Notions of Stack-manipulating Computation and Relative Monads

YUCHEN JIANG, University of Michigan, USA
RUNZE XUE, University of Michigan, USA
MAX S. NEW, University of Michigan, USA

We present *relative monads* in a polymorphic call-by-push-value (CBPV) calculus as a common abstraction for stack-based implementations of effects. This brings the powerful abstraction of monadic programming to lower-level stack-based languages. We demonstrate the applicability of the relative monad abstraction by modeling stack-based implementations of common monads used in functional programming as relative monads such as exceptions, state, continuations and free monads.

Next, we demonstrate that relative monads in CBPV are more compositional than monads in pure languages. First, while the use of monads allows for the use of embedded call-by-value programming in a pure language using do-notation, in contrast a relative monad allows for embedded CBPV programming within a CBPV programming language. This means that all code in CBPV can be reinterpreted to work with an arbitrary relative monad. Inspired by this, we extend our CBPV calculus with a new feature called "monadic blocks" that allow for reinterpretation of code with respect to any user-defined monad. We then show that monadic blocks allow us to take any monad implementation in CBPV and automatically derive the corresponding monad transformer, a process which is not automatic for monads in pure languages.

All examples are executable in Zydeco, a stack-based functional language we have designed and implemented based on polymorphic CBPV and extended with monadic blocks.

1 INTRODUCTION

Since Moggi's seminal work [Moggi 1991], monads have become a wildly successful tool in two main areas of programming languages. First, as Moggi originally showed, denotational models of functional programming languages with effects are naturally modeled using monads. Secondly, monads have become an indispensable programming abstraction for effects in functional languages, most notably in Haskell, where core I/O primitives are made available through a monadic interface [Peyton Jones and Wadler 1993; Wadler 1990].

The power of monads comes from their ability to "overload the semicolon" by instantiating a quite simple structure: a type constructor, return and bind, satisfying some natural equations. This gives programmers access to do notation, which allows for embedded call-by-value programming within the context of a pure host language. Additionally, polymorphic languages allow for the implementation of combinators polymorphic in the underlying monad, providing powerful macros for embedded effectful programming. For semanticists, monads provide a simple recipe to construct mathematical models of effectful languages. These are really two views on the same idea: the monadic programmer is essentially using the model construction within their host language to embed their effectful call-by-value programming language. Programmers and semanticists have developed further refinements of monads such as monad transformers and algebraic effects to work with programs or semantics that mix multiple effects in a compositional way while still providing sensible equational reasoning principles that programmers intuitively understand [Liang et al. 1995; Plotkin and Power 2001].

Authors' addresses: Yuchen Jiang, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, lighght@umich.edu; Runze Xue, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, cactus@umich.edu; Max S. New, Computer Science and Engineering, University of Michigan, Street1 Address1, Ann Arbor, Michigan, Post-Code1, USA, maxsnew@umich.edu.

2018. ACM 2475-1421/2018/1-ART1
https://doi.org/

But the purview of effectful programming is hardly limited to abstract mathematical semantics or embedded effectful programming. Most popular programming languages allow for relatively free use of several effects: local and global mutable state, exceptions, threads, and operating system services to name a few. The implementation of these effects is handled not by a source-language programmer implementing a high-level monad interface but instead by the language implementor. While some effects may be directly implemented through dedicated hardware mechanisms, most control effects are essentially virtualized, not unlike those of the embedded monadic programmer. A major difference is that the language implementor typically has a much lower-level view of this virtualization. Effects are implemented using various techniques for *stack manipulation*: stack walking, stack swapping, or passing of multiple continuations. Designing these effect implementations thus necessitates thinking at a lower level of abstraction than allowed in high-level languages, which by design limit the control that programmers have over the stack structure. However, some connections between monads and stacks are understood: correspondences between monadic evaluators and stack-based abstract machines [Ager et al. 2005], as well as efficient monad implementations in Haskell [Kiselyov and Ishii 2015].

In this paper, we show that it is possible to adapt the monad abstraction to model stack-based implementations of effects. We approach the problem by working in a calculus for stack-manipulation: a polymorphic variant of Levy's call-by-push-value (CBPV). Just as λ calculus is the "standard model" of pure functional programming, we argue that CBPV should be seen as a standard model for stack-manipulating computations. The key is that CBPV has two kinds of types: the value types, which classify first class data that can be passed and returned in functions, and the computation types, which classify programs that manipulate machine states. In this work we take a *dual* view of computation types and view them as *stack types* that classify stack structures with which a computation interacts[1]. This viewpoint is directly supported by the stack-based abstract machine semantics.

After demonstrating some examples of stack-based programming, we show how to adapt the monadic interface to the CBPV setting. We observe that the natural design for stack-based effects is not the ordinary kind of monad, because the kind of our "monads" is kind VTy $\longrightarrow$ CTy, as it takes a value type A to a computation type that describes how the stack must be structured for an A-returning computation to run. That is, our "monads" are not even endofunctors. Fortunately, Altenkirch, Chapman and Uustalu showed that "monads need not be endofunctors", in that monads can be generalized to *relative* monads, and these relative monads capture a natural form of effectful programming with stacks [Altenkirch et al. 2010]. We give a definition of a relative monad in CBPV and show that standard monadic programming patterns (many common monads, algebras of a monad and monad transformers) can all be adapted to this relative setting.

