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 2001], (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:

\[(C \times A) \rightarrow B \cong B \rightarrow (C \Rightarrow B)\]  

(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 \(\text{eval}_{A,B}: (A \Rightarrow B) \times A \rightarrow B\) 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:

\[(A \Leftarrow B) \rightarrow C \cong C \rightarrow (A \Rightarrow B)\]  

(2)

Continuing the analogy with currying, the dual type \(A \Leftarrow B\) (or coexponential object) should come with a (universal) coevaluation function \(\text{coeval}_{A,B}: B \rightarrow A + (A \Leftarrow B)\). 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:

\[(-) \times A + (-)^A\]  

\[(-)^A + (-) + A\]  

(3)
Exponential objects \((-)^A\) give right adjoints to product functors \((-) \times A\), and coexponential objects \((-)^A\) give left adjoints to coproduct functors \(A + (-)\). If \(C\) is a cartesian closed category (a model for the simply-typed lambda-calculus), then formally \(C^{\text{op}}\) 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:

\[
\begin{align*}
0 \times A & \cong 0 & A + 1 & \cong 1
\end{align*}
\]

In logic, the first isomorphism is the tautology \(\bot \land A \leftrightarrow \bot\), and the second isomorphism is \(\top \lor A \leftrightarrow \top\), 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 – \(\text{Bool} \cong 1 + 1 \cong 1\), leading to a degenerate language.

\[
\begin{tikzpicture}[->,shorten >=1pt,auto, thick]
\node (A) at (0,1) {$A$};
\node (1) at (-1,0) {$1$};
\node (1+1) at (1,0) {$1 + 1$};
\node (f) at (0,-1) {$f$};
\node (g) at (0,0) {$g$};
\node (fg) at (0,-0.5) {$[f, g]$};
\path (f) edge (1)
      (f) edge (1+1)
      (g) edge (1+1)
      (fg) edge (A)
      (fg) edge (1+1)
;\end{tikzpicture}
\]

Since \(1 + 1 \cong 1\), it is a terminal object, making \(t_1, t_2 : 1 \rightarrow 1 + 1\) equal. If \(f, g : 1 \rightarrow A\) are any two closed programs, then \(f = [f, g] \circ t_1 = [f, g] \circ t_2 = g\). 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 \(\mu\mu\)). 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 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 $\lambda$-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 $\lambda$ and $\tilde{\lambda}$!

- We present a $\lambda\tilde{\lambda}$ calculus, which exhibits two dual abstraction mechanisms: $\lambda$ 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 $\lambda$ 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 $\text{fn}$ bind values, and cofunctions $\text{cofn}$ bind covalues. Covalues have $\text{co}$ types, and $\text{cofn}$ 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
  ⇝ 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
  ⇝ 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
```

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

\[
\begin{align*}
mult\ [1, 2, 0, 3, 4] & \equiv 1 \times (2 \times (0 \times (3 \times (4 \times 1)))) \\
& \equiv 1 \times (2 \times (0 \times (3 \times 4))) \\
& \equiv 1 \times (2 \times (0 \times 12)) \\
& \equiv 1 \times (2 \times 0) \\
& \equiv 1 \times 0 \\
& \equiv 0
\end{align*}
\]

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

\[
\begin{align*}
\text{fun } \text{mult} \ (l : \text{list int}) : \text{int} &= \\
\text{let fun } \text{loop} \ [\] = 1 \\
\text{| loop} \ (0 :: _) = 0 \\
\text{| loop} \ (h :: t) = h \times \text{loop} \ t \\
\text{in } \text{loop} \ l
\end{align*}
\]

which proceeds as:

\[
\begin{align*}
mult\ [1, 2, 0, 3, 4] & \equiv 1 \times (2 \times 0) \\
& \equiv 1 \times 0 \\
& \equiv 0
\end{align*}
\]

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.

\[
\begin{align*}
\text{fun } \text{mult} \ (l : \text{list int}) : \text{int} + \text{int} &= \\
\text{let fun } \text{loop} \ [\] = \text{INR} \ 1 \\
\text{| loop} \ (0 :: _) = \text{INL} \ 0 \\
\text{| loop} \ (h :: t) = \text{trace} \ ("at " ^ \text{Int.toString} \ h) \\
\quad (\text{mapRight} \ (\text{fn} \ x \Rightarrow h \times x) \ (\text{loop} \ t)) \\
\text{in } \text{loop} \ l
\end{align*}
\]

This computes as:

\[
\begin{align*}
mult\ [1, 2, 0, 3, 4] & \equiv \text{mapRight} \ (\text{fn} \ x \Rightarrow 1 \times x) \ (\text{mapRight} \ (\text{fn} \ x \Rightarrow 2 \times x) \ (\text{loop} \ [0, 3, 4])) \\
& \equiv \text{mapRight} \ (\text{fn} \ x \Rightarrow 1 \times x) \ (\text{mapRight} \ (\text{fn} \ x \Rightarrow 2 \times x) \ (\text{INL} \ 0)) \\
& \equiv \text{mapRight} \ (\text{fn} \ x \Rightarrow 1 \times x) \ (\text{INL} \ 0) \\
& \equiv \text{INL} \ 0
\end{align*}
\]

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.

```haskell
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 \( k : \text{int co} \), 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 \((\text{int})\) argument" to the left branch. The `loop` function is the same as before, except when it hits a 0, it coapplies \( \text{INL} \ 0 \) to the bound covalue \( k \), jumping to the left branch and exiting the program. This
computes as:

```haskell
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 \( \text{fn} \ (x : \text{int}) ⇒ 0 \), when applied to any argu-
ment, will always return the value 0. Similarly, we have constant cofunctions, like \( \text{cofn} \ (k : \text{co int}) ⇒ 0 \),
which are right-biased sums — but unlike functions (lambdas), they aren’t frozen thunks — for ex-
ample, constant cofunctions collapse and reduce to an ordinary right sum.

