The Duality of Abstraction

ANONYMOUS AUTHOR(S)

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 abstraction.

We formalise this language by giving an extension of a call-by-value simply-typed lambda-calculus with coexponentials. We develop the semantics of this language using the axiomatic structure of continuations, which we use to produce an equational theory, that justifies control effects. We use this to derive the classical control operators and computational interpretation of classical logic, and encode common patterns of control flow using continuations, such as backtracking and exceptions. 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 their applications.

Additional Key Words and Phrases: duality, continuations, categorical semantics, type theory, effects

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) expressions and contexts [Curien and Herbelin 2000], (4) producers and consumers [Girard 1991], (5) client and server in session types [Honda 1993], (6) strict and lazy evaluation, (7) products and sums, (8) effects (monads) and coeffects (comonads).

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:

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

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 function $\mathsf{eval}_{A,B} \colon (A \Rightarrow B) \times A \longrightarrow B$ by uncurrying, allowing us to apply a function to an argument.

Duality is a fashionable trend in programming languages – can we dualise currying? Formally, this is a matter of reversing the arrows, turning the products into sums (coproducts), and turning the function type $\Rightarrow$ into a $\Leftarrow$ type:

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

Continuing the analogy with currying, the dual type $A \Leftarrow B$ (or coexponential object) should come with a (universal) coevaluation function $\mathsf{coeval}_{A,B} \colon B \longrightarrow A + (A \Leftarrow B)$. Programming languages have both products and sums, could we also have both $\Rightarrow$ and $\Leftarrow$?

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

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

POPL'24, January 17–19, 2024, London, UK
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Exponential objects $(-)^A$ give right ajoints 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 formally $\mathcal{C}^{\mathrm{op}}$ 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 (Lambek and Scott 1988, p.67; Girard 2011, § 7.A.4; Crolard 2001, thm 1.14; Abramsky 2012; Eades III and Bellin 2017).

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

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

In logic, the first isomorphism is the tautology $\bot \wedge A \leftrightarrow \bot$, and the second isomorphism is $\top \vee A \leftrightarrow \top$, which is well-known in classical logic. However, by Curry-Howard, in a programming language this means that the booleans would have no computational content – $\mathsf{Bool} \cong 1 + 1 \cong 1$, leading to a degenerate language.

$$\begin{tikzcd}
 & A & \\
1 \arrow[ur, "f"] \arrow[r, "\iota_1"'] & 1 + 1 \arrow[u, "{[f,g]}"] & 1 \arrow[ul, "g"'] \arrow[l, "\iota_2"]
\end{tikzcd}$$

Since $1 + 1 \cong 1$, it is a terminal object, making $\iota_1, \iota_2 : 1 \longrightarrow 1 + 1$ equal. If $f, g : 1 \longrightarrow A$ 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 *do not* embrace linear logic, yet produce a programming language with a computational interpretation, that has both currying and cocurrying!

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 contexts (Curien and Herbelin [2000]'s $\mu\tilde{\mu}$). Continuations give a computational interpretation of classical logic, as discovered by Griffin [1989]. The classical nature of the isomorphism in equation (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 equation (2), using their favorite control operators. This is the main insight on which this paper builds. Semantically-minded readers might want to skip ahead to § 4 to see what the trick is about.

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 is inspired by the works of 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 a sum(!) type – the interaction of usual sums and cofunctions allows values and covalues to interact. We introduce this language by examples in § 2, and give a formal presentation in § 3.
- We develop the semantics of $\lambda\tilde{\lambda}$ in two different ways, in § 4. First, we use continuations for covalues, and give a CPS semantics, essentially by pulling it out of a hat. Second, we perform a micrological study of continuations, understanding their axiomatic categorical structure, and how it produces exponentials and coexponentials. We interpret our language using this categorical semantics in § 5, and show that it matches the CPS semantics.
- Using our denotational semantics, we develop an equational theory for our language, in § 6. 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 dualizing functional completeness, in § 7. These languages are understood operationally using continuations as handlers.
- Unlike other dual calculi for continuations, ours is a natural deduction calculus. This means that the $\tilde{\lambda}$ duality readily adapts to a control operators, which we can implement or retrofit in real-world programming languages. In § 8, we implement them in SML and Haskell, and discuss our applications.

We include a partial formalisation of our languages in Agda, and implementations in SML and Haskell, as supplementary material. Some details are skipped in the main text, and included in the supplementary appendix.

2 DUALITY BY EXAMPLE

We illustrate the duality of abstraction by programming in a hypothetical language, whose syntax is similar to a typed programming language (like ML or Haskell).

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 → string) (g : int + string) : int → int + string =
    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 → string`, and a sum (or cofunction) `g : int + string`, and returns something of type `int → int + string`. 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 + string`. 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.toString (INR "0") 0
⇝   cofn (k : co int) ⇒ if 0 = 0 then (INR "0") @ k else Int.toString 0
⇝   cofn (k : co int) ⇒ (INR "0") @ k                                        (5)
⇝   INR "0"
```

```
     ex1 Int.toString (INL 1) 1
⇝   cofn (k : co int) ⇒ if 1 = 0 then (INL 1) @ k else Int.toString 1
⇝   cofn (k : co int) ⇒ Int.toString 1                                       (6)
⇝   cofn (k : co int) ⇒ "1"
⇝   INR "1"
```

The standard rules for capture-avoiding substitutions apply to both functions and cofunctions, which we perform implicitly. In equation (5), 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, in equation (6), the body of the inner cofunction reduces by following the right branch of the conditional, which produces the cofunction `cofn (k : co int) ⇒ Int.toString 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 → string) (g : int + string) : int → int + string =
    fn (x : int) ⇒
        if x = 0 then g else INR (f x)
```

What is then the point of cofunctions? We will see that, `INL`/`INR` produce ordinary sums, but cofunctions produce "Faustian" sums, which have control effects!

Exceptional cofunctions. Consider a simple program which multiplies the elements in a list (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
```

which 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.

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

`cofn` binds a covalue `k : int co`, and speculatively executes its body, assuming that its 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 : int co) ⇒ loop [1, 2, 0, 3, 4]
⇝  cofn (k : int co) ⇒ 1 * loop [2, 0, 3, 4]
⇝  cofn (k : int co) ⇒ 1 * (2 * loop [0, 3, 4])
⇝  cofn (k : int co) ⇒ 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 something of this type – a choice between a value and a covalue:

```
let val (idc : a + co a) = cofn (k : co a) ⇒ k
```

This is a cofunction that doesn't reduce on its own, not until you observe it by pattern matching! 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
```

The subtraction type is derived from covalues and is not included as a primitive type in our language, for «good reasons». 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) =
    fn x ⇒ fn y ⇒ f (x, y)
fun uncurry (f : 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 : co b) ⇒ (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 LEM. 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) : (b - c) + (a - b) =
    case coeval (#1 f) of
        INL b ⇒ INL (b, #2 f)
      | INR (a, k) ⇒ INR (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)) @ k
```

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
```

To a continuations afficionado, these operators will look familiar:

```
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)))
```

3 SYNTAX

We present a formal calculus called $\lambda\tilde{\lambda}$ which exhibits abstraction and coabstraction.

3.1 Typing

The syntax and typing of $\lambda\tilde{\lambda}$ is presented in figure 1. It is a simply-typed lambda calculus with products, functions, and sums, extended with covalue types, coabstraction, and coapplication.

In figure 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 bar 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. Importantly, this is a *call-by-value* language – 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 figure 1c. Unit and products have the usual *rightist* typing rules. 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 $A \Rightarrow B$. Application eliminates a function, which applies a function $A \Rightarrow B$ 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. Sums have *bipartisan* status – the interaction of leftist case and rightist $\tilde{\lambda}$ 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 binds a covalue $\tilde{A}$, it can use the bound variable $x : \tilde{A}$ in its body in any way that satisfies typing constraints.

We can think of colambdas as producing a right-biased choice, bargaining for a covalue for the left side of the choice. The informal idea behind a covalue is that it 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 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 the covalue opened up. This is explained by the computational behavior of coapplication.