We then demonstrate that the shift from monads to relative monads results in a more compositional notion of effect. That is, while monads allow for embedding call-by-value programming in a pure host language using do-notation, relative monads allow for embedding *call-by-push-value* programming within a call-by-push-value language. That is, the embedded effectful language supports *all* features of the host language itself. Inspired by this, we extend our CBPV calculus with a new feature we call a *monadic block* which allows for code written in ordinary CBPV to be reinterpreted with respect to any user-specified monad. We show that monadic blocks can be implemented by a source-to-source transformation, similar to do-notation.

All examples in the paper have been implemented in a proof of concept implementation of our calculus we call Zydeco which supports polymorphic CBPV with monadic blocks. This language is a bit more ergonomic to use, featuring a bidirectional type system as well as unifying most of

[1]this duality can be viewed as a variant of linear logic duality, see Egger et al. [2014]

the core primitive type constructors under generalized `data` and `codata` forms. Additionally it includes primitive types and built-in functions to allow for simple shell scripting examples.

One application of monadic blocks is that contrary to the situation for pure languages, *any* relative monad definable in CBPV can be extended to a relative monad transformer. Intuitively this is because CBPV comes with an ambient notion of effect, and so all relative monads must be relative to this "ambient" effect, and the monadic blocks allow us to reinterpret the ambient effect as any effect of our choosing. This also provides a new method for constructing monad transformers in pure languages, as any relative monad/monad transformer transformer in CBPV gives a corresponding one in a pure language by trivializing the CBPV structure. This gives a new perspective for how standard monad transformers arise.

The remainder of the paper is structured as follows:

(1) In Section 2 we present the syntax, typing and stack machine semantics of our polymorphic CBPV calculus along with some examples of how CBPV allows for stack-oriented functional programming.
(2) In Section 3 we introduce relative monads as a programming abstraction in CBPV and demonstrate several examples how we can abstract over uses of the stack to implement state and control effects.
(3) In Section 4 we extend CBPV with *monadic blocks* and show how these can be implemented using *algebras* of a relative monad
(4) In Section 5 we apply our monadic blocks to derive relative monad transformers automatically from relative monads.
(5) In Section 6 we connect our CBPV programming constructs with their corresponding category-theoretic notions. We show that the implementation of monadic blocks corresponds to a theorem we call the *fundamental theorem of CBPV relative monads.*
(6) In Section 7 we discuss related and future work.

2 A POLYMORPHIC CBPV LANGUAGE FOR STACK MANIPULATION

In this section, we introduce our calculus $\mathrm{CBPV}_{\omega}^{\forall,\exists,\nu}$ an extension of CBPV calculus with F_{ω}-style impredicative polymorphism and recursive computation types.

2.1 Call-by-push-value Syntax

A full overview of syntactic forms of $\mathrm{CBPV}_{\omega}^{\forall,\exists,\nu}$ is given in Figure 1. There are six basic syntactic forms. Since we have F_{ω}-style of polymorphism [Girard 1972], we have *kinds.* In addition to function kinds, we have two different kinds of types in CBPV. *Value types* (kind VTy) classify inert data that can be passed around as first-class values. *Computation types* (kind CTy) classify imperative computations that interact with the stack, or dually, can be thought of as classifying the structure of the current state of the stack. Type environments Δ contain type variables of various kinds. Value environments Γ contain value variables with their associated value types. Then the two types of terms are inert first-class *values* and imperative *computations.*

We present the kinding judgment in Figure 2. We have standard rules for type variables and function kinds. Next we have the value type constructors. These include nullary and binary sums and products, which act as in a call-by-value language, as well as existentially quantified packages $\exists\, X : K.\, A$ which can quantify over types of any kind. Lastly, we have the type constructor Thk of *thunks* that takes a computation type to the type of first class suspended computations of that type, i.e., *closures.* Next we have the computation type constructors, which as we recall classify how imperative computations can interact with the runtime stack, and therefore, what the state of the stack may be. The type $A \to B$ for A a value type and B a computation type is the type