```haskell
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:

```haskell
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 \( a \rightarrow b \), and is defined as a
product of a value and a covalue:

```haskell
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 \texttt{ftoc} and \texttt{ctof}):

\begin{verbatim}
fun curry (f : (c * a) \rightarrow b) = fn x \Rightarrow fn y \Rightarrow f (x, y)

fun uncurry (f : c \rightarrow (a \rightarrow b)) = fn (x, y) \Rightarrow f x y

fun cocurry (f : c \rightarrow a + b) : (c - a) \rightarrow b = fn (c, k) \Rightarrow (f c) @ k

fun coun curry (f : (c - a) \rightarrow b) : c \rightarrow a + b = fn c \Rightarrow cofn k \Rightarrow f (c, k)
\end{verbatim}

Subtraction and sums interplay in the following way:

\begin{verbatim}
fun coeval (x : a) : b + (a - b) = cofn (k : co b) \Rightarrow (x, k)

fun coun eval (f : (b + a) - b) : a = (#1 f) @ (#2 f)
\end{verbatim}

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:

\begin{verbatim}
fun cocompose (f : a - c) : (b - c) + (a - b) =
  case coeval (#1 f)
    of INL b \Rightarrow INL (b, #2 f)
    | INR (a, k) \Rightarrow INR (a, k)
\end{verbatim}

\textbf{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 \textbf{throw} – lifting the value to the left, then coapplying the covalue, producing \(b\) out of nothing.

\begin{verbatim}
fun throw (p : a - a) : b = (INL (#1 p)) @ k
\end{verbatim}

Dually(!), the sum type \(a + a\) can be collapsed to an \(a\):

\begin{verbatim}
fun codiag (s : a + a) : a = case s of INL a \Rightarrow a | INR a \Rightarrow a
\end{verbatim}

To a continuations afficionado, these operators will look familiar:

\begin{verbatim}
fun callcc (f : co a \rightarrow a) : a = codiag (cofn (k : co a) \Rightarrow f k)

fun call/cc (f : (a \rightarrow b) \rightarrow a) : a =
  codiag (cofn (k : co a) \Rightarrow f (fn a \Rightarrow throw (a, k)))
\end{verbatim}

\section{Syntax}

We present a formal calculus called \(\lambda\bar{\lambda}\) which exhibits abstraction and coabstraction.

\subsection{Typing}

The syntax and typing of \(\lambda\bar{\lambda}\) 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 \(\bar{A}\), 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 \textit{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 \textit{rightist} typing rules. Sums have the usual \textit{leftist} typing rules, with two injections, and a case construct. As is standard, functions are \textit{rightist} – they are introduced by lambda abstraction, binding a value \(x : A\) in the body \(e : B\), producing a term of type \(A \multimap B\). Application eliminates a function, which applies a function \(A \multimap B\) to an \(A\), producing a \(B\).

Now we add two more rules which look completely symmetric – colambda binds a covalue \(\tilde{x} : \tilde{A}\) 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 \textit{rightist} rule for sums. Sums have \textit{bipartisan} status – the interaction of leftist case \(\tilde{\lambda}\) and rightist \(\hat{\lambda}\) 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 covales 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 \textit{Theorem 3.1}.

\textbf{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 \(\Gamma \vdash \theta(e) : A\).

\textbf{Definition 3.2 (Substitution on variables).}

\[
\theta[x] \triangleq \begin{cases} \\
\begin{array}{c}
\phi \quad \theta = \langle \rangle \\
\phi \qquad \theta = \langle \psi, e/x \rangle \\
\phi[x] \quad \theta = \langle \phi, e/y \rangle, x \neq y
\end{array}
\end{cases}
\]
The Duality of Abstraction

\[ A, B ::= 1 | A \times B | A + B | A \Rightarrow B | \tilde{A} \]

**Terms**
\[ e ::= \ast | (e_1, e_2) | \text{fst}(e) | \text{snd}(e) | \text{inl}(e) | \text{inr}(e) | \text{case}(e_1, x, e_2, y, e_3) \]
\[ | x | \lambda(x : A). e | e_1 e_2 | \tilde{\lambda}(x : \tilde{A}). e | e_1 \overline{e_2} \]

**Values**
\[ v ::= \ast | (v_1, v_2) | \text{inl}(v) | \text{inr}(v) | x | \lambda(x : A). e \]

**Contexts**
\[ \Gamma, \Delta, \Psi ::= \cdot | \Gamma, x : A \]

**Substitutions**
\[ \theta, \phi ::= \langle \rangle | \langle \theta, v \rangle / x \]

(a) Grammar for \( \tilde{\lambda} \)
\[ x : A \in \Gamma \quad x \text{ is a variable of type } A \text{ in context } \Gamma \]
\[ \Gamma \supseteq \Delta \quad \Gamma \text{ is a weakening of } \Delta \]
\[ \Gamma \vdash \theta : \Delta \quad \theta \text{ is a substitution from } \Gamma \text{ to } \Delta \]
\[ \Gamma \vdash e : A \quad e \text{ is an expression of type } A \text{ in context } \Gamma \]
\[ \Gamma \vdash e_1 \approx e_2 : A \quad e_1 \text{ and } e_2 \text{ are equal expressions of type } A \text{ in context } \Gamma \]