3.2 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 figure 2, and define substitution on raw terms in definition 3.3 and definition 3.2. This is standard, and when substituting under a binder, we do a renaming of the bound variable by extending the substitution. Finally, we prove admissibility of weakening and substitution in theorem 3.1.

Theorem 3.1 (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 3.2 (Substitution on variables).

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

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

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

$$
\begin{array}{rl}
x : A \in \Gamma & x \text{ is a variable of type } A \text{ in context } \Gamma \\
\Gamma \supseteq \Delta & \Gamma \text{ is a weakening of } \Delta \\
\Gamma \vdash \theta : \Delta & \theta \text{ is a substitution from } \Gamma \text{ to } \Delta \\
\Gamma \vdash e : A & e \text{ is an expression of type } A \text{ in context } \Gamma \\
\Gamma \vdash e_1 \approx e_2 : A & e_1 \text{ and } e_2 \text{ are equal expressions of type } A \text{ in context } \Gamma
\end{array}
$$

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

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

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

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

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

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

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

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

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

(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

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

(c) Substitution Rules

Fig. 2. Membership, Weakening and Substitution Rules

Definition 3.3 (Substitution on raw terms).

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

4 SEMANTICS

4.1 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, shown in figure 3, which is familiar in the continuations literature [Streicher and Reus 1998]. It is given by a family of semantic functions indexed by types and contexts: $(\!|-|\!)^{\gamma}_{A} : \lambda\tilde{\lambda}_A \to (A \to R) \to R$.

The base language is assumed to have functions, products, and sums, 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 $A \to (B \to R) \to R$, as is standard.

The continuation of a lambda $\lambda x.\, e\colon A \Rightarrow B$ is $k\colon A \to (B \to R) \to R$, which we apply to a function that binds a value $a\colon A$, a continuation $k_B\colon B \to R$, and evaluates the body e in the extended context γ, a, using the continuation k_B. When translating an application $e_1\ e_2\colon B$, we grab

$$
\begin{aligned}
\llparenthesis \star \rrparenthesis^{\gamma}_{1} &\triangleq \lambda k.\, k \star \\
\llparenthesis (e_1, e_2) \rrparenthesis^{\gamma}_{A\times B} &\triangleq \lambda k.\ \llparenthesis e_2 \rrparenthesis^{\gamma}_{B}(\lambda b.\ \llparenthesis e_1 \rrparenthesis^{\gamma}_{A}(\lambda a.\, k\ (a, b))) \\
\llparenthesis \mathsf{fst}(e) \rrparenthesis^{\gamma}_{A} &\triangleq \lambda k.\ \llparenthesis e \rrparenthesis^{\gamma}_{A\times B}(\lambda p.\, k\ \mathsf{fst}(p)) \\
\llparenthesis \mathsf{snd}(e) \rrparenthesis^{\gamma}_{B} &\triangleq \lambda k.\ \llparenthesis e \rrparenthesis^{\gamma}_{A\times B}(\lambda p.\, k\ \mathsf{snd}(p)) \\
\llparenthesis \mathsf{inl}(e) \rrparenthesis^{\gamma}_{A+B} &\triangleq \lambda k.\ \llparenthesis e \rrparenthesis^{\gamma}_{A}(\lambda a.\, k\ \mathsf{inl}(a)) \\
\llparenthesis \mathsf{inr}(e) \rrparenthesis^{\gamma}_{A+B} &\triangleq \lambda k.\ \llparenthesis e \rrparenthesis^{\gamma}_{B}(\lambda b.\, k\ \mathsf{inr}(b)) \\
\llparenthesis \mathsf{case}(e, x.\, e_1, y.\, e_2) \rrparenthesis^{\gamma}_{C} &\triangleq \lambda k.\ \llparenthesis e \rrparenthesis^{\gamma}_{A+B}\left(\lambda \begin{cases} \mathsf{inl}(a).\ \llparenthesis e_1 \rrparenthesis^{\gamma,a}_{C}\ k \\ \mathsf{inr}(b).\ \llparenthesis e_2 \rrparenthesis^{\gamma,b}_{C}\ k \end{cases}\right) \\
\llparenthesis x \rrparenthesis^{\gamma}_{A} &\triangleq \lambda k.\, k(\gamma(x)) \\
\llparenthesis \lambda x.\, e \rrparenthesis^{\gamma}_{A\Rightarrow B} &\triangleq \lambda k.\, k(\lambda a.\, \lambda k_B.\ \llparenthesis e \rrparenthesis^{\gamma,a}_{B}\ k_B) \\
\llparenthesis e_1\ e_2 \rrparenthesis^{\gamma}_{B} &\triangleq \lambda k_B.\ \llparenthesis e_2 \rrparenthesis^{\gamma}_{A}(\lambda a.\ \llparenthesis e_1 \rrparenthesis^{\gamma}_{A\Rightarrow B}(\lambda f.\, f\ a\ k_B)) \\
\llparenthesis \widetilde{\lambda} x.\ e \rrparenthesis^{\gamma}_{A+B} &\triangleq \lambda k.\ \begin{array}{ll} \mathit{let} & \left\{ \begin{array}{lcl} k_A & \triangleq & \lambda a.\, k\ \mathsf{inl}(a) \\ k_B & \triangleq & \lambda b.\, k\ \mathsf{inr}(b) \end{array} \right. \\ \mathit{in} & \llparenthesis e \rrparenthesis^{\gamma,k_A}_{B}(k_B) \end{array} \\
\llparenthesis \widetilde{e_1\ e_2} \rrparenthesis^{\gamma}_{B} &\triangleq \lambda k_B.\ \llparenthesis e_2 \rrparenthesis^{\gamma}_{\widetilde{A}}\left(\lambda k_A.\ \llparenthesis e_1 \rrparenthesis^{\gamma}_{A+B}\left(\lambda \begin{cases} \mathsf{inl}(a).\ k_A\ a \\ \mathsf{inr}(b).\ k_B\ b \end{cases}\right)\right)
\end{aligned}
$$

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

a continuation k_B, first evaluating the argument e_2, which requires a continuation $A \rightarrow R$. We pass a continuation that binds the value $a\colon A$, then evaluates the function e_1 in the same context, passing a and k_B as arguments.

Dual to functions, the continuation of a colambda $\widetilde{\lambda} x.\ e\colon A + B$ is $k\colon (A + B) \rightarrow R$, which we can (crucially) split into two continuations $k_A\colon A \rightarrow R$ and $k_B\colon B \rightarrow R$. 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 $\widetilde{e_1\ e_2}\colon B$, we grab a continuation k_B, then evaluate the argument e_2, which requires a continuation $(A \rightarrow R) \rightarrow R$. We pass a continuation which binds the covalue (or continuation) $k_A\colon A \rightarrow R$, then evaluates the body e_1 in the same context. This requires a continuation $(A + B) \rightarrow R$, 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 slogan "higher-order continuations". The cocurrying isomorphism from equation (2) is:

$$\Gamma \times (A \rightarrow R) \rightarrow ((B \rightarrow R) \rightarrow R) \cong \Gamma \rightarrow ((A + B) \rightarrow R) \rightarrow R,$$

the lesson being, if you CPS your program, you can dualise functions!

4.2 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 reasons or beauty behind the duality in this language. The right way to understand this language is to understand the abstract structure of continuation semantics using category theory. This is necessary to further develop the metatheory of our language – its equational theory. The ideas here are well-known in the semantics of continuations (from Hofmann, Streicher, Thielecke, Selinger, Fuhrmann). We present a slightly different point of view.