$$
\begin{array}{rrcl}
\text{Kind} & K & ::= & \mathsf{VTy} \mid \mathsf{CTy} \mid K_1 \to K_2 \\
\\
\text{Type Env} & \Delta & ::= & \cdot \mid \Delta,\ X : K \\
\text{Type} & S, A, B & ::= & X \mid \lambda\, X : K.\, S \mid S\, S_0 \\
 & & \mid & \mathtt{Thk}\, B \mid \mathtt{Unit} \mid A_1 \times A_2 \mid A_1 + A_2 \mid \exists\, X : K.\, A \\
 & & \mid & \mathtt{Ret}\, A \mid A \to B \mid \mathtt{Top} \mid B_1 \,\&\, B_2 \mid \forall\, X : K.\, B \mid \nu\, Y : K.\, B \mid \mathtt{OS} \\
\\
\text{Value Env} & \Gamma & ::= & \cdot \mid \Gamma,\ x : A \\
\text{Value} & V & ::= & x \mid \{M\} \mid () \mid (V_1, V_2) \mid \mathsf{inj}_i(V) \mid (S, V) \mid \mathtt{halt} \\
\text{Computation} & M & ::= & !\, V \mid \mathsf{let}\ (x_1, x_2) = V\ \mathsf{in}\ M \\
 & & \mid & (\mathsf{match}\ V \mid \mathsf{inj}_0(x_0) \Rightarrow M_0 \mid \mathsf{inj}_1(x_1) \Rightarrow M_1) \\
 & & \mid & \mathsf{let}\ (X, x) = V\ \mathsf{in}\ M \\
 & & \mid & \mathsf{ret}\ V \mid \mathsf{do}\ x \leftarrow M_0\, ;\, M \mid \lambda\, x : A.\, M \mid M\, V \\
 & & \mid & (\mathsf{comatch}\ |) \mid (\mathsf{comatch} \mid .0 \Rightarrow M_0 \mid .1 \Rightarrow M_1) \mid M\, .i \\
 & & \mid & \Lambda\, X : K.\, M \mid M\, S \mid \mathsf{roll}(M) \mid \mathsf{unroll}(M) \mid \mathsf{fix}\, x.\, M
\end{array}
$$

Fig. 1. Syntax of $\text{CBPV}_{\omega}^{\forall,\exists,\nu}$

of computations that are *functions* that pop an A value off of the stack and then proceed as a B computation. Therefore the stack must consist of an A value pushed onto a B stack. Similarly, polymorphic functions $\forall X.B$ are functions that take a type as input. The type $B_0 \& B_1$ is a "lazy" binary product type, it says the computation must pop a tag off the stack, if the stack is 0 proceed as a B_0 computation and if it's 1 proceed as a B_1 computation. Therefore the stack must be a pair of a tag with a matching tail of the stack. The type `Top` is a nullary product, so it must pop off a tag drawn from the empty set. In other words, there are no possible stacks, and so this type represents computations that are dead code. The recursive computation types $\nu Y.B$ allow for recursive specifications of computations, or dually, of recursive types of stacks, which we will see allows us to define stacks that can be arbitrarily large. Finally we have the type constructor `Ret` which takes a type of values A to the type of computations that can return values of type A. Dually, the type of stacks is that of an opaque continuation for values of type A.

We have departed slightly from Levy's original CBPV syntax: inspired by denotational models, he uses U for `Thk` and F for `Ret`, but we have chosen more operationally-motivated name. Additionally we use the standard linear logic syntax for computation products.

Lastly we give the typing rules for values and computations in Figures3,4. First, we have the rules for values, which include variables, thunks, which we denote with "suspenders" following the Frank language [Lindley et al. 2017], and constructors for unit, product and sum types. The elimination forms for these values, which are given by pattern matching, are constructors on computations.

The `Ret` rules are similar to a monad, with `ret` as the introduction rule and a bind for the elimination, but with an arbitrary computation type allowed for the continuation. We can force a `Thunk B` to get a `B` computation. We add standard let and recursion rules, though note that the recursive rule must put a `Thunk B` into the context, as all variables are of value type. Ordinary and polymorphic functions are given by λ/Λ and application rules. Computation products have projections as their destructors and their introduction rule is given by "copattern match" on which destructor is on the stack.

$\boxed{\Delta \vdash S : K}$ type (constructor) S has kind K under type environment Δ

$$\frac{X : K \in \Delta}{\Delta \vdash X : K}\ [\textsc{TyTVar}] \qquad \frac{\Delta, X : K_0 \vdash S : K}{\Delta \vdash \lambda\, X : K_0.\, S : K_0 \to K}\ [\textsc{TyTLam}] \qquad \frac{\Delta \vdash S : K_0 \to K \qquad \Delta \vdash S_0 : K_0}{\Delta \vdash S\, S_0 : K}\ [\textsc{TyTApp}]$$

$$\mathtt{Thk} : \mathsf{CTy} \to \mathsf{VTy} \qquad \mathtt{Unit}, \mathtt{Empty} : \mathsf{VTy} \qquad (\times), (+) : \mathsf{VTy} \to \mathsf{VTy} \to \mathsf{VTy}$$

$$\frac{\Delta, X : K_0 \vdash A : \mathsf{VTy}}{\Delta \vdash \exists\, X : K_0.\, A : \mathsf{VTy}}\ [\textsc{TyExists}]$$

$$\mathtt{Ret} : \mathsf{VTy} \to \mathsf{CTy} \qquad (\to) : \mathsf{VTy} \to \mathsf{CTy} \to \mathsf{CTy} \qquad \mathtt{Top} : \mathsf{CTy} \qquad (\&) : \mathsf{CTy} \to \mathsf{CTy} \to \mathsf{CTy}$$

$$\frac{\Delta, X : K_0 \vdash B : \mathsf{CTy}}{\Delta \vdash \forall\, X : K_0.\, B : \mathsf{CTy}}\ [\textsc{TyForall}] \qquad \frac{\Delta, Y : \mathsf{CTy} \vdash B : \mathsf{CTy}}{\Delta \vdash \nu\, Y : \mathsf{CTy}.\, B : \mathsf{CTy}}\ [\textsc{TyNu}]$$