(b) Judgements for \( \tilde{\lambda} \)
\[ \frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B} \times I \]
\[ \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \text{fst}(e) : A} \times E_1 \]
\[ \frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \text{snd}(e) : B} \times E_2 \]
\[ \frac{\Gamma \vdash e : A}{\Gamma \vdash \text{inl}(e) : A + B} + I_1 \]
\[ \frac{\Gamma \vdash e : B}{\Gamma \vdash \text{inr}(e) : A + B} + I_2 \]
\[ \frac{\Gamma \vdash e_1 : A + B \quad \Gamma, x : e_2 \vdash A : C \quad \Gamma, y : e_3 \vdash B : C}{\Gamma \vdash \text{case}(e_1, x, e_2, y, e_3) : C} + E \]
\[ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ VAR} \]
\[ \frac{\Gamma \vdash \lambda(x : A). e : A \Rightarrow B}{\Gamma \vdash e_1 : A \Rightarrow B \quad \Gamma \vdash e_2 : A} \Rightarrow E \]
\[ \frac{\Gamma \vdash e_1 : e_2 : B}{\Gamma \vdash \lambda(x : A). e : A + B} \Rightarrow I \]
\[ \frac{\Gamma \vdash e_1 : A + B \quad \Gamma \vdash e_2 : \tilde{A}}{\Gamma \vdash \lambda(x : \tilde{A}). e : \tilde{A} + B} \Rightarrow E \]

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

Fig. 1. Syntax and typing for \( \tilde{\lambda} \)
\(\frac{}{x : A \in \Gamma} \quad \frac{x : A \in \Gamma}{x : A \in \Gamma, x : A} \quad \frac{x : A \in \Gamma, x : A}{x : A \in (\Gamma, x : A)} \quad \frac{\epsilon \cdot \text{id}}{x : A \in \Gamma, x : A \in \Delta} \quad \frac{\epsilon \cdot \text{cong}}{x : A \in \Gamma, x : A \in \Delta} \quad \frac{\epsilon \cdot \text{wk}}{x : A \in \Gamma, x : A \in \Delta} \)

(a) Context Membership Rules

\(\frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta, x : A} \quad \frac{\Gamma \supseteq \Delta}{\Gamma, x : A \supseteq \Delta, x : A} \)

(b) Weakening Rules

\(\frac{\Gamma \vdash \langle \rangle : \cdot}{\Gamma \vdash \langle \rangle : \cdot} \quad \frac{\Gamma \vdash \delta : \Delta}{\Gamma \vdash \langle \delta, \gamma/x \rangle : \Delta, x : A} \quad \frac{\Gamma \vdash v : A}{\Gamma \vdash \langle \delta, \gamma/x \rangle : \Delta, x : A} \)

(c) Substitution Rules

\(\theta(\ast) \triangleq \ast \)
\(\theta((e_1, e_2)) \triangleq (\theta(e_1), \theta(e_2)) \)
\(\theta(\text{fst}(e)) \triangleq \text{fst}(\theta(e)) \)
\(\theta(\text{snd}(e)) \triangleq \text{snd}(\theta(e)) \)
\(\theta(x) \triangleq \theta[x] \)
\(\theta(\lambda x. e) \triangleq \lambda y. (\theta(y/x))(e) \)
\(\theta(e_1 e_2) \triangleq \theta(e_1) \theta(e_2) \)
\(\theta(\text{inl}(e)) \triangleq \text{inl}(\theta(e)) \)
\(\theta(\text{inr}(e)) \triangleq \text{inr}(\theta(e)) \)
\(\theta(\text{case}(e, x, e_1, y, e_2)) \triangleq \text{case}(\theta(e), x, \theta(y/x)(e_1), x, \theta(y/x)(e_2)) \)
\(\theta(\text{case}(e, x, e_1, y, e_2)) \triangleq \text{case}(\theta(e), x, \theta(y/x)(e_1), x, \theta(y/x)(e_2)) \)

\(\theta(\text{case}(e_1, e_2)) \triangleq \theta(e_1) \theta(e_2) \)

Definition 3.3 (Substitution on raw terms).

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:

\(\langle - \rangle^\Gamma_A : \lambda \lambda A \rightarrow (A \rightarrow R) \rightarrow R\)

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 \(\tilde{\Lambda}\) type is translated as \(A \rightarrow R\), and the function type \(A \Rightarrow B\) as \(A \rightarrow (B \rightarrow R) \rightarrow R\), as is standard.

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

### 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.

#### Dual to functions

Dual to functions, the continuation of a colambda \( \tilde{\lambda}x. e \) is \( (A + B) \rightarrow R \), which we can (crucially) split into two continuations \( k_A : A \rightarrow R \) and \( k_B : B \rightarrow R \). We pass the continuation \( k_A \) into the environment \( \gamma \), and evaluate the body \( e \) in this extended environment, continuing with \( k_B \). To translate a coapplication \( e_1. e_2 : B \), we grab a continuation \( k_B \), then evaluate the argument \( e_2 \), which requires a continuation \( (A + B) \rightarrow R \). We pass a continuation which binds the covalue (or continuation) \( k_A : A \rightarrow R \), 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 \( \text{inl}(a) \), the computation continues as \( k_A \) applied to \( a \) - when it receives an \( \text{inr}(b) \), 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:

\[
\Gamma \times (A \rightarrow R) \rightarrow ((B \rightarrow R) \rightarrow R) \equiv \Gamma \rightarrow ((A + B) \rightarrow R) \rightarrow R,
\]