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 Hofmann [1995], Streicher and Reus [1998], and Hofmann and Streicher [2002]

We develop the abstract structure of the first point of view, then instantiate it with the second point of view. Let $\mathcal{C}$ be a (locally small) category with a functor $\neg\colon \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$, that is self-adjoint on the right. This means, for any objects $A, B \in \mathcal{C}$, we have the hom-set isomorphism: $\mathcal{C}(B, \neg A) \cong \mathcal{C}(A, \neg B)$.

$$\begin{tikzcd} \mathcal{C}^{\mathrm{op}} \arrow[r, bend right, "\neg"'] \arrow[r, phantom, "\perp"] & \mathcal{C} \arrow[l, bend right, "\neg"'] \end{tikzcd}$$

The full image of $\neg\colon \mathcal{C}^{\mathrm{op}} \to \mathcal{C}$, written $\mathcal{C}_\neg$, is the (bijective-on-objects, fully faithful) factorisation of $\neg$, which upto equivalence of categories, is determined by $\mathcal{C}_\neg(A, B) \triangleq \mathcal{C}(\neg A, \neg B)$. The functor $\neg_{\mathrm{bo}}\colon \mathcal{C}^{\mathrm{op}} \to \mathcal{C}_\neg$ is identity on objects, and negates morphisms, and the functor $\neg_{\mathrm{ff}}\colon \mathcal{C}_\neg \to \mathcal{C}$ negates objects, and is identity on morphisms. Dually, the full image of $\neg^{\mathrm{op}}\colon \mathcal{C} \to \mathcal{C}^{\mathrm{op}}$ is $\mathcal{C}_\neg^{\mathrm{op}}$.

$$\begin{tikzcd} \mathcal{C}^{\mathrm{op}} \arrow[rr, "\neg"] \arrow[dr, "\neg_{\mathrm{bo}}"'] & & \mathcal{C} \\ & \mathcal{C}_\neg \arrow[ur, "\neg_{\mathrm{ff}}"'] & \end{tikzcd}$$

With this situation in mind, we observe the following:

Proposition 4.1.

(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,* $\neg_{\mathrm{bo}}^{\mathrm{op}}$ *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* $\mathcal{C}_\neg^{\mathrm{op}}$ *is equivalent to the Kleisli category.*

Proof. We have 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. And, $\mathcal{C}_\neg(A, B) = \mathcal{C}(\neg A, \neg B) \cong \mathcal{C}(B, \neg\neg A) \equiv \mathcal{C}_{\neg\neg}^{\mathrm{op}}(A, B)$. □

This general situation is exploited to understand the structure of the Kleisli category of the continuation monad.

Proposition 4.2. *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.*

Proposition 4.3. *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.*

Here the trick is revealed, we have a cartesian closed category of values, a cocartesian coclosed Kleisli category (of a strong monad) of computations, and we put them together in a call-by-value language using the ideas of Moggi [1989]! There is no mathematical trickery here, and we're simply exploiting well-known mathematical structure and dressing it up. This is a matter of appearances – a common trait of good magic tricks, and programming language design. Following Taylor [2002], values are X, and we think of covalues/continuations as observations R^X, and computations R^{R^X} are meta-observations.

In terms of Selinger [2001, remark 1.1], $\mathcal{C}_{R^{(-)}}$ is a control category (the interpretation of cbn), and the Kleisli category is a co-control category (the interpretation of cbv). Explicitly, we give all the structure below, which we will use to give a categorical and denotational semantics for $\lambda\tilde{\lambda}$. Following Taylor [2002], we write R^2X for R^{R^X}.

Definition 4.4.

$$\begin{array}{rcl} K\colon \mathcal{C} & \to & \mathcal{C} \\ X & \mapsto & R^2X \\ X \xrightarrow{f} Y & \mapsto & \begin{array}{rcl} K(f)\colon R^2X & \to & R^2Y \\ k' & \mapsto & \lambda(k:R^Y).\,k'(k\circ f) \end{array} \end{array}$$

The monad structure is given by:

Definition 4.5.

$$\begin{array}{rcl} \eta_X\colon X & \to & K(X) \\ x & \mapsto & \lambda(k:R^X).\,k(x) \end{array} \qquad \begin{array}{rcl} \mu_X\colon K^2(X) & \to & K(X) \\ k' & \mapsto & \lambda(k:R^X).\,k'(\lambda(k'':R^2X).\,k''(k)) \end{array}$$

K is canonically strong with respect to $\times$, because $\mathcal{C}$ is cartesian closed. The left and right strengths are given by:

Definition 4.6.

$$\begin{array}{rcl} \tau_{X,Y}\colon X\times KY & \to & K(X\times Y) \\ (x,k') & \mapsto & K(\lambda(y:Y).\,(x,y))(k') \end{array} \qquad \begin{array}{rcl} \sigma_{X,Y}\colon KX\times Y & \to & K(X\times Y) \\ (k',y) & \mapsto & K(\lambda(x:X).\,(x,y))(k') \end{array}$$

The continuation monad K is not commutative because there are two ways to go from $KX\times KY \to K(X\times Y)$ and they are not necessarily equal. If K were commutative, then $\mathcal{C}_K$ (the Kleisli category of K) would be star-autonomous (an observation by Hasegawa, see [Melliès and Tabareau 2007]). We write one of the maps (following right-to-left evaluation order) as:

Definition 4.7.

$$\beta_{X,Y} \triangleq K(X)\times K(Y) \xrightarrow{\tau_{KX,Y}} K(KX\times Y) \xrightarrow{K\sigma_{X,Y}} K^2(X\times Y) \xrightarrow{\mu_{X\times Y}} K(X\times Y)$$

Given a Kleisli arrow $f\colon X\to KY$, its lift is $f^\dagger\colon KX \xrightarrow{Kf} K^2Y \xrightarrow{\mu_Y} KY$. The Kleisli composition of $X\xrightarrow{f} KY$ and $Y\xrightarrow{g} KZ$ is $X\xrightarrow{f} KY \xrightarrow{g^\dagger} KZ$. K has Kleisli exponentials since:

$$\mathcal{C}_K(Z\times X, Y) \equiv \mathcal{C}(Z\times X, KY) \cong \mathcal{C}(Z, X\to KY) \equiv \mathcal{C}(Z, X\Rightarrow Y).$$

Coproducts in $\mathcal{C}_K$ are given by the underlying coproducts in $\mathcal{C}$:

$$\mathcal{C}_K(X+Y,Z) \equiv \mathcal{C}(X+Y,KZ) \cong \mathcal{C}(X,KZ)\times\mathcal{C}(Y,KZ) \equiv \mathcal{C}_K(X,Z)\times\mathcal{C}_K(Y,Z).$$

Finally, we have this hom-set isomorphism in $\mathcal{C}_K$:

$$\mathcal{C}_K(Z\times R^X, Y) \equiv \mathcal{C}(Z\times R^X, R^{R^Y}) \cong \mathcal{C}(Z\times R^X\times R^Y, R) \cong \mathcal{C}(Z\times R^{X+Y}, R) \cong \mathcal{C}(Z, R^{R^{X+Y}}) \equiv \mathcal{C}_K(Z, X+Y).$$

Observe that $(-) \times R^X$ is an endofunctor on $\mathcal{C}_K$, since:

$$Y \xrightarrow{f} KZ \mapsto Y \times R^X \xrightarrow{f \times R^X} KZ \times R^X \xrightarrow{\sigma_{Z,R^X}} K(Z \times R^X).$$

This means, we have the following adjoint situation in $\mathcal{C}_K$, where ${}^X Y \triangleq Y \times R^X$:

$${}^X(-) \dashv X + (-).$$

To work with exponentials and co-exponentials, we adopt the musical notation of adjuncts (flats on the left, sharps on the right) as follows: 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 4.8.

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

4.3 The Indiana Control Operators

The conceptual understanding of the negation functor and the interplay of sums and products in $\mathcal{C}_R$, $\mathcal{C}_K$ is crucial to understanding the computational behaviour of control. We don't say much about the axiomatics here, but this is used in § 6.

The Kleisli inclusion is an ioo functor, and has a right adjoint, hence preserves coproducts. 0 is an initial object in $\mathcal{C}_K$, and further $K(0) = R^2 0 \cong R^1 \cong R$. Note that, 0 is not a strict initial object, meaning that, having an arrow $A \to 0$ does not imply $A \cong 0$. We can have non-trivial arrows to 0, and these are indeed continuations of A in $\mathcal{C}_K$. The R object in $\mathcal{C}_K$ enjoys a special status, producing control operators.