Fig. 2. Kinding Rules

$\boxed{\Delta; \Gamma \vdash V : A}$ V has type A under type environment Δ and value environment Γ

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ [\textsc{ValVar}] \qquad \frac{\Delta; \Gamma \vdash M : B}{\Delta; \Gamma \vdash \{M\} : \mathtt{Thk}\ B}\ [\textsc{ValThunk}] \qquad () : \mathtt{Unit}$$

$$\frac{\Delta; \Gamma \vdash V_1 : A_1 \qquad \Delta; \Gamma \vdash V_2 : A_2}{\Delta; \Gamma \vdash (V_1, V_2) : A_1 \times A_2}\ [\textsc{ValPair}] \qquad \frac{\Delta; \Gamma \vdash V : A_i}{\Delta; \Gamma \vdash \mathsf{inj}_i(V) : A_0 + A_1}\ [\textsc{ValInj}]$$

$$\frac{\Delta; \Gamma \vdash V : A[S/X]}{\Delta; \Gamma \vdash (S, V) : \exists\, (X : K).\, A}\ [\textsc{ValPack}]$$

Fig. 3. Value and Stack Typing for Zydeco

2.2 Operational Semantics with a Stack Machine

We present the operational semantics of Zydeco in Figure 5, using an abstract stack machine. First we inductively define stacks to be either an empty stack $\bullet$ or a stack "frame" for one of the computation types. The `Ret` frame is a continuation, application and type application frames have an argument pushed on the stack. The computation product has a destructor tag pushed onto the stack and the recursive computation has an unroll on the stack. We note that the type application and unroll are not strictly necessary at runtime but make it easier to keep track of the typing of the stack. An initial state is a returning computation running against the empty stack $\langle M \mid \bullet \rangle$ and the computation terminates if it reaches a state $\langle \text{ret } V \mid \bullet \rangle$.

The rules that appear in Figure 5 are standard for a CBPV calculus with a stack machine, similar to Levy's original stack machine semantics. Each of the computation elimination rules operates by pushing a frame onto the stack, and each computation introduction rule operates by popping the stack frame off and proceeding in a manner dependent on the contents of the frame. Forcing a thunk executes the suspended computation within. Let, fixed points and pattern matching are straightforward.

$\boxed{\Delta; \Gamma \vdash M : B}$ M has type B under type environment Δ and value environment Γ

$$\frac{\Delta; \Gamma \vdash V : \mathsf{Thk}\ B}{\Delta; \Gamma \vdash\ !\ V : B}\ [\textsc{ComForce}] \qquad \frac{\Delta; \Gamma \vdash V : A \qquad \Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \mathsf{let}\ x = V\ \mathsf{in}\ M : B}\ [\textsc{ComLet}]$$

$$\frac{\Delta; \Gamma \vdash V : A_0 \times A_1 \qquad \Delta; \Gamma, x_0 : A_0, x_1 : A_1 \vdash M : B}{\Delta; \Gamma \vdash \mathsf{let}\ (x_0, x_1) = V\ \mathsf{in}\ M : B}\ [\textsc{ComPrj}]$$

$$\frac{\Delta; \Gamma \vdash V : A_0 + A_1 \qquad \forall i \in \{0,1\},\ \ \Delta; \Gamma, x_i : A_i \vdash M_i : B}{\Delta; \Gamma \vdash (\mathsf{match}\ V \mid \mathsf{inj}_0(x_0) \Rightarrow M_0 \mid \mathsf{inj}_1(x_1) \Rightarrow M_1) : B}\ [\textsc{ComMatch}]$$

$$\frac{\Delta; \Gamma \vdash V : \exists\ (Y : K).\ A \qquad \Delta, X : K; \Gamma, x : A[Y/X] \vdash M : B}{\Delta; \Gamma \vdash \mathsf{let}\ (X, x) = V\ \mathsf{in}\ M : B}\ [\textsc{ComUnpack}]$$

$$\frac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathsf{ret}\ V : \mathsf{Ret}\ A}\ [\textsc{ComReturn}] \qquad \frac{\Delta; \Gamma \vdash M_0 : \mathsf{Ret}\ A \qquad \Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \mathsf{do}\ x \leftarrow M_0\ ; M : B}\ [\textsc{ComBind}]$$

$$\frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda\ (x : A).\ M : A \to B}\ [\textsc{ComLam}] \qquad \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash M\ V : B}\ [\textsc{ComApp}]$$

