. on p. 1).

Amr Sabry and Matthias Felleisen. Nov. 1, 1993. "Reasoning about Programs in Continuation-Passing Style." *LISP and Symbolic Computation*, 6, 3, (Nov. 1, 1993), 289–360. DOI: https://doi.org/10.1007/BF01019462 (cit. on p. 24).

Philip Wadler. 1993. "Monads for Functional Programming." In: *Program Design Calculi* (NATO ASI Series). Ed. by Manfred Broy. Springer, Berlin, Heidelberg, 233–264. ISBN: 978-3-662-02880-3. DOI: https://doi.org/10.1007/978-3-662-02880-3_8 (cit. on p. 23).

Masahito Hasegawa. 1995. "Decomposing Typed Lambda Calculus into a Couple of Categorical Programming Languages." In: *Category Theory and Computer Science* (Lecture Notes in Computer Science). Ed. by David Pitt, David E. Rydeheard, and Peter Johnstone. Springer, Berlin, Heidelberg, 200–219. ISBN: 978-3-540-44661-3. DOI: https://doi.org/10.1007/3-540-60164-3_28 (cit. on p. 15).

Martin Hofmann. Dec. 1995. "Sound and Complete Axiomatisations of Call-by-Value Control Operators." *Mathematical Structures in Computer Science*, 5, 4, (Dec. 1995), 461–482. DOI: https://doi.org/10.1017/S0960129500001195 (cit. on pp. 3, 8, 13, 14, 27, 32).

Martin Hofmann and Thomas Streicher. 1997. "Continuation Models Are Universal for λ/Sub μ/-Calculus." In: *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. Twelfth Annual IEEE Symposium on Logic in Computer Science. IEEE Comput. Soc, Warsaw, Poland, 387–395. ISBN: 978-0-8186-7925-4. DOI: https://doi.org/10.1109/LICS.1997.614964 (cit. on pp. 14, 32).

C.-H. Luke Ong and Charles A. Stewart. Jan. 1, 1997. "A Curry-Howard Foundation for Functional Computation with Control." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '97). Association for Computing Machinery, New York, NY, USA, (Jan. 1, 1997), 215–227. ISBN: 978-0-89791-853-4. DOI: https://doi.org/10.1145/263699.263722 (cit. on p. 27).

Hayo Thielecke. 1997. "Categorical Structure of Continuation Passing Style." The University of Edinburgh. Retrieved Jan. 6, 2023 from https://era.ed.ac.uk/handle/1842/14533 (cit. on pp. 3, 8).

Thomas Streicher and Bernhard Reus. Nov. 1998. "Classical Logic, Continuation Semantics and Abstract Machines." *Journal of Functional Programming*, 8, 6, (Nov. 1998), 543–572. DOI: https://doi.org/10.1017/S0956796898003141 (cit. on pp. 3, 7, 8).

Hayo Thielecke. June 1999. "Continuations, Functions and Jumps." *ACM SIGACT News*, 30, 2, (June 1999), 33–42. DOI: https://doi.org/10.1145/568547.568561 (cit. on p. 14).

Pierre-Louis Curien and Hugo Herbelin. Sept. 1, 2000. "The Duality of Computation." *ACM SIGPLAN Notices*, 35, 9, (Sept. 1, 2000), 233–243. DOI: https://doi.org/10.1145/357766.351262 (cit. on pp. 1, 3, 27).

Tristan Crolard. Mar. 2001. "Subtractive Logic." *Theoretical Computer Science*, 254, 1-2, (Mar. 2001), 151–185. DOI: https://doi.org/10.1016/S0304-3975(99)00124-3 (cit. on pp. 2, 18).

Jean-Yves Girard. June 2001. "Locus Solum: From the Rules of Logic to the Logic of Rules." *Mathematical Structures in Computer Science*, 11, 3, (June 2001), 301–506. DOI: https://doi.org/10.1017/S096012950100336X (cit. on p. 4).

Paul Blain Levy. 2001. "Call-by-Push-Value." Ph.D. Dissertation. Queen Mary, University of London. Retrieved June 6, 2023 from https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233 (cit. on p. 8).