The Indiana control operators (Felleisen's $\mathcal{A}$ and $\mathcal{C}$) are developed in our semantics as follows. These are axiomatically understood using Hofmann [1995]'s equations. (also see [Hyland et al. 2007, prop 1, 2]).

$$\mathcal{C}_{\mathsf{A}} \triangleq R^{R^A} \xrightarrow{\sim} R^{R^A}$$

$$\mathcal{A}_{\mathsf{A}} \triangleq (A \xrightarrow{\iota_1} A + B \xrightarrow{\eta_{A+B}} K(A+B))^\flat \colon A \times R^A \to K(B)$$

$$\mathsf{tnd}_{\mathsf{A}} \triangleq (\Gamma \times R^A \xrightarrow{\pi_2} R^A \xrightarrow{\eta_{R^A}} K(R^A))^\sharp \colon \Gamma \to K(A + R^A)$$

$$\mathsf{callcc}_{\mathsf{A}}(\Gamma \times R^A \xrightarrow{f} K(A)) \triangleq \Gamma \xrightarrow{f^\sharp} K(A+A) \xrightarrow{K\nabla_A} K(A)$$

$$\mathsf{call/cc}_{\mathsf{A,B}}(\Gamma \times (R^{R^B \times A}) \xrightarrow{f} K(A)) \triangleq \Gamma \xrightarrow{f^\sharp} K(R^B \times A + A) \xrightarrow{K[\pi_2, 1_A]} K(A)$$

The duplicating nature of pattern matching on sums is understood using:

$$\begin{tikzcd}
K(A+B) \arrow[r, "{K[f,g]}"] \arrow[d, "\wr"'] & KC \arrow[d, equal] \\
R^{R^A \times R^B} \arrow[r, "R^{\langle R^f, R^g \rangle}"'] & R^2 C
\end{tikzcd}$$

$$
\begin{aligned}
[\![\mathbf{1}]\!] &\triangleq 1 \\
[\![\mathbf{0}]\!] &\triangleq 0 \\
[\![A \Rightarrow B]\!] &\triangleq (K[\![B]\!])^{[\![A]\!]}
\end{aligned}
\qquad
\begin{aligned}
[\![A \times B]\!] &\triangleq [\![A]\!] \times [\![B]\!] \\
[\![A + B]\!] &\triangleq [\![A]\!] + [\![B]\!] \\
[\![\tilde{A}]\!] &\triangleq R^{[\![A]\!]}
\end{aligned}
$$

(a) $[\![A]\!]$: Obj

$$
\begin{aligned}
[\![\cdot]\!] &\triangleq 1 \\
[\![\Gamma, x : A]\!] &\triangleq [\![\Gamma]\!] \times [\![A]\!]
\end{aligned}
$$

(b) $[\![\Gamma]\!]$: Obj

Fig. 4. Interpretation of types and contexts

5 INTERPRETATION

We now give an interpretation for $\lambda\tilde{\lambda}$ using the categorical structure we've defined. This is standard call-by-value semantics from Moggi [1989], with the addition of coproducts and coexponentials.

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

Expressions. Expressions are interpreted as Kleisli arrows, that is, morphisms in $\mathcal{C}_K$, as shown in figure 5. This is standard call-by-value semantics from Moggi [1989], with sums and cofunctions. Sums use the cocartesian structure, and distributivity for case. Cofunctions are interpreted using the coexponential adjunction.

Weakening and Substitution. Membership and weakening are interpreted using projections of contexts, as shown in figure 6. To interpret substitutions, we need a value interpretation. The interpretation of values and substitutions is shown in figure 7. The value interpretation is coherent with the expression interpretation, which we use to prove semantic weakening and substitution, in theorem 5.1.

Theorem 5.1 (Semantic Weakening and Substitution).

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

Soundness. Using § 4.2, and by unpacking the definition of the continuation monad, we can show that this is sound with respect to the CPS translation in § 4.1.

Theorem 5.2. *If $\Gamma \vdash e : A$, then $(\!|e|\!)^{\gamma}_{A}(k) = [\![\Gamma \vdash e : A]\!](\gamma, k)$, for any $\gamma \in [\![\Gamma]\!]$ and $k \in [\![A]\!] \to R$.*

6 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. The equivalence and congruence rules are standard, and we give the additional conversion rules in figure 8.

The conversion rules are the basic ones for call-by-value – extended with sums and co-exponentials in figure 8. Beta laws for both functions and cofunctions are upto values, because 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 eta laws for expressions! These equations don't perform any control effects – so far they're only exploiting the two adjunctions to validate binding rules.

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 figure 8. These are inspired

$$\left[\!\left[\frac{}{\Gamma \vdash \star : \mathbf{1}}\right]\!\right] \triangleq !_\Gamma \,;\, \eta_1$$

$$\left[\!\left[\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\right]\!\right] \triangleq \begin{array}{l} \mathit{let} \left\{ \begin{array}{lcl} f & \triangleq & [\![\Gamma \vdash e_1 : A]\!] \\ g & \triangleq & [\![\Gamma \vdash e_2 : B]\!] \end{array} \right. \\ \mathit{in} \;\; \langle f \,,\, g \rangle \,;\, \beta_{A,B} \end{array}$$

$$\left[\!\left[\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{fst}(e) : A}\right]\!\right] \triangleq [\![\Gamma \vdash e_1 : A]\!] \,;\, K\pi_1 \qquad \left[\!\left[\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \mathsf{snd}(e) : B}\right]\!\right] \triangleq [\![\Gamma \vdash e_2 : B]\!] \,;\, K\pi_2$$

$$\left[\!\left[\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\right]\!\right] \triangleq [\![x : A \in \Gamma]\!] \,;\, \eta_A$$

$$\left[\!\left[\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).\, e : A \Rightarrow B}\right]\!\right] \triangleq \begin{array}{ll} \mathit{let} & f \;\triangleq\; [\![\Gamma, x : A \vdash e : B]\!] \\ \mathit{in} & f^\sharp \,;\, \eta_{A \to KB} \end{array}$$

$$\left[\!\left[\frac{\Gamma \vdash e_1 : A \Rightarrow B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\right]\!\right] \triangleq \begin{array}{l} \mathit{let} \left\{ \begin{array}{lcl} f & \triangleq & [\![\Gamma \vdash e_1 : A \Rightarrow B]\!] \\ g & \triangleq & [\![\Gamma \vdash e_2 : A]\!] \end{array} \right. \\ \mathit{in} \;\; \langle f \,,\, g \rangle \,;\, \beta_{A \to KB, A} \,;\, K\mathsf{ev}_{A,KB} \,;\, \mu_B \end{array}$$

$$\left[\!\left[\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}(e) : A + B}\right]\!\right] \triangleq [\![\Gamma \vdash e : A]\!] \,;\, Ki_1 \qquad \left[\!\left[\frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}(e) : A + B}\right]\!\right] \triangleq [\![\Gamma \vdash e : B]\!] \,;\, Ki_2$$

$$\left[\!\left[\frac{\Gamma \vdash e_1 : A + B \qquad \Gamma, x : e_2 \vdash A : C \qquad \Gamma, y : e_3 \vdash B : C}{\Gamma \vdash \mathsf{case}(e_1, x.\, e_2, y.\, e_3) : C}\right]\!\right] \triangleq \begin{array}{l} \mathit{let} \left\{ \begin{array}{lcl} f & \triangleq & [\![\Gamma \vdash e_1 : A + B]\!] \\ g_1 & \triangleq & [\![\Gamma, x : A \vdash e_2 : C]\!] \\ g_2 & \triangleq & [\![\Gamma, y : B \vdash e_3 : C]\!] \end{array} \right. \\ \mathit{in} \;\; \langle \mathsf{id}_\Gamma \,,\, f \rangle \,;\, \sigma_{\Gamma,A} \,;\, K\delta_{\Gamma,A,B} \,;\, K[g_1 \,,\, g_2] \,;\, \mu_C \end{array}$$

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

$$\left[\!\left[\frac{\Gamma \vdash e_1 : A + B \qquad \Gamma \vdash e_2 : \tilde{A}}{\Gamma \vdash \widetilde{e_1\ e_2} : B}\right]\!\right] \triangleq \begin{array}{l} \mathit{let} \left\{ \begin{array}{lcl} f & \triangleq & [\![\Gamma \vdash e_1 : A + B]\!] \\ g & \triangleq & [\![\Gamma \vdash e_2 : \tilde{A}]\!] \end{array} \right. \\ \mathit{in} \;\; \langle f \,,\, g \rangle \,;\, \tau_{K(A+B),R^A} \,;\, K\mathsf{id}^\flat_{K(A+B)} \,;\, \mu_B \end{array}$$

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

by Hofmann [1995] and Hofmann and Streicher [2002]'s equations, and Selinger [2001] equations for cbv $\lambda\mu$. We use evaluation contexts instead of commuting conversions, with appropriate freeness assumptions $\mathcal{E} = \mathcal{E}«\cdot» \mid e\ \ \mathcal{E} \mid \mathcal{E}\ \ v \mid \widetilde{e\ \mathcal{E}} \mid \widetilde{e\ \mathcal{E}} \mid \mathsf{fst}(\mathcal{E}) \mid \mathsf{snd}(\mathcal{E}) \mid (e, \mathcal{E}) \mid (\mathcal{E}, v)$. We remark that these operators have better types (than callcc/abort) – better types give better equations!