$$\frac{}{\Delta; \Gamma \vdash (\mathsf{comatch}\ |) : \mathsf{Top}}\ [\textsc{ComTop}] \qquad \frac{\forall i \in \{0,1\},\ \ \Delta; \Gamma \vdash M_i : B_i}{\Delta; \Gamma \vdash (\mathsf{comatch} \mid .0 \Rightarrow M_0 \mid .1 \Rightarrow M_1) : B_0\ \&\ B_1}\ [\textsc{ComCoMatch}]$$

$$\frac{\Delta; \Gamma \vdash M : B_0\ \&\ B_1}{\Delta; \Gamma \vdash M\ .i : B_i}\ [\textsc{ComDtor}] \qquad \frac{\Delta, Y : K; \Gamma \vdash M : B[Y/X]}{\Delta; \Gamma \vdash \Lambda\ (Y : K).\ M : \forall\ (X : K).\ B}\ [\textsc{ComTyLam}]$$

$$\frac{\Delta; \Gamma \vdash M : \forall\ (X : K).\ B}{\Delta; \Gamma \vdash M\ S : B[S/X]}\ [\textsc{ComTyApp}] \qquad \frac{\Delta; \Gamma \vdash M : B[\nu\ (Y : K).\ B/Y]}{\Delta; \Gamma \vdash \mathsf{roll}(M) : \nu\ (Y : K).\ B}\ [\textsc{ComRoll}]$$

$$\frac{\Delta; \Gamma \vdash M : \nu\ (Y : K).\ B}{\Delta; \Gamma \vdash \mathsf{unroll}(M) : B[\nu\ (Y : K).\ B/Y]}\ [\textsc{ComUnroll}] \qquad \frac{\Delta; \Gamma, x : \mathsf{Thk}\ B \vdash M : B}{\Delta; \Gamma \vdash \mathsf{fix}\ x.\ M : B}\ [\textsc{ComFix}]$$

Fig. 4. Computation Typing for Zydeco

2.3 Call-by-push-value by Example

Now that we have seen the basic constructs of CBPV$_\omega^{\forall,\exists,\nu}$, we present a few example to get a feel for how this allows for a kind of stack-oriented functional programming language.

First, consider the at first unintuitive fact that the connective is a connective on computation types. The benefit of this choice is that the function type in CBPV is more compositional than in call-by-value. In call-by-value, the currying isomorphism $A \to A' \to A'' \cong A \times A' \to A''$ is not valid because a curried function has two chances to perform effects, while the uncurried function has only one. However in CBPV, the two types $A \to A' \to B$ and $A \times A' \to B$ are isomorphic, which makes intuitive sense in terms of their stacks. The former has a stack with an A pushed onto

$$\text{Stack} \quad \mathcal{K} \quad ::= \quad \bullet \mid \mathsf{Kont}(x\;.\;M) :: \mathcal{K} \mid \mathsf{App}(V) :: \mathcal{K} \mid \mathsf{App}(V) :: \mathcal{K} \mid \mathsf{Dtor}(i) :: \mathcal{K} \mid \mathsf{Unroll} :: \mathcal{K}$$

$\boxed{\langle M \mid \mathcal{K}\rangle \longrightarrow \langle M' \mid \mathcal{K}'\rangle}$ M with stack $\mathcal{K}$ steps to M' with stack $\mathcal{K}'$

$$\frac{}{\langle \mathsf{ret}\; V \mid \mathsf{Kont}(x\;.\;M) :: \mathcal{K}\rangle \longrightarrow \langle M[V/x] \mid \mathcal{K}\rangle}\;[\textsc{OpRet}]$$

$$\frac{}{\langle \mathsf{do}\; x \leftarrow M_1\;;M_2 \mid \mathcal{K}\rangle \longrightarrow \langle M_1 \mid \mathsf{Kont}(x\;.\;M_2) :: \mathcal{K}\rangle}\;[\textsc{OpBind}]$$

$$\frac{}{\langle\, !\, \{M\} \mid \mathcal{K}\rangle \longrightarrow \langle M \mid \mathcal{K}\rangle}\;[\textsc{OpForceThunk}] \qquad \frac{}{\langle \mathsf{let}\; x = V \;\mathsf{in}\; M \mid \mathcal{K}\rangle \longrightarrow \langle M[V/x] \mid \mathcal{K}\rangle}\;[\textsc{OpLet}]$$

$$\frac{}{\langle \mathsf{let}\; (x_0, x_1) = (V_0, V_1)\;\mathsf{in}\; M \mid \mathcal{K}\rangle \longrightarrow \langle M[V_0/x_0][V_1/x_1] \mid \mathcal{K}\rangle}\;[\textsc{OpPrj}]$$

$$\frac{}{\langle (\mathsf{match}\; \mathsf{inj}_i(V) \mid \mathsf{inj}_0(x_0) \Rightarrow M_0 \mid \mathsf{inj}_1(x_1) \Rightarrow M_1) \mid \mathcal{K}\rangle \longrightarrow \langle M_i[V/x_i] \mid \mathcal{K}\rangle}\;[\textsc{OpMatch}]$$