Peter Selinger. Apr. 2001. "Control Categories and Duality: On the Categorical Semantics of the Lambda-Mu Calculus." *Mathematical Structures in Computer Science*, 11, 2, (Apr. 2001), 207–260. DOI: https://doi.org/10.1017/S096012950000311X (cit. on pp. 1, 3, 10, 13, 27, 35).

Martin Hofmann and Thomas Streicher. Dec. 15, 2002. "Completeness of Continuation Models for $\Lambda\mu$-Calculus." *Information and Computation*, 179, 2, (Dec. 15, 2002), 332–355. DOI: https://doi.org/10.1006/inco.2001.2947 (cit. on pp. 3, 8, 13).

Paul Taylor. July 2002. "Sober Spaces and Continuations." *Theory and Applications of Categories*, 10, 12, (July 2002), 248–299. PaulTaylor.EU/ASD/sobsc (cit. on pp. 9, 31).

Paul Blain Levy. 2003. "Jump-With-Argument." In: *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantic Structures in Computation. Ed. by Paul Blain Levy. Springer Netherlands, Dordrecht, 141–168. ISBN: 978-94-007-0954-6. DOI: https://doi.org/10.1007/978-94-007-0954-6_7 (cit. on pp. 14, 27).

Peter Selinger. 2003. "From Continuation Passing Style to Krivine's Abstract Machine." https://www.mscs.dal.ca/~selinger/papers/krivine.pdf (cit. on p. 27).

Philip Wadler. Aug. 25, 2003. "Call-by-Value Is Dual to Call-by-Name." In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (ICFP '03). Association for Computing Machinery, New York, NY, USA, (Aug. 25, 2003), 189–201. ISBN: 978-1-58113-756-9. DOI: https://doi.org/10.1145/944705.944723 (cit. on pp. 1, 3, 14, 19).

Carsten Führmann and Hayo Thielecke. Jan. 2004. "On the Call-by-Value CPS Transform and Its Semantics." *Information and Computation*, 188, 2, (Jan. 2004), 241–283. DOI: https://doi.org/10.1016/j.ic.2003.08.001 (cit. on p. 23).

Paul Blain Levy. 2006. "Jumbo λ-Calculus." In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Springer, Berlin, Heidelberg, 444–455. ISBN: 978-3-540-35908-1. DOI: https://doi.org/10.1007/11787006_38 (cit. on p. 4).

Martin Hyland, Paul Blain Levy, Gordon Plotkin, and John Power. May 1, 2007. "Combining Algebraic Effects with Continuations." *Theoretical Computer Science*. Festschrift for John C. Reynolds's 70th Birthday 375, 1, (May 1, 2007), 20–40. DOI: https://doi.org/10.1016/j.tcs.2006.12.026 (cit. on p. 13).

Paul-André Melliès and Nicolas Tabareau. 2007. "Linear Continuations and Duality." working paper or preprint. (2007). Retrieved Dec. 19, 2022 from https://hal.archives-ouvertes.fr/hal-00339156 (cit. on p. 31).

Alexis Saurin. 2008. "On the Relations between the Syntactic Theories of $\Lambda\mu$-Calculi." In: *Computer Science Logic*. Ed. by Michael Kaminski and Simone Martini. Springer, Berlin, Heidelberg, 154–168. ISBN: 978-3-540-87531-4. DOI: https://doi.org/10.1007/978-3-540-87531-4_13 (cit. on p. 27).

Zena M. Ariola, Hugo Herbelin, and Amr Sabry. Sept. 1, 2009. "A Type-Theoretic Foundation of Delimited Continuations." *Higher-Order and Symbolic Computation*, 22, 3, (Sept. 1, 2009), 233–273. DOI: https://doi.org/10.1007/s10990-007-9006-0 (cit. on p. 27).

Robert Atkey. July 2009. "Parameterised Notions of Computation." *Journal of Functional Programming*, 19, 3-4, (July 2009), 335–376. DOI: https://doi.org/10.1017/S095679680900728X (cit. on p. 27).

Noam Zeilberger. 2009. "The Logical Basis of Evaluation Order and Pattern-Matching." Ph.D. Dissertation. Carnegie Mellon University, USA. 191 pp. (cit. on p. 27).