$\widetilde{\lambda}$-CONST is the constant rule – when the covalue is not used, it's a right-biased sum (cf. Selinger's $\mathrm{let}_{\mathrm{name}}$). The next two rules are the interaction of normal sums and cofunctions. $\widetilde{\lambda}$-inr-PASS is like transparent passthrough, or that inr produces normal return – if the covalue was created by $\tilde{\lambda}$ but then used with 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" (see[Thielecke 1999; Levy 2003]), where we throw to

$$[\![\frac{}{x : A \in (\Gamma, x : A)}]\!] \triangleq \pi_2$$

$$[\![\frac{x : A \in \Gamma \qquad (x \neq y)}{x : A \in (\Gamma, y : B)}]\!] \triangleq \pi_1 \,;\, [\![x : A \in \Gamma]\!]$$

(a) $[\![x : A \in \Gamma]\!] : \mathrm{Hom}([\![\Gamma]\!], [\![A]\!])$

$$[\![\frac{}{\cdot \supseteq \cdot}]\!] \triangleq id_1$$

$$[\![\frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta}]\!] \triangleq \pi_1 \,;\, [\![\Gamma \supseteq \Delta]\!]$$

$$[\![\frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta, x : A}]\!] \triangleq [[\![\Gamma \supseteq \Delta]\!] \times id_A]$$

(b) $\mathrm{Wk}(\Gamma \supseteq \Delta) \triangleq [\![\Gamma \supseteq \Delta]\!] : \mathrm{Hom}([\![\Gamma]\!], [\![\Delta]\!])$

Fig. 6. Interpretation of Membership and Weakening

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

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

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

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

$$[\![\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}(e) : A + B}]\!]_v \triangleq [\![\Gamma \vdash v : A]\!]_v \,;\, i_1 \qquad [\![\frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}(e) : A + B}]\!]_v \triangleq [\![\Gamma \vdash v : B]\!]_v \,;\, i_2$$

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

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

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

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

Fig. 7. Interpretation of values and substitution

the left. This behaves like Felleisen's $\mathcal{A}$ (Hofmann'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 shows the duplication of evaluation contexts on both branches of case. These equations are proven sound with respect to the semantics developed.

Theorem 6.1 (Soundness of equational theory). *If* $\Gamma \vdash e_1 \approx e_2 : A$, *then* $[\![\Gamma \vdash e_1 : A]\!] = [\![\Gamma \vdash e_2 : A]\!]$.

$$\frac{\Gamma \vdash v : \mathbf{1}}{\Gamma \vdash v \approx \star : \mathbf{1}}\ \mathbf{1}\eta$$

$$\frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash \mathsf{fst}((v_1, v_2)) \approx v_1 : A}\ \times_1\beta \qquad \frac{\Gamma \vdash v_1 : A \qquad \Gamma \vdash v_2 : B}{\Gamma \vdash \mathsf{snd}((v_1, v_2)) \approx v_2 : B}\ \times_2\beta$$

$$\frac{\Gamma \vdash v : A \times B}{\Gamma \vdash (\mathsf{fst}(v), \mathsf{snd}(v)) \approx v : A \times B}\ \times\eta$$

$$\frac{\Gamma \vdash v : A \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{case}(\mathsf{inl}(v), x.\, e_1, y.\, e_2) \approx [v/x]e_1 : C}\ +\mathsf{inl}\beta$$

$$\frac{\Gamma \vdash v : B \qquad \Gamma, x : A \vdash e_1 : C \qquad \Gamma, y : B \vdash e_2 : C}{\Gamma \vdash \mathsf{case}(\mathsf{inr}(v), x.\, e_1, y.\, e_2) \approx [v/y]e_2 : C}\ +\mathsf{inr}\beta$$

$$\frac{\Gamma \vdash e : A + B}{\Gamma \vdash \mathsf{case}(e, x.\, \mathsf{inl}(x), y.\, \mathsf{inr}(y)) \approx e : A + B}\ +\eta$$

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

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

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

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

Proof. These are checked using universal properties, and value and substitution lemmas. For control effects, we develop some axiomatic structure of control operators and exploit them. □

From this, it also follows that the equational theory is sound with respect to the continuation semantics.

Corollary 6.2. *If* $\Gamma \vdash e_1 \approx e_2 : A$*, then for any* γ *and* k*,* $(\!|e_1|\!)^{\gamma}_{A}(k) = (\!|e_2|\!)^{\gamma}_{A}(k)$.

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

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

$$\frac{\Gamma \vdash e : B \qquad \Gamma \vdash \mathcal{E}«e» : C \qquad \Gamma, x : \tilde{A} \vdash \mathcal{E}«\widetilde{\mathsf{inr}_A(e)\ x}» : C}{\Gamma \vdash (\widetilde{\lambda}(x : \tilde{A}).\ \mathcal{E}«\widetilde{\mathsf{inr}_A(e)\ x}») \approx \mathsf{inr}_A(\mathcal{E}«e») : A + C}\ \widetilde{\lambda}\text{-inr-PASS}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : \tilde{A} \vdash \mathcal{E}«\widetilde{\mathsf{inl}_C(e)\ x}» : B}{\Gamma \vdash (\widetilde{\lambda}(x : \tilde{A}).\ \mathcal{E}«\widetilde{\mathsf{inl}_C(e)\ x}») \approx \mathsf{inl}_B(e) : A + B}\ \widetilde{\lambda}\text{-inl-JUMP}$$

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

$$\frac{\Gamma \vdash \mathcal{E}«\mathsf{case}(e, x.\ e_1, y.\ e_2)» : C \qquad \Gamma \vdash \mathsf{case}(e, x.\ \mathcal{E}«e_1», y.\ \mathcal{E}«e_2») : C}{\Gamma \vdash \mathcal{E}«\mathsf{case}(e, x.\ e_1, y.\ e_2)» \approx \mathsf{case}(e, x.\ \mathcal{E}«e_1», y.\ \mathcal{E}«e_2») : C}\ \text{CASE-}\zeta$$

Fig. 8. Control effects in $\lambda\tilde{\lambda}$

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

We can define $\mathcal{C}_A : \tilde{\tilde{A}} \to A$ and its inverse from TND (double negation elimination and introduction).

```
fun C (kka : co (co a)) =
  case (cofn x => x) of
    INL a => a
  | INR ka => (INL ka) @ kka
```

```
fun C^ (a : a) =
  case (cofn x => x) of
    INL ka => (INL a) @ ka
  | INR kka => ka
```

We can further verify that these equations validate Hofmann and Streicher [1997]'s axiomatics of cbv control operators. Since we don't have **0**, some of these equations have to be adjusted from Hofmann's versions.