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

The Duality of Abstraction  
POPL’24, January 17–19, 2024, London, UK

\[
\begin{align*}
\langle \star \rangle^r_A & \triangleq \lambda k. \ k \star \\
\langle e_1, e_2 \rangle^r_{A \times B} & \triangleq \lambda k. \ (\langle e_2 \rangle^r_B(\lambda b. \ (\langle e_1 \rangle^r_A(\lambda a. \ k(a, b)))) \\
\langle \text{fst}(e) \rangle^r_A & \triangleq \lambda k. \ (\langle e \rangle^r_{A \times B}(\lambda p. \ k \ \text{fst}(p))) \\
\langle \text{snd}(e) \rangle^r_A & \triangleq \lambda k. \ (\langle e \rangle^r_{A \times B}(\lambda p. \ k \ \text{snd}(p))) \\
\langle \text{inl}(e) \rangle^r_{A + B} & \triangleq \lambda k. \ (\langle e \rangle^r_A(\lambda a. \ k \ \text{inl}(a))) \\
\langle \text{inr}(e) \rangle^r_{A + B} & \triangleq \lambda k. \ (\langle e \rangle^r_B(\lambda b. \ k \ \text{inr}(b))) \\
\langle \text{case}(e, x, e_1, e_2) \rangle^r_C & \triangleq \lambda k. \ (\langle e \rangle^r_{A + B}(\lambda \left(\begin{array}{l}
\text{inl}(a). \ (\langle e_1 \rangle^r_C a k) \\
\text{inr}(b). \ (\langle e_2 \rangle^r_C b k)
\end{array}\right)))
\end{align*}
\]

Fig. 3. Continuation semantics for \( \tilde{\lambda} \)
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 \( \check{A} \) 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 \( \neg : \mathcal{C}^{\text{op}} \to \mathcal{C} \), that is self-adjoint on the right.

This means, for any objects \( A, B \in \mathcal{C} \), we have the hom-set isomorphism:

\[
\mathcal{C}(B, \neg A) \cong \mathcal{C}(A, \neg B).
\]

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

With this situation in mind, we observe the following:

**Proposition 4.1.**

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

**Proof.** We have that, \( \mathcal{C}^{\text{op}}(\neg \neg A, B) \cong \mathcal{C}(B, \neg \neg A) \cong \mathcal{C}(\neg A, \neg B) \cong \mathcal{C}_{\neg}(A, B) \cong \mathcal{C}_{\neg}(\neg A, \neg B) \), making \( \neg_{\text{bo}} \) a right adjoint. And, \( \mathcal{C}_{\neg}(A, B) = \mathcal{C}(\neg A, \neg B) \cong \mathcal{C}(B, \neg \neg A) \equiv \mathcal{C}^{\text{op}}_{\neg}(A, B) \). \( \square \)

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

**Proposition 4.3.** If \( \mathcal{C} \) is bicartesian closed with a fixed object \( R \) (the object of responses),

1. \( \neg \equiv R^{(\neg)} \) 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. \( \neg_{\text{ff}} \) 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 \( R^{RX} \)
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 \( \lambda \lambda \).
Following Taylor [2002], we write \( R^2X \) for \( R^{RX} \).

**Definition 4.4.**

\[
\begin{align*}
K : \mathcal{C} & \to \mathcal{C} \\
X & \mapsto R^2X \\
X \xrightarrow{f} Y & \mapsto K(f) : R^2X \to R^2Y \\
K & \mapsto \lambda (k : R^X). k'(k \circ f)
\end{align*}
\]

The monad structure is given by:

**Definition 4.5.**

\[
\begin{align*}
\eta_X : X & \to K(X) \\
x & \mapsto \lambda (k : R^X). k(x) \\
\mu_X : K^2(X) & \to K(X) \\
k' & \mapsto \lambda (k : R^X). k'((\lambda (k' : R^2X). k''(k)))
\end{align*}
\]

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

**Definition 4.6.**

\[
\begin{align*}
\tau_{X,Y} : X \times KY & \to K(X \times Y) \\
(x, k') & \mapsto K(\lambda (y : Y). (x, y))(k') \\
\sigma_{X,Y} : KKX \times Y & \to K(X \times Y) \\
(k', y) & \mapsto K(\lambda (x : X). (x, y))(k')
\end{align*}
\]

The continuation monad \( K \) is not commutative because there are two ways to go from \( KKX \times KY \to K(X \times Y) \) 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.**

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

Given a Kleisli arrow \( f : X \to KY \), its lift is \( f^\dagger : KX \xrightarrow{KF} K^2Y \xrightarrow{\nu_Y} KY \). 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:

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

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

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

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

\( \mathcal{C}_K(Z \times RX, Y) \equiv \mathcal{C}(Z \times RX, R^Y) \equiv \mathcal{C}(Z \times RX \times R^Y, R) \equiv \mathcal{C}(Z \times RX^Y, R) \equiv \mathcal{C}(Z, R^{RX^Y}) \equiv \mathcal{C}_K(Z, X+Y) \).
Observe that \((-) \times R^X\) is an endofunctor on \(C_K\), since:
\[
Y \xrightarrow{f} KZ \leftrightarrow Y \times R^X \xrightarrow{f \times R^X} KZ \times R^X \xrightarrow{\sigma_{Z,R^X}} K(Z \times R^X).
\]
This means, we have the following adjoint situation in \(C_K\), 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 \(f : C \times A \to B\), the currying of \(f\) is \(f^\# : C \to B^A\).
- Given \(g : C \to B^A\), the uncurrying of \(g\) is \(g^\# : C \times A \to B\).
- Evaluation is \(\text{ev}_{A,B} : B^A \times A \to B \triangleq \text{id}_{B^A}\).
- The co-exponential of \(B\) by \(A\) is written as \(A^B\).
- Given \(f : B \to A + C\), the co-currying of \(f\) is \(f^\flat : A^B \to C\).
- Given \(g : A^B \to C\), the co-uncurrying of \(g\) is \(g^\flat : B \to A + C\).
- Co-evaluation is \(\text{coev}_{A,B} : B \to A + A^B \triangleq \text{id}_{A^B}\).