Jean-Yves Girard. Sept. 25, 2011. *The Blind Spot: Lectures on Logic*. (1st ed.). EMS Press, (Sept. 25, 2011). ISBN: 978-3-03719-088-3. DOI: https://doi.org/10.4171/088 (cit. on p. 2).

Samson Abramsky. Mar. 19, 2012. *No-Cloning In Categorical Quantum Mechanics*. Comment: 35 pages. Appeared in Semantic Techniques in Quantum Computation, ed. S. Gay and I. Mackie, pages 1–28, Cambridge University Press 2010 Comment: 35 pages. Appeared in Semantic Techniques in Quantum Computation, ed. S. Gay and I. Mackie, pages 1–28, Cambridge University Press 2010. (Mar. 19, 2012). arXiv: 0910.2401 [quant-ph]. Retrieved June 1, 2023 from http://arxiv.org/abs/0910.2401. Pre-published (cit. on pp. 2, 3).

Marcelo Fiore. Sept. 22, 2013. "An Equational Metalogic for Monadic Equational Systems." *Theory and Applications of Categories*, 27, 18, (Sept. 22, 2013), 464–492. http://www.tac.mta.ca/tac/volumes/27/18/27-18abs.html (cit. on p. 27).

Jules Hedges. June 5, 2014. "Monad Transformers for Backtracking Search." *Electronic Proceedings in Theoretical Computer Science*, 153, (June 5, 2014), 31–50. DOI: https://doi.org/10.4204/EPTCS.153.3 (cit. on p. 25).

Tomas Petricek, Dominic Orchard, and Alan Mycroft. Aug. 19, 2014. "Coeffects: A Calculus of Context-Dependent Computation." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (ICFP '14). Association for Computing Machinery, New York, NY, USA, (Aug. 19, 2014), 123–135. ISBN: 978-1-4503-2873-9. DOI: https://doi.org/10.1145/2628136.2628160 (cit. on p. 1).

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. Jan. 11, 2016. "A Theory of Effects and Resources: Adjunction Models and Polarised Calculi." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '16). Association for Computing Machinery, New York, NY, USA, (Jan. 11, 2016), 44–56. ISBN: 978-1-4503-3549-2. DOI: https://doi.org/10.1145/2837614.2837652 (cit. on p. 27).

Harley Eades III and Gianluigi Bellin. Aug. 19, 2017. *A Cointuitionistic Adjoint Logic*. Comment: 54 pages. (Aug. 19, 2017). arXiv: 1708.05896 [cs]. Retrieved Jan. 3, 2023 from http://arxiv.org/abs/1708.05896. Pre-published (cit. on pp. 2, 3).

Paul-André Melliès. Feb. 1, 2017. "A Micrological Study of Negation." *Annals of Pure and Applied Logic*. Eighth Games for Logic and Programming Languages Workshop (GaLoP) 168, 2, (Feb. 1, 2017), 321–372. DOI: https://doi.org/10.1016/j.apal.2016.10.008 (cit. on p. 27).

Vikraman Choudhury and Neel Krishnaswami. Aug. 2, 2020. "Recovering Purity with Comonads and Capabilities." *Proceedings of the ACM on Programming Languages*, 4, (Aug. 2, 2020), 1–28, ICFP, (Aug. 2, 2020). DOI: https://doi.org/10.1145/3408993 (cit. on p. 18).

Paul Blain Levy. Apr. 2021. "Typed λ-Calculus: Course Notes." Midlands Graduate School in the Foundations of Computing Science, (Apr. 2021). https://www.cs.bham.ac.uk/~pbl/mgsall.pdf (cit. on p. 4).

[SW] Vikraman Choudhury, *Artifact for The Duality of λ-Abstraction* version v0.3.1, Oct. 30, 2024. Zenodo. DOI: https://doi.org/10.5281/ZENODO.14015102, URL: https://doi.org/10.5281/zenodo.14015102 (cit. on pp. 4, 28).

Jacob Errington. 2024. *Jacob Errington | SAT Solving with Higher-Order Continuations*. Retrieved Oct. 26, 2024 from https://jerrington.me/posts/2022-10-22-higher-order-continuations.html (cit. on p. 25).

Received 2024-07-11; accepted 2024-11-07