Proposition 6.3. *The equational theory validates these equations:*

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

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

$$\mathsf{call/cc}_{A,B}(\lambda(k : A \to B).\ \mathcal{E}«k\ e») = e \qquad \mathsf{call/cc}\text{-}ABS$$

$$\mathsf{call/cc}_{A,B}(e) = \mathsf{call/cc}_{A,C}(\lambda(k : A \to C).\ e(\lambda(x : A).\ \mathcal{E}«kx»)) \qquad \mathsf{call/cc}\text{-}APP$$

$$\mathcal{E}«\mathsf{call/cc}_{A,B}» = \mathsf{call/cc}_{C,B}(\lambda(k : C \to B).\ \mathcal{E}«e(\lambda(x : A).\ k(\mathcal{E}«x»))») \qquad \mathsf{call/cc}\text{-}NAT$$

Not including the **0** type is a stylistic choice, not a semantic one, since it is interpreted by R, and $\tilde{A}$ becomes equivalent to $A \Rightarrow \mathbf{0}$ after adding equations for $\mathcal{A}$. 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.

7 DUALITY OF ARROWS

The λ-calculus (or higher-order functions) can be decomposed into first-order κ/ζ calculi [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.

The essential idea behind this is Lambek [1974]'s functional completeness – a consequence of cartesian closure. We perform a conceptual reconstruction of Hasegawa's ideas using abstract properties of (co)monads and adjunctions, allowing us to dualise each step.

7.1 Dualizing Functional Completeness

From an observation, originally due to Hermida [1993]:

Proposition 7.1 (Hermida [1993, Prop. 5.2.1]). *Given a comonad $G\colon \mathcal{C} \to \mathcal{C}$ and its Kleisli resolution $F_G \dashv U_G\colon \mathcal{C} \to \mathcal{C}_G$,*

the following are equivalent:

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

Under either of the above equivalent hypotheses, $T(= R \circ U_G)$ 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 $U_T \dashv F_T\colon \mathcal{C} \to \mathcal{C}_T$, the following are equivalent:

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

Then, $G(= L \circ F_T)$ is the functor part of a comonad, and the corresponding 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 $T = R \circ U_G$ 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. Dually, $C + (-)$ is an exception monad, with a free covalue $C \rightsquigarrow 0$ in its environment, that is, an escape hatch to jump to C. In $\tilde{\lambda}$ (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 for C.

Proposition 7.2. *In a cartesian closed category 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.*

In a cocartesian coclosed category with ${}^{c}(-) \dashv c + (-)\mathcal{C} \to \mathcal{C}$:

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) $e_c : 1 \longrightarrow c$, 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 element (covalue) $e_{\tilde{c}} : c \longrightarrow 0$, given by $c \xrightarrow{\sim} c + 0$. From this, we derive functional completeness and its dual:

Proposition 7.3 (Functional completeness). *Let $\mathcal{C}$ and $\mathcal{D}$ be cartesian closed categories and $F\colon \mathcal{C} \longrightarrow \mathcal{D}$ a ccc functor. Let $c \in \mathcal{C}$, and $t : F(1) \cong 1 \longrightarrow F(c)$ be an element in $\mathcal{D}$. There is a unique (upto isomorphism) extension of F to a ccc functor $\tilde{F}\colon \mathcal{C}[c] \longrightarrow \mathcal{D}$, such that $\tilde{F} \circ U_{c\times(-)} \cong F$, and $F(e_c) = t$.*

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

$$\begin{tikzcd} \mathcal{C} \arrow[r, "U_{c\times(-)}"] \arrow[dr, "F"'] & \mathcal{C}[c] \arrow[d, "\tilde{F}"] \\ & \mathcal{D} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{C} \arrow[r, "F_{c+(-)}"] \arrow[dr, "F"'] & \mathcal{C}[\tilde{c}] \arrow[d, "\tilde{F}"] \\ & \mathcal{D} \end{tikzcd}$$

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

$$F(a) \xrightarrow{\sim} 1 \times F(a) \xrightarrow{\sim} F(1) \times F(a) \xrightarrow{t \times F(a)} F(c) \times F(a) \xrightarrow{\sim} F(c \times a) \xrightarrow{F(f)} F(b)$$

$$F(a) \xrightarrow{F(f)} F(c + b) \xrightarrow{\sim} F(c) + F(b) \xrightarrow{t + F(b)} F(0) + F(b) \xrightarrow{\sim} 0 + F(b) \xrightarrow{\sim} F(b)$$

□

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.

$$\begin{array}{ll} F_{c\times(-)} \dashv U_{c\times(-)} : \mathcal{C} \longrightarrow \mathcal{C}[c] & \quad F_{(-)^c} \dashv U_{(-)^c} : \mathcal{C}[c] \longrightarrow \mathcal{C} \\ F_{c+(-)} \dashv U_{c+(-)} : \mathcal{C}[\tilde{c}] \longrightarrow \mathcal{C} & \quad F_{{}^{c}(-)} \dashv U_{{}^{c}(-)} : \mathcal{C} \longrightarrow \mathcal{C}[\tilde{c}] \end{array}$$

7.2 κ/ζ and $\tilde{\kappa}/\tilde{\zeta}$ calculi

From this analysis, we extract a presentation of the dual $\tilde{\kappa}/\tilde{\zeta}$ calculi, with substitution and equations, given in figures 9 and 10. We see the $-$ type in action, for the handler comonad.

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 $C \longrightarrow R^2 0 \cong C \longrightarrow R$ – a continuation, and these first-order operations bind and apply covalues on arrows, changing control flow. The type $A - C$ is interpreted as A with a handler for C attached, and $C + B$ is a B program that could throw an A. The operators themselves don't have 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):

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

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

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

(a) κ calculus

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

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

(b) ζ calculus

Fig. 9. κ and ζ calculi

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

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

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

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

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

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

Fig. 10. $\tilde{\kappa}$ and $\tilde{\zeta}$ calculi

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

8 IMPLEMENTING $\tilde{\lambda}$

Conor McBride once said:

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

Unlike other dual calculi, ours readily adapts to control operators because of its natural deduction style presentation, that can be implemented in or retrofit into a real-world programming language. We can implement $\tilde{\lambda}$ using native continuations (like in SML), or we can implement it using a continuation monad (like in Haskell). We describe both implementations, and their applications are in the supplementary material.

SML. In SML, we implement them using the native continuation type with control operators callcc/throw, and sum types. This encoding shows how colam/coapp are like callcc/throw, but with fancier types.

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

We then recover callcc/throw from colam/coapp.

```
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
```

Haskell. In Haskell, we implement using the continuation monad `Cont r`.

```
colam :: ((a → r) → Cont r b) → Cont r (a + b)
colam f = cont $ \k →
  let k1 = k . Left
      k2 = k . Right
  in runCont (f k1) k2

coapp :: Cont r (a + b) → (a → r) → Cont r b
coapp e1 k1 = cont $ \k2 →
  runCont e1 $ \case
    Left a → k1 a
    Right b → k2 b
```