4.3 The Indiana Control Operators

The conceptual understanding of the negation functor and the interplay of sums and products in \(C_R\), \(C_K\) 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 \(C_K\), and further \(K(0) = R^20 \equiv R^1 \equiv R\). Note that, 0 is not a strict initial object, meaning that, having an arrow \(A \to 0\) does not imply \(A \equiv 0\). We can have non-trivial arrows to 0, and these are indeed continuations of \(A\) in \(C_K\). The \(R\) object in \(C_K\) enjoys a special status, producing control operators.

The Indiana control operators (Felleisen’s \(A\) and \(C\)) 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]).

\[
C_A \triangleq R^{R^A} \xrightarrow{\sim} R^{R^A}
\]

\[
A_A \triangleq (A \xrightarrow{\eta_A} A + B \xrightarrow{\eta_{A+B}} K(A + B)) : A \times R^A \to K(B)
\]

\[
tnd_A \triangleq (\Gamma \times R^A \xrightarrow{\pi_1} R^A \xrightarrow{\eta_{R^A}} K(R^A)) : \Gamma \to K(A + R^A)
\]

\[
callcc_A (\Gamma \times R^A \xrightarrow{f} K(A)) \triangleq \Gamma \xrightarrow{\sharp} K(A + A) \xrightarrow{K\eta_A} K(A)
\]

\[
call/cc_{A,B} (\Gamma \times (R^B \times A) \xrightarrow{f} K(A)) \triangleq \Gamma \xrightarrow{f^\flat} K(R^B \times A + A) \xrightarrow{K[\pi_2,1_A]} K(A)
\]

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

\[
K(A + B) \xrightarrow{K[f,g]} KC
\]

\[
R^{R^A \times R^B} \xrightarrow{R^{(B,R^A)}} R^{R^C}
\]
We now give an interpretation for which is to be understood as an axiomatic theory generated by an operational semantics. On top of with the expression interpretation, which we use to prove semantic weakening and substitution, the purpose of giving a categorical semantics is to produce an equational theory for the language – interaction with sums. We design equations for control effects in \( \lambda \tilde{\lambda} \) [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

\[
\begin{align*}
\begin{array}{ll}
[1] & \triangleq 1 \\
[0] & \triangleq 0 \\
\mathcal{A} \Rightarrow \mathcal{B} & \triangleq (K[\mathcal{B}])[\mathcal{A}], \\
\mathcal{A} & \triangleq R[\mathcal{A}], \\
\end{array}
\end{align*}
\]

(a) \([A] : \text{Obj}\)

\[
\begin{align*}
\begin{array}{ll}
\mathcal{A} \times \mathcal{B} & \triangleq \mathcal{A} \times \mathcal{B}, \\
\mathcal{A} + \mathcal{B} & \triangleq \mathcal{A} + \mathcal{B}, \\
\cdot & \triangleq 1
\end{array}
\end{align*}
\]

(b) \([\Gamma] : \text{Obj}\)

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 \( \Gamma \vdash \sigma : A \) = \( \Gamma \vdash \sigma : A \) \( \eta_A \).
- If \( \Gamma \supseteq \Delta \) and \( \Delta \vdash e : A \), then \( \Gamma \vdash \theta : A \) = \( \Gamma \vdash \theta : A \) \( \eta_A \).

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 \( (e)_\mathcal{A} (k) = \Gamma \vdash e : A (\gamma, k) \), for any \( \gamma \in \Gamma \) and \( k \in \mathcal{A} \rightarrow R \).

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
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 $E = E_{\leftarrow}$ $e \ E$ $\mid e \ E$ $\mid e \ E \mid e \ E$ $\mid \text{snd}(E)$ $\mid (e, E) \mid (E, v)$. We remark that these operators have better types (than calcc/abort) – better types give better equations!

$\lambda$-CONST is the constant rule – when the covalue is not used, it’s a right-biased sum (cf. Selinger’s letname). The next two rules are the interaction of normal sums and cofunctions. $\lambda$-inr-pass is like transparent passthrough, or that inr produces normal return – if the covalue was created by $\lambda$ but then used with inr, it might as well have never been created. $\lambda$-inl-jump is the interesting control effect, it is a non-local “jump with argument” (see Thielecke 1999; Levy 2003), where we throw to
If \( \Gamma \vdash e_1 \approx e_2 : A \), then \( \llbracket \Gamma \vdash e_1 : A \rrbracket = \llbracket \Gamma \vdash e_2 : A \rrbracket \).
\[
\begin{align*}
\Gamma \vdash v : 1 \\
\Gamma \vdash v \approx \star : 1 \quad 1\eta \\
\Gamma \vdash v_1 : A & \quad \Gamma \vdash v_2 : B \\
\Gamma \vdash \text{fst}((v_1, v_2)) \approx v_1 : A \quad \times_1 \beta \\
\Gamma \vdash \text{snd}((v_1, v_2)) \approx v_2 : B \quad \times_2 \beta \\
\Gamma \vdash v : A \times B \\
\Gamma \vdash (\text{fst}(v), \text{snd}(v)) \approx v : A \times B \quad \times \eta \\
\Gamma \vdash v : A & \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C \\
\Gamma \vdash \text{case}(\text{inl}(v), x. e_1, y. e_2) \approx [x/y]e_1 : C \quad +\text{inl} \beta \\
\Gamma \vdash v : B & \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, y : B \vdash e_2 : C \\
\Gamma \vdash \text{case}(\text{inr}(v), x. e_1, y. e_2) \approx [y/x]e_2 : C \quad +\text{inr} \beta \\
\Gamma \vdash e : A + B \\
\Gamma \vdash \text{case}(e, x. \text{inl}(x), y. \text{inr}(y)) \approx e : A + B \quad +\eta \\
\Gamma, x : A \vdash e : B & \quad \Gamma \vdash v : A \\
\Gamma \vdash (\lambda (x : A). e) \ v \approx [x/y]e : B \quad \Rightarrow \beta \\
\Gamma \vdash v : A \Rightarrow B & \quad \Gamma \vdash (\lambda (x : A). e) \ v \ x \approx v : A \Rightarrow B \quad \Rightarrow \eta \\
\Gamma, x : \tilde{A} \vdash e : B & \quad \Gamma \vdash v : \tilde{A} \\
\Gamma \vdash (\tilde{\lambda} (x : \tilde{A}). e) \ v \approx [x/y]e : B \quad \tilde{\lambda} \beta \\
\Gamma \vdash e : A + B & \quad \Gamma \vdash (\tilde{\lambda} (x : \tilde{A}). e) \ x \approx e : A + B \quad \tilde{\lambda} \eta
\end{align*}
\]