$$\frac{}{\langle \mathsf{let}\; (X, x) = (S, V)\;\mathsf{in}\; M \mid \mathcal{K}\rangle \longrightarrow \langle M[V/x] \mid \mathcal{K}\rangle}\;[\textsc{OpUnpack}]$$

$$\frac{}{\langle M\; V \mid \mathcal{K}\rangle \longrightarrow \langle M \mid \mathsf{App}(V) :: \mathcal{K}\rangle}\;[\textsc{OpApp}] \qquad \frac{}{\langle \lambda\, x.\, M \mid \mathsf{App}(V) :: \mathcal{K}\rangle \longrightarrow \langle M[V/x] \mid \mathcal{K}\rangle}\;[\textsc{OpLam}]$$

$$\frac{}{\langle M\, .i \mid \mathcal{K}\rangle \longrightarrow \langle M \mid \mathsf{Dtor}(i) :: \mathcal{K}\rangle}\;[\textsc{OpDtor}] \qquad \frac{}{\langle (\mathsf{comatch} \mid .0 \Rightarrow M_0 \mid .1 \Rightarrow M_1) \mid \mathsf{Dtor}(i) :: \mathcal{K}\rangle \longrightarrow \langle M_i \mid \mathcal{K}\rangle}\;[\textsc{OpComat}$$

$$\frac{}{\langle M\; S \mid \mathcal{K}\rangle \longrightarrow \langle M \mid \mathcal{K}\rangle}\;[\textsc{OpTyApp}] \qquad \frac{}{\langle \Lambda\, X.\, M \mid \mathsf{TyApp}(S) :: \mathcal{K}\rangle \longrightarrow \langle M[S/X] \mid \mathcal{K}\rangle}\;[\textsc{OpTyLam}]$$

$$\frac{}{\langle \mathsf{roll}(M) \mid \mathsf{Unroll} :: \mathcal{K}\rangle \longrightarrow \langle M \mid \mathcal{K}\rangle}\;[\textsc{OpRoll}] \qquad \frac{}{\langle \mathsf{unroll}(M) \mid \mathcal{K}\rangle \longrightarrow \langle M \mid \mathsf{Unroll} :: \mathcal{K}\rangle}\;[\textsc{OpUnroll}]$$

$$\frac{}{\langle \mathsf{fix}\; x.\, M \mid \mathcal{K}\rangle \longrightarrow \langle M[\{\mathsf{fix}\; x.\, M\}/x] \mid \mathcal{K}\rangle}\;[\textsc{OpFix}]$$

Fig. 5. Operational Semantics of Zydeco via a Stack Machine

a stack with an A' pushed onto a B stack, while the latter has a stack with an A and an A' pushed onto a B stack which is obviously equivalent.

Additionally, we can use more complex computation types to define more complex stack-based "calling conventions". For example, we can define functions $A \rightarrow^{?} B$ that take 0 or one arguments, and functions $A \rightarrow^{*} B$ that take 0, 1 or many arguments:

$$\begin{aligned} \mathsf{def}\ (\rightarrow^{?}) : \mathsf{VTy} \rightarrow \mathsf{CTy} \rightarrow \mathsf{CTy} \;&\triangleq\; \lambda\, A.\, \lambda\, B.\, (A \rightarrow B) \& B \\ \mathsf{def}\ (\rightarrow^{*}) : \mathsf{VTy} \rightarrow \mathsf{CTy} \rightarrow \mathsf{CTy} \;&\triangleq\; \nu(\rightarrow^{*}) \lambda\, A.\, \lambda\, B.\, (A \rightarrow A \rightarrow^{*} B) \& B \end{aligned}$$

Because these are computation type constructors, they can be combined together, providing a compositional type language for defining stack structure. For instance we can define a function that takes at least one argument as $A \rightarrow^{+} B := A \rightarrow A \rightarrow^{*} B$.

As program examples, in Figure 6, we define two functions. First, `abort` is a polymorphic function of type $\forall\, A.\, A \rightarrow^{+}$ `Ret` A which takes a variable number of arguments and returns the first one. Here to make the code more readable, we use the suggestive names `.more` and `.done` for the two projections in the definition of $\rightarrow^{*}$. It works by popping the type A and argument a off of the stack, and then defining a loop called `unwind` that copattern matches to see if there are any remaining

```
def abort : Thk (∀ A : VTy. A → A →* Ret A)
    ≜ { Λ A. λ a.
        fix unwind. comatch
          | .more _ → ! unwind
          | .done → ret a }
```

```
def sum_and_print : Thk (Int → Int →* OS)
    ≜ { fix loop. λ sum. comatch
        | .more i  ⇒  do j ← ! add sum i ; ! loop j
        | .done  ⇒ ! write_int sum { ! halt }}
```

Fig. 6. Two Code Examples

arguments. If there are more arguments, it continues, and otherwise it returns the original argument x. Here unwind has type Thk A $\to^*$ Ret A, i.e., it is a closure and we explicitly execute run it by ! unwind.