Backtracking. 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.

```
swap :: a + b → b + a
swap = either Right Left

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

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

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

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

Using this DSL for backtracking, we program a SAT solver, and backtracking tree search.

Effect handlers. Effect handlers are a natural example for managing stacks of continuations – the handler algebra $fr \to r$, and the generator $a \to r$, where f is the signature. Of course, this requires the result type to be manipulated in the type of the handler algebra. With this fancier type, effect handlers can be encoded by using a CPS-encoded Free monad, and using the co-exponential combinators to manipulate the stack of handlers.

These applications are worked out in the supplementary material.

9 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 have the same ontological status. Continuations producing left adjoints to sums provides insights into the behavior of sums and control flow.

Axiomatics of control effects. We conflated value sums and computational sums – but this language can also be presented in fine-grained call-by-value, and then axiomatized using Freyd categories. This is a framework for studying equations for control effects, which we will pursue in future work, and also understand the status of completeness with respect to CPS semantics. The state of the art is in the work of Führmann and Thielecke [2004], and their reflection/structure theorems.

This calculus presented shows how covalues/continuations are *capabilities* – they provide an escape catch to perform control effects. This is a good fit for the purity comonads of Choudhury and Krishnaswami [2020]. Dropping capability variables from the context would block control effects, recovering pure sums from backtracking sums! This needs to be worked out in the denotational semantics.

Dual calculi. The modern view of dualities of computation is in polarised adjunction calculi and their models [Curien, Fiore, et al. 2016]. The duality exploited here shows up in their cartesian polarised structure theorems (Ex.27). Freyd categories with closed, coclosed structure can be related to Fiore's biclosed action models, which we will explore further.

In general, dual calculi for values and covalues take both cbv and cbn points of view, but ours restricts to the cbv case and exhibits a different duality. This can be understood by looking at the cbv variants of $\lambda\mu$ and $\mu\tilde{\mu}$. Selinger [2001] presents a cbv version of $\lambda\mu$ with μ binding two variables, which uses the coexponential interpretation. Other presentations of $\lambda\mu$ in cbv, such as the one by Ong and Stewart [1997], does not 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.

There are too many presentations of CPS calculi in the literature to compare against. The crucial difference is that we stick to a natural deduction style presentation – the closest cousins are Levy's JwA and Zeilberger's cbv cps calculus (which use non-returning judgements), or Harper's callcc in SML. Compared to these, we exploit coexponentials to exhibit a duality. Ariola et al. [2009] present a sequent calculus language for subtraction, but we have not added subtraction as a primitive. Instead we show it in our dual arrow calculi.

Logical aspects. This is a calculus for classical logic – provability in Gentzen's LK (without $\perp$) is equivalent to typability in $\lambda\tilde{\lambda}$. Similarly, provability in Crolard [2001]'s subtractive logic + TND is equivalent to typability in $\lambda\tilde{\lambda}$.

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 $\multimapinv$. 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] has a similar adjunction, but it is not a monad: $C \times S^A \cong C \to R^{S^{A+B}}$.

Lawvere's boundary operator. The type $\partial A = A - A$ is Lawvere's boundary operator of co-Heyting algebras. Using the subtraction type, this admits a computational interpretation – producing differential structure, for example, it admits a Leibniz rule: $\partial A \times B = \partial A \times B + A \times \partial B$.

REFERENCES

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

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.

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: 10.1007/BFb0018355.

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: 10.1145/96709.96714.

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: 10.1109/LICS.1989.39155.

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

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: 10.1007/BFb0013061.

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: 10.1017/S095679680000085X.

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

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: 10.1007/3-540-57208-2_35.

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: 10.1007/978-3-662-02880-3_8.

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: 10.1007/3-540-60164-3_28.

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: 10.1017/S0960129500001195.

Martin Hofmann and Thomas Streicher. June 29, 1997. "Continuation Models Are Universal for Lambda-Mu-Calculus." In: *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science* (LICS '97). IEEE Computer Society, USA, (June 29, 1997), 387. ISBN: 978-0-8186-7925-4.

C.-H. L. Ong and C. 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: 10.1145/263699.263722.

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.

Thomas Streicher and B. Reus. Nov. 1998. "Classical Logic, Continuation Semantics and Abstract Machines." *Journal of Functional Programming*, 8, 6, (Nov. 1998), 543–572. DOI: 10.1017/S0956796898003141.

Hayo Thielecke. June 1999. "Continuations, Functions and Jumps." *ACM SIGACT News*, 30, 2, (June 1999), 33–42. DOI: 10.1145/568547.568561.

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

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

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.

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: 10.1017/S096012950000311X.

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: 10.1006/inco.2001.2947.

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

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: 10.1007/978-94-007-0954-6_7.

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: 10.1145/944705.944723.

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: 10.1016/j.ic.2003.08.001.

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: 10.1016/j.tcs.2006.12.026.

Paul-André Melliès and Nicolas Tabareau. 2007. "Linear Continuations and Duality." (2007). Retrieved Dec. 19, 2022 from https://hal.archives-ouvertes.fr/hal-00339156.

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: 10.1007/s10990-007-9006-0.

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

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

Jean-Yves Girard. Sept. 25, 2011. *The Blind Spot: Lectures on Logic*. (1st ed.). EMS Press, (Sept. 25, 2011). ISBN: 978-3-03719-088-3 978-3-03719-588-8. DOI: 10.4171/088.

Samson Abramsky. Mar. 19, 2012. *No-Cloning In Categorical Quantum Mechanics*. (Mar. 19, 2012). arXiv: 0910.2401 [quant-ph]. Retrieved June 1, 2023 from http://arxiv.org/abs/0910.2401. preprint.

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: 10.1145/2837614.2837652.

Harley Eades III and Gianluigi Bellin. Aug. 19, 2017. *A Cointuitionistic Adjoint Logic*. (Aug. 19, 2017). arXiv: 1708.05896 [cs]. Retrieved Jan. 3, 2023 from http://arxiv.org/abs/1708.05896. preprint.

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: 10.1016/j.apal.2016.10.008.

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: 10.1145/3408993.

A SUPPLEMENTARY MATERIAL FOR SECTION 1 (INTRODUCTION)

For logicians: the duality is in the symmetry of these two logical equivalences:

$$\frac{\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\,{\rightarrow}I \qquad \Gamma \vdash A}{\Gamma \vdash B}\,{\rightarrow}E \qquad\qquad \frac{\dfrac{\Gamma, \neg A \vdash \neg\neg B}{\Gamma \vdash \neg\neg(A \vee B)} \qquad \Gamma \vdash \neg A}{\Gamma \vdash \neg\neg B}$$

The second derivation holds in intuitionistic logic, which crucially depends on the encoding of $\neg A$ as $A \rightarrow \bot$.

$$\frac{\dfrac{\dfrac{\overline{\Gamma, A \vdash A \vee B}}{\Gamma, \neg(A \vee B), A \vdash A \vee B} \qquad \dfrac{\overline{\Gamma, \neg(A \vee B) \vdash \neg(A \vee B)}}{\Gamma, \neg(A \vee B), A \vdash \neg(A \vee B)}}{\Gamma, \neg(A \vee B), A \vdash \bot}}{\Gamma, \neg(A \vee B) \vdash \neg A} \qquad \frac{\dfrac{\dfrac{\overline{\Gamma, B \vdash A \vee B}}{\Gamma, \neg(A \vee B), B \vdash A \vee B} \qquad \dfrac{\overline{\Gamma, \neg(A \vee B) \vdash \neg(A \vee B)}}{\Gamma, \neg(A \vee B), B \vdash \neg(A \vee B)}}{\Gamma, \neg(A \vee B), B \vdash \bot}}{\Gamma, \neg(A \vee B) \vdash \neg B}$$

$$\frac{\dfrac{\dfrac{\Gamma, \neg A \vdash \neg\neg B \qquad \overline{\Gamma, \neg(A \vee B) \vdash \neg A}}{\Gamma, \neg(A \vee B) \vdash \neg\neg B} \qquad \overline{\Gamma, \neg(A \vee B) \vdash \neg B}}{\Gamma, \neg(A \vee B) \vdash \bot}}{\Gamma \vdash \neg\neg(A \vee B)}$$

In type theory and category theory, these are the two isomorphisms:

$$C \times A \rightarrow B \cong C \rightarrow B^A \qquad\qquad C \times R^A \rightarrow R^{R^B} \cong C \rightarrow R^{R^{A+B}}$$

or in terms of adjunctions:

$$(-) \times A \dashv (-)^A \qquad\qquad (-) \times R^A \dashv A + (-)$$

where the first adjunction lives in a cartesian closed category of values, and the second adjunction lives in a cocartesian coclosed category of computations: the Kleisli category of the double negation, or double dualization, or continuation monad.

B SUPPLEMENTARY MATERIAL FOR SECTION 2 (DUALITY BY EXAMPLE)

Using case, we can define the familiar `callcc` control operator:

```
fun callcc (f : co a → a) : a =
    let (s : a + a) = cofn (k : co a) ⇒ f k
    in case s of
         INL a ⇒ a
       | INR a ⇒ a
```

This definition highlights the duplicating nature of `callcc` – if it weren't a sum type, we wouldn't have `case`. So far, we have seen values and covalues, and they live in harmony, by the use of functions and cofunctions, and they don't interact. To make them interact we need a `throw` operation, which we can define using coapplication:

```
fun throw (x : a) (k : co a) : b = (INL x) @ k
```

To a continuations afficionado, these operators are familiar, except we have a primitive type `co a` for covalues (which are, not surprisingly, continuations), and we can program with them recovering the computational interpretation of classical logic. Tertium Non Datur (or the Law of the Excluded

Middle) is the identity cofunction, as we have already seen, saying that any type can produce a value or a covalue out of nothing, with no third possibility. Exploiting `tnd` and `case`, we can perform double negation introduction:

```
fun tnd () : a + co a = cofn (k : co a) ⇒ k
fun dni (x : a) : co (co a) =
    case tnd () of
        INL ka ⇒ throw x ka
      | INR kka ⇒ kka
```

Crucially, note that we are working with a primitive notion of negation, the `co a` type, instead of negation as a function `a → 0`. The self-adjointness of negation is fundamental, and is captured by the following term:

```
fun adj (f : co a → b) : co b → a =
    let val (s : a + b) = cofn (k : co a) ⇒ f k in
        let val (t : b + a) = case s of INL a ⇒ INR a | INR b ⇒ INL b in
            fn (kb : co b) ⇒ t @ kb
        end
    end
```

Using this, we can define double negation elimination, and the contravariance of negation:

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

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

```
fun lam (f : a → b) : co a + b =
  cofn kka ⇒ f (dne kka)
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 ((ka , kb) : co a * co b) : co (a + b) =
  contramap (fn e ⇒ e @ ka) kb
fun deMorgan3 (kp : co (a * b)) : co a + co b =
  adj (fn ks ⇒ let val (kka, kkb) = deMorgan1 ks in (dne kka, dne kkb) 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 kb ⇒ throw b kb)
  | INR kp ⇒ kp
```

The subtraction type is the 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 kkb ⇒ dne kkb
```

Finally, Peirce's law, which is the type of the general `callCC` can be derived as follows:

```
fun peirce (f : ('a → 'b) → 'a) : 'a =
  case lam f of
    INL kg ⇒ #1 (adj ctof kg)
  | INR a ⇒ a
```

This highlights how the covalue for `b` is dropped.

C SUPPLEMENTARY MATERIAL FOR SECTION 4 (SEMANTICS)

C.1 Other Co-exponentials

As an aside, we give some other examples of co-exponentials.

A Heyting algebra is a bounded lattice with an implication operator $\rightarrow$, such that, $c \wedge a \leqslant b$ iff $c \leqslant a \rightarrow b$, making $a \rightarrow b$ the relative pseudo-complement of a with respect to b. This can equivalently be described as a poset (thin category) with finite products (meets), finite coproducts (joins), which is cartesian closed. Since product functors are left adjoints, they preserve coproducts, hence meets distribute over joins, making them a distribute lattice.

The dual of a Heyting algebra is a co-Heyting algebra: it has finite meets and joins, and a co-implication operator $\backslash$, such that $a \leqslant b \vee c$ iff $a \setminus b \leqslant c$. This can equivalently be described as a poset with finite products (meets), finite coproducts (joins), which is co-cartesian co-closed. Any Heyting algebra can be turned into a co-Heyting algebra by inverting the poset ordering. If a lattice carries both Heyting and co-Heyting structures, it s a bi-Heyting algebra.

Consider the vertical natural numbers $\mathbb{N} \cup \{\omega\}$, with $\{0 \leqslant 1 \leqslant 2 \leqslant \ldots \leqslant \omega\}$. This is an example of a bi-Heyting algebra.

(1) $m \wedge n \triangleq \min(m, n)$, bounded by ω.
(2) $m \vee n \triangleq \max(m, n)$, bounded by 0.
(3) $m \rightarrow n \triangleq \max\{\, p \mid \min(p, m) \leqslant n \,\}$.
(4) $m \setminus n \triangleq \min\{\, p \mid m \leqslant \max(n, p) \,\}$.

Every Boolean algebra gives a bi-Heyting algebra, where $a \rightarrow b \triangleq \neg a \vee b$ and $a \setminus b \triangleq a \wedge \neg b$. As an example, consider the powerset lattice $\mathfrak{P}(C)$ of any set C, ordered by $\subseteq$.

(1) $X \wedge Y \triangleq X \cap Y$, bounded by C.
(2) $X \vee Y \triangleq X \cup Y$, bounded by $\emptyset$.
(3) $X \rightarrow Y \triangleq X^c \cup Y$.
(4) $X \setminus Y \triangleq X \cap Y^c$.

More generally, the subobject classifier of any presheaf category $\mathsf{PSh}(C)$ for any small category C is a bi-Heyting algebra (Johnstone). For any topological space, the lattice of open subsets is a Heyting algebra and the lattice of closed subsets is a co-Heyting algebra. Also see Brouwerian algebras, semi-Boolean algebras (Rauszer).

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \approx e : A}\ \textsc{refl} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A}{\Gamma \vdash e_2 \approx e_1 : A}\ \textsc{sym} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \qquad \Gamma \vdash e_2 \approx e_3 : A}{\Gamma \vdash e_1 \approx e_3 : A}\ \textsc{trans}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A \times B}{\Gamma \vdash \mathsf{fst}(e_1) \approx \mathsf{fst}(e_2) : A}\ \mathsf{fst}()\text{-}\textsc{cong} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \times B}{\Gamma \vdash \mathsf{snd}(e_1) \approx \mathsf{snd}(e_2) : B}\ \mathsf{snd}()\text{-}\textsc{cong}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A \qquad \Gamma \vdash e_3 \approx e_4 : B}{\Gamma \vdash (e_1, e_3) \approx (e_2, e_4) : A \times B}\ \textsc{pair-cong}$$

$$\frac{\Gamma, x : A \vdash e_1 \approx e_2 : B}{\Gamma \vdash \lambda x.\, e_1 \approx \lambda x.\, e_2 : A \Rightarrow B}\ \lambda\text{-}\textsc{cong} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \Rightarrow B \qquad \Gamma \vdash e_3 \approx e_4 : A}{\Gamma \vdash e_1\ e_3 \approx e_2\ e_4 : B}\ \textsc{app-cong}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A}{\Gamma \vdash \mathsf{inl}(e_1) \approx \mathsf{inl}(e_2) : A + B}\ \mathsf{inl}\text{-}\textsc{cong} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : B}{\Gamma \vdash \mathsf{inr}(e_1) \approx \mathsf{inr}(e_2) : A + B}\ \mathsf{inr}\text{-}\textsc{cong}$$

$$\frac{\Gamma \vdash e_1 \approx e_2 : A + B \qquad \Gamma, x : A \vdash e_3 \approx e_4 : C \qquad \Gamma, y : B \vdash e_5 \approx e_6 : C}{\Gamma \vdash \mathsf{case}(e_1, x.\, e_3, y.\, e_5) \approx \mathsf{case}(e_2, x.\, e_4, y.\, e_6) : C}\ \textsc{case-cong}$$

$$\frac{\Gamma, x : \tilde{A} \vdash e_1 \approx e_2 : B}{\Gamma \vdash \widetilde{\lambda} x.\, e_1 \approx \widetilde{\lambda} x.\, e_2 : A \Rightarrow B}\ \widetilde{\lambda}\text{-}\textsc{cong} \qquad \frac{\Gamma \vdash e_1 \approx e_2 : A \Rightarrow B \qquad \Gamma \vdash e_3 \approx e_4 : B}{\Gamma \vdash \widetilde{e_1\ e_3} \approx \widetilde{e_2\ e_4} : A}\ \textsc{coapp-cong}$$

(a) Equivalence and congruence rules for the equational theory of $\lambda\tilde{\lambda}$

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

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

Fig. 12. Backtracking in $\lambda\tilde{\lambda}$ (derivable)

However, all these examples are posets, and there is no computational content if we choose to use this as a denotational semantics. Instead we have a given a construction of a co-cartesian co-closed category, starting from a bi-cartesian closed category, using the continuation, or double dualization monad (Kock). Since we do not have the cartesian closure and co-cartesian co-closure on the same category, we do not encounter the problem of degeneracy. This means we still retain computational content, as we will see in later sections, by giving an equational theory.

D SUPPLEMENTARY MATERIAL FOR SECTION 6 (EQUATIONAL THEORY)

From the $\widetilde{\lambda}\eta$ rule, we can derive the two backtracking rules in figure 12.