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

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

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 $\gamma$ 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 $\tilde{\lambda} x. \ x$ with case.
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:

\[
\begin{align*}
\text{C}_A(\text{C}(e)) &= e & \text{C-APP} \\
\text{callcc}_A(\lambda(k : \hat{A}). \text{C}(\text{inl}(\text{inl}(1) k))) &= e & \text{callcc-APP} \\
\text{call/cc}_{A,B}(\lambda(k : A \rightarrow B). \text{C}(\text{inr} e)) &= e & \text{call/cc-ABS} \\
\text{call/cc}_{A,B}(e) &= \text{call/cc}_{A,C}(\lambda(k : A \rightarrow C). e(\lambda(x : A). \text{C}(\text{inr} k x))) & \text{call/cc-APP} \\
\text{call/cc}_{A,B}(\text{inr} e) &= \text{call/cc}_{C,B}(\lambda(k : C \rightarrow B). \text{C}(\text{inr}(\text{inr}(1) k))) & \text{call/cc-NAT}
\end{align*}
\]

Not including the 0 type is a stylistic choice, not a semantic one, since it is interpreted by R, and \( \hat{A} \) becomes equivalent to \( A \Rightarrow 0 \) after adding equations for \( \hat{A} \). 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 \( \lambda \)-calculus (or higher-order functions) can be decomposed into first-order \( \kappa/\zeta \) calculi [Hasegawa 1995] with value/variable arrows. Building on the theme of duality – we show the decomposition of \( \hat{\lambda} \) (or higher-order cofunctions) into first-order \( \tilde{\kappa}/\tilde{\zeta} \) 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 \( C : \mathcal{C} \to \mathcal{C} \) and its Kleisli resolution \( F_G \dashv U_G : \mathcal{C} \to \mathcal{C}_G \),

the following are equivalent:

1. \( G \) has a right adjoint \( G + T : \mathcal{C} \to \mathcal{C} \).
2. \( U_G \) has a right adjoint \( U_G + R : \mathcal{C}_G \to \mathcal{C} \).

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

1. \( T \) has a left adjoint \( G + T : \mathcal{C} \to \mathcal{C} \).
2. \( U_T \) has a left adjoint \( L + U_T : \mathcal{C}_T \to \mathcal{C} \).

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 \( Ta \xrightarrow{Th_a} TGGa \xrightarrow{TGea} TGa \xrightarrow{Tf} Tb \). This makes \( T = R \circ U_G \) a monad. Hermida gives a direct calculation of the monad structure. \( \square \)

The informal idea is that in the \( \lambda \)-calculus, \( C \times (\_ ) \) is a reader/coreader/environment comonad, with a free value \( 1 \xrightarrow{\_} C \), and its right adjoint \( C \Rightarrow (\_ ) \) is a reader monad, with a free value \( 1 \xrightarrow{\_} C \) injected into its environment. Dually, \( C + (\_ ) \) is an exception monad, with a free covalue \( C \xrightarrow{\_} 0 \) in its environment, that is, an escape hatch to jump to \( C \). In \( \hat{\lambda} \) (with cocartesian coclosure), this has a left adjoint comonad \( C((\_ )) \), which merits the name: exception/coexception/handler comonad. It has a free covalue \( C \xrightarrow{\_} 0 \) injected into its environment, or, a handler for \( C \).

**Proposition 7.2.** In a cartesian closed category with \( c \times (\_ ) \) and \( (\_ )^c : \mathcal{C} \to \mathcal{C} \):

1. \( c \times (\_ ) : \mathcal{C} \to \mathcal{C} \) is a comonad.
2. \( (\_ )^c : \mathcal{C} \to \mathcal{C} \) 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 \( c((-) + c + (\_ )) : \mathcal{C} \to \mathcal{C} \):

1. \( c + (\_ ) : \mathcal{C} \to \mathcal{C} \) is a monad.
2. \( c((-)) : \mathcal{C} \to \mathcal{C} \) 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 $\mathcal{C}_{\text{cx}}(-)$, written $\mathcal{C}[c]$, has a generic element (value) $e_c : 1 \to c$, given by $c \times 1 \sim c$. This is Hasegawa’s “fullness condition”: $\mathcal{C}[c](1, -) \cong \mathcal{C}(c, -)$. Dually, the Kleisli category $\mathcal{C}_{\text{cxe}}(-)$, written $\mathcal{C}[\tilde{c}]$, has a generic element (covalue) $e_{\tilde{c}} : c \to 0$, given by $c \to c + 0$. 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 $F : \mathcal{C} \to \mathcal{D}$ a ccc functor. Let $c \in \mathcal{C}$, and $t : F(1) \cong 1 \to F(c)$ be an element in $\mathcal{D}$. There is a unique (upto isomorphism) extension of $F$ to a ccc functor $\tilde{F} : \mathcal{C}[c] \to \mathcal{D}$, such that $\tilde{F} \circ U_{\text{cx}(-)} \cong F$, and $F(e_c) = t$.