The second example is another demonstration of consuming a stack of type $\to^*$ that sums up integers on the stack, prints the result, and halt. Some builtin types supported by the implementation of Zydeco are used in this function. To make them well-typed in $\mathrm{CBPV}_{\omega}^{\forall,\exists,\nu}$, we should consider these to have a free computation type variable OS, which we think of as the type of computations that have access to the operating system, analogous in function to Haskell's IO (). Then the primitives for working with IO are given in continuation passing style[2]. For example, write_int has type Thk (Int → Thk OS → OS), taking an integer, printing it to the console, and running the passed-in continuation Thk OS. Finally, we use a primitive function add of type Thk (Int → Int → Ret Int) for addition. Altogether the code follows a similar pattern to abort: we define a tail recursive loop, using the Int parameter as the state, popping integers from the stack one by one to sum them up, exiting if the stack is empty. Check out Appendix A.1 to see how sum_and_print can be combined with read_and_stack to build a simple command-line program. This example demonstrates how the primitive CBPV do binding is *not* simply a monadic do. The continuation in the do binding for add is not of type Ret A but instead is the complex computation type Int $\to^*$ OS. This is a distinctive feature of CBPV: continuations of a do binding can be *any* computation type.

3 RELATIVE MONADS IN CBPV

Now that we have established a calculus for stack-manipulating computations, we consider how to embed effectful programming using *relative* monads.

3.1 Relative Monads and Their Laws

Similar to standard monads, we define a relative monad to consist of a type constructor T with return and bind operations:

$$\begin{aligned}\text{def Monad} : (\mathsf{VTy} \to \mathsf{CTy}) \to \mathsf{CTy} \triangleq\ & \lambda\, T.\, (\forall\, A.\, A \to T\ A)\\ & \&(\forall\, A\, A'.\ \mathsf{Thk}\ (T\ A) \to \mathsf{Thk}\ (A \to T\ A') \to T\ A')\end{aligned}$$

In examples, we will use return and bind as the names of the two projections. The main difference here from ordinary monads is that we define the type constructor T takes value types to *computation* types, similar to the Ret type. Our intuition for relative monads is that they generalize the Ret type: while Ret computations can return to their opaque continuation, T computations can return but may have other capabilities as well such as performing effects. We can thus think of a relative monad as defining a *custom* type of continuations, which may differ in what capabilities it provides besides the two basic operations of (1) returning to the current continuation and (2) binding a return value to construct a larger continuation from a smaller one.

[2] once we have introduced relative monads, these can be seen as typed using the continuation relative monad

Just as with monads, there are relative monad *laws*.

Definition 3.1. A monad implementation is *lawful* if it satisfies the following equational principles:

(1) **left unit law:** for any $a : A$ and $f : \mathsf{Thk}\ (A \to T\ A')$,

$$!\,\mathtt{bind}\ A\ A'\ \{\ !\,\mathtt{return}\ A\ a\}\ f \equiv\ !\,f\ a$$

(2) **right unit law:** for any $m : \mathsf{Thk}\ (T\ A)$,

$$!\,\mathtt{bind}\ A\ A\ m\ \{\ !\,\mathtt{return}\ A\} \equiv\ !\,m$$

(3) **associativity law:** for any $f : \mathsf{Thk}\ (A \to T\ A')$, $g : \mathsf{Thk}\ (A' \to T\ A'')$, and $m : \mathsf{Thk}\ (T\ A)$,

$$!\,\mathtt{bind}\ A'\ A''\ \{\ !\,\mathtt{bind}\ A\ A'\ m\ f\}\ g \equiv\ !\,\mathtt{bind}\ A\ A'\ m\ \{\lambda\, x.\ !\,\mathtt{bind}\ A'\ A''\ \{\ !\,f\ x\}\ g\}$$

(4) **linearity law:** for any $tm : \mathsf{Thk}\ (\mathsf{Ret}\ (\mathsf{Thk}\ (T\ A)))$,

$$\mathsf{do}\ m \leftarrow\ !\,tm\,;\ !\,\mathtt{bind}\ A\ A'\ m \equiv\ !\,\mathtt{bind}\ A\ A'\ \{\mathsf{do}\ m \leftarrow\ !\,tm\,;\ !\,m\}$$

The first three of these are analogous to the ordinary monad laws. The left unit law says that returning to a continuation constructed by `bind` is the same as calling the continuation directly. The right unit law says that constructing a continuation that always returns is equivalent to using the ambient continuation. The associativity law says that composing multiple continuations with bind is associative. The fourth, the linearity law, is a new law that is necessary in a system like CBPV which may have ambient effects, as opposed to ordinary monads which are defined in a "pure" language. Intuitively, bind being linear states that it uses its first input *strictly* and never again. For instance, if bind simply pushes something onto the stack and executes the thunk, then it will be linear. The precise formulation used here says that for `tm` a "double thunk", it doesn't matter if we evaluate `tm` first, and bind the resulting thunk or if we bind a thunk that will evaluate `tm` each time it is called. This formulation is taken from prior work on CBPV [Levy 2017; Munch-Maccagnoni 2014]. One motivation for why the linearity rule is necessary is that we want a relative monad to "overload" not just the `do` notation that the primitive `Ret` type provides, we also want all of the *reasoning principles* that the `Ret` provides to hold for relative monads. And the bind for `Ret` simply pushes a continuation onto the stack and executes the input, so it is linear.