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

\[
\begin{array}{ccc}
\mathcal{C} & \xrightarrow{F} & \mathcal{C}[c] \\
\xrightarrow{\tilde{F}} & \downarrow & \\
\mathcal{D} & \xrightarrow{\tilde{F}} & \mathcal{D}
\end{array}
\]

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

\[
F(a) \Rightarrow 1 \times F(a) \Rightarrow F(1) \times F(a) \xrightarrow{\tau \times F(a)} F(c) \times F(a) \Rightarrow F(c \times a) \xrightarrow{F(f)} F(b)
\]

\[
F(a) \xrightarrow{F(f)} F(c + b) \Rightarrow F(c) + F(b) \xrightarrow{\tau + F(b)} F(0) + F(b) \Rightarrow 0 + F(b) \Rightarrow F(b)
\]

\[\square\]

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

\[
\begin{align*}
F_{\text{cx}(-)} \dashv U_{\text{cx}(-)} : \mathcal{C} & \Rightarrow \mathcal{C}[c] & F_{\text{cx}(-)} \dashv U_{\text{cx}(-)} : \mathcal{C}[c] & \Rightarrow \mathcal{C} \\
F_{\text{cxe}(-)} \dashv U_{\text{cxe}(-)} : \mathcal{C}[\tilde{c}] & \Rightarrow \mathcal{C} & F_{\text{cxe}(-)} \dashv U_{\text{cxe}(-)} : \mathcal{C}[\tilde{c}] & \Rightarrow \mathcal{C}[\tilde{c}]
\end{align*}
\]

### 7.2 $\kappa/\zeta$ and $\bar{\kappa}/\bar{\zeta}$ calculi

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

Just as $\kappa$ and $\zeta$ can be understood as first-order languages for understanding functions, $\bar{\kappa}$ and $\bar{\zeta}$ can be understood as first-order languages for understanding exceptions and handlers. By interpreting in our running ccc category $\mathcal{C}_{\bar{\kappa}}$, a covalue $c : C \leadsto 0$ is a $C \to R^{2 \to} \cong C \to R$ – a continuation, and these first-order operations bind and apply covalues on arrows, changing control flow. The type $A - C$ is interpreted as $A$ with a handler for $C$ attached, and $A + B$ 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):

\[
A \xrightarrow{f} B \xrightarrow{\tilde{c} \tilde{e}, g} Z + C \xrightarrow{h, Z} Z + D \xrightarrow{\text{pass}_{\text{p}}(Z)} D \xrightarrow{e} E
\]
\[
\begin{align*}
[x:1 \leadsto C] \\
\vdots \\
f : A \leadsto B & \quad \kappa^+ \\
kx^C.f : (C \times A) \leadsto B & \quad \times L \\
c : 1 \leadsto C & \quad \kappa^- \\
lift_A(c) : A \leadsto (C \times A) & \quad \times R
\end{align*}
\]

\( (kx^C.f) \circ \text{lift}_A(c) \equiv f[\xi/c] : A \leadsto B \)

\( h : (C \times A) \leadsto B \)

\( \kappa^+ \)

\( \kappa^- \)

\( \text{pass}_B(c) : (C \Rightarrow B) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \Rightarrow B) \)

\( \xi^- \)

\( \xi^+ \)

\begin{align*}
[x:1 \leadsto C] \\
\vdots \\
f : A \leadsto B & \quad \kappa^+ \\
kx^C.f : (C \times A) \leadsto B & \quad \times L \\
c : 1 \leadsto C & \quad \kappa^- \\
lift_A(c) : A \leadsto (C \times A) & \quad \times R
\end{align*}

\( h : A \leadsto (C \Rightarrow B) \)

\( \xi^- \)

\( \xi^+ \)

\( \text{pass}_B(c) : (C \Rightarrow B) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \Rightarrow B) \)

\( \xi^- \)

\( \xi^+ \)

\begin{align*}
[x:1 \leadsto C] \\
\vdots \\
f : A \leadsto B & \quad \kappa^+ \\
kx^C.f : (C \times A) \leadsto B & \quad \times L \\
c : 1 \leadsto C & \quad \kappa^- \\
lift_A(c) : A \leadsto (C \times A) & \quad \times R
\end{align*}

\( h : A \leadsto (C \times A) \leadsto B \)

\( \kappa^- \)

\( \text{pass}_B(c) : (C \times A) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \times A) \leadsto B \)

\( \kappa^- \)

\( \text{pass}_B(c) : (C \times A) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \times A) \leadsto B \)

\( \kappa^- \)

\( \text{pass}_B(c) : (C \times A) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \times A) \leadsto B \)

\( \kappa^- \)

\( \text{pass}_B(c) : (C \times A) \leadsto B \)

\( \Rightarrow_L \)

\( h : A \leadsto (C \times A) \leadsto B \)

\( \kappa^- \)
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 $\hat{\lambda}$

Conor McBride once said:


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 $\hat{\lambda}$ 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 $\rightarrow$ 'b) $\rightarrow$ ('a, 'b) either
  val coapp : ('a, 'b) either $\rightarrow$ 'a cont $\rightarrow$ 'b
end
structure Coexp: COEXP =

struct
  type 'a cont = 'a cont
  fun colam (f : 'a cont $\rightarrow$ 'b) : ('a, 'b) either =
    callcc (fn (k : ('a, 'b) either cont) $\Rightarrow$
      let val a = callcc (fn (ka : 'a cont) $\Rightarrow$ 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 $\Rightarrow$ throw k a
    | INR b $\Rightarrow$ b
end

We then recover callcc/throw from colam/coapp.

fun codiag (e : ('a, 'a) either) : 'a =
  case e of
  INL a $\Rightarrow$ a
  | INR a $\Rightarrow$ a

fun callcc (f : 'a cont $\rightarrow$ '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.
\textbf{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.

\begin{verbatim}
swap :: a + b \to b + a
swap = either Right Left

assumeRight :: ((a \to r) \to Cont r b) \to Cont r (a + b)
assumeRight = colam

resolveRight :: Cont r (a + b) \to (a \to r) \to Cont r b
resolveRight = coapp

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

resolveLeft :: Cont r (a + b) \to (b \to r) \to Cont r a
resolveLeft = coapp \cdot fmap swap
\end{verbatim}

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

\textbf{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.

\textbf{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 $\mu$ 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 $\bot$) 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 $\rightarrow$ and $\leftarrow$. 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: $C \times S^A \cong C \rightarrow R^{S^A}$.

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


A SUPPLEMENTARY MATERIAL FOR SECTION 1 (INTRODUCTION)

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

\[
\begin{align*}
\Gamma, A &\vdash B & \Gamma, A &\vdash A \\
\Gamma &\vdash A \rightarrow B & \Gamma &\vdash A \\
\Gamma &\vdash B & \Gamma &\vdash \neg A \\
\Gamma &\vdash \neg \neg B & \Gamma &\vdash \neg (A \lor B) \\
\end{align*}
\]

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

\[
\begin{align*}
\Gamma, A &\vdash A \lor B & \Gamma, B &\vdash A \lor B \\
\Gamma, \neg (A \lor B), A &\vdash \bot & \Gamma, \neg (A \lor B), B &\vdash \bot \\
\Gamma, \neg (A \lor B) &\vdash \neg A & \Gamma, \neg (A \lor B) &\vdash \neg B \\
\end{align*}
\]

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

\[
\begin{align*}
C \times A &\rightarrow B \equiv C \rightarrow B^A & C \times R^A &\rightarrow R^{B^A} \equiv C \rightarrow R^{A \lor B} \\
\end{align*}
\]

or in terms of adjunctions:

\[
\begin{align*}
(-) \times A &\dashv (-)^A & (-) \times R^A &\dashv A + (-) \\
\end{align*}
\]

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:

```ocaml
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:

```ocaml
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 \( \text{tnd} \) and \( \text{case} \), we can perform double negation introduction:

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

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

```plaintext
fun \text{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:

```plaintext
fun \text{dne} (kka : co (co a)) : 'a =
    \text{adj} (fn ka ⇒ ka) kka
fun \text{contramap} (f : a → b) : co b → co a =
    \text{adj} (fn kka ⇒ f (dne kka))
```

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

```plaintext
fun \text{lam} (f : a → b) : co a + b =
    cofn kka ⇒ f (dne kka)
fun \text{app} (e : co a + b) : a → b =
    fn a ⇒ e @ dni a
```

de Morgan’s laws are obtained as follows:

```plaintext
fun \text{deMorgan1} (k : co (a + b)) : (co a * co b) =
    (contramap INL k, contramap INR k)
fun \text{deMorgan2} ((ka , kb) : co a * co b) : co (a + b) =
    contramap (fn e ⇒ e @ ka) kb
fun \text{deMorgan3} (kp : co (a * b)) : co a + co b =
    adj (fn ks ⇒ let val (kka, kkb) = \text{deMorgan1} ks in (dne kka, dne kkb) end) kp
fun \text{deMorgan4} (ks : co a + co b) : co (a * b) =
    case \text{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 \texttt{ftoc} and \texttt{ctof} (named after the operators in [Führmann and Thielecke 2004, § 3]).

\begin{verbatim}
fun ftoc (f : a \rightarrow b) : co (a - b) =
    case lam f of
    | INL ka \Rightarrow deMorgan4 (INL ka)
    | INR b \Rightarrow deMorgan4 (INR (dni b))

fun ctof (k : co (a - b)) : 'a \rightarrow 'b =
    fn a \Rightarrow case deMorgan3 k of
    | INL ka \Rightarrow throw a ka
    | INR kkb \Rightarrow dne kkb
\end{verbatim}

Finally, Peirce’s law, which is the type of the general \texttt{callCC} can be derived as follows:

\begin{verbatim}
fun peirce (f : ('a \rightarrow 'b) \rightarrow 'a) : 'a =
    case lam f of
    | INL kg \Rightarrow #1 (adj ctof kg)
    | INR a \Rightarrow a
\end{verbatim}

This highlights how the covalue for \texttt{b} is dropped.

\section*{C Supplementary Material for Section 4 (Semantics)}

\subsection*{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, \(c \land a \leq b\) iff \(c \leq a \rightarrow b\), 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 distributive 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 \(a \leq b \lor c\) iff \(a \backslash b \leq c\). 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 is a bi-Heyting algebra.

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

\begin{enumerate}
    \item \(m \land n \triangleq \min(m, n)\), bounded by \(\omega\).
    \item \(m \lor n \triangleq \max(m, n)\), bounded by \(0\).
    \item \(m \rightarrow n \triangleq \max \{ p \mid \min(p, m) \leq n \}\).
    \item \(m \backslash n \triangleq \min \{ p \mid m \leq \max(n, p) \}\).
\end{enumerate}

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

\begin{enumerate}
    \item \(X \land Y \triangleq X \cap Y\), bounded by \(C\).
    \item \(X \lor Y \triangleq X \cup Y\), bounded by \(\emptyset\).
    \item \(X \rightarrow Y \triangleq X^c \cup Y\).
    \item \(X \backslash Y \triangleq X \cap Y^c\).
\end{enumerate}

More generally, the subobject classifier of any presheaf category \(\text{PSh}(C)\) 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).
The Duality of Abstraction

From the computational content, as we will see in later sections, by giving an equational theory. On the same category, we do not encounter the problem of degeneracy. This means we still retain dualization monad (Kock). Since we do not have the cartesian closure and co-cartesian co-closure.

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 \(\lambda \eta\) rule, we can derive the two backtracking rules in figure 12.