3.2 Exception Monads

The first example of a relative monad is the simplest one: `Ret` itself.

$$\begin{aligned}
\mathsf{def}\ \mathtt{mret} : \mathsf{Monad\ Ret} \triangleq\ &\{\ \mathsf{comatch}\\
&|\ .\mathtt{return}\ \Rightarrow\ \Lambda\, A.\ \lambda\, a.\ \mathsf{ret}\ a\\
&|\ .\mathtt{bind}\ \Rightarrow\ \Lambda\, A\, A'.\ \lambda\, m\, f.\ \mathsf{do}\ a\ \leftarrow\ !\,m;\ !\,f\ a\ \}
\end{aligned}$$

The `return` and `bind` for this monad are really just primitives in CBPV. In fact, the `Ret` monad plays the same role in CBPV that the `Identity` monad does in Haskell, representing "no effects" or at least no *more* effects than the ambient notion of effect allowed in the language.

Next, we consider three different "exception" monads in CBPV, whose type constructors are given in Figure 7.

The first, `Exn` is the most direct translation of Haskell's `Either` type. Because we are constructing a computation, we wrap the sum in a `Ret`, meaning that the computation passes the `Either` typed value to the continuation stored on top of the stack. A downside of this formulation is that every `bind` pattern matches on the sum, causing branching even though the exception "handler" doesn't change.

```
def Exn ≜ λ E A. Ret (E + A)
def mexn ≜ { Λ (E : VTy). comatch
    | .return A a ⇒ ret (inj₁(a))
    | .bind A A′ m f ⇒
        do a? ← ! m;
        match a?
        | inj₀(e) ⇒ ret (inj₀(e))
        | inj₁(a) ⇒ ! f a }
```

```
def ExnK ≜ λ E A. ∀ R : CTy. Thk(E → R) → Thk(A → R) → R
def mexnk ≜ { Λ (E : VTy). comatch
        | .return ⇒ Λ A. λ a.
            Λ R. λ ke ka. ! ka a
        | .bind ⇒ Λ A A′. λ m f.
            Λ R. λ ke ka. ! m R ke
              { λ a. ! f a R ke ka } }
```

```
def ExnDe ≜ νExnDe.λ E A.
                ∀ (E′ : VTy). Thk (A → ExnDe E′ A) → ExnDe E′ A
                &∀ (A′ : VTy). Thk (A → ExnDe E A′) → ExnDe E A′
                &Ret (E + A)
def mexnde ≜ { fix mexnde. Λ (E : VTy). comatch
                | .return ⇒ Λ A. λ a. comatch
                    | .try ⇒ Λ E′. λ k. ! mexnde E′ .return A a
                    | .kont ⇒ Λ A′. λ k. ! k a
                    | .done ⇒ ret (inj₁(a))
                | .bind ⇒ Λ A A′. λ m f. ! m .kont A′ f }
```

Fig. 7. Three Exception Monads

The downsides of the first design are addressed by the second implementation: the Church encoded exception type ExnK. This is the well-known technique of "double-barreled" continuation-passing style where two continuations are passed in: one for errors and one for "success"[Thielecke 2001]. The stack for this computation is one that contains two continuations, and a tail of the stack which has an abstract type R. Parametricity of the language ensures that this result type R cannot leak any information [Møgelberg and Simpson 2009]. We can easily tell from the implementation of the monad interface that different from the last solution, no intermediate data constructor is produced, eliminating unnecessary conditional branching as the control flow is carried out by invoking the correct continuation directly. As we are in a continuation-passing style now, the stack grows only by the continuations themselves growing bigger and bigger along the way, eventually being run when a return or raise is executed.

A third method for implementing raising of exceptions is via explicit stack-walking to find the nearest enclosing exception handler. We can encode this structure as well in CBPV by using a coinductive codata type, which we can view as "defunctionalizing" the continuations into explicit stack frames [Reynolds 1972; Wand 1980]. By removing the unnecessary pair of handlers in favor of dedicated handlers, `bind` is simply pushing `.kont` onto the stack, but `return` and `fail` inspect and traverse the stack of handlers, and apply them accordingly. The user push `.try` handlers onto the stack to perform error handling.

We can verify that the first two exception monads we provided satisfy the monad laws using CBPV $\beta\eta$ equality and parametricity, but surprisingly, the defunctionalized exception monad implementation does not! The reason is that by defunctionalizing the stack of continuations, we expose low-level details and so "non-standard" computations can inspect the stack and observe, for instance, the number of frames pushed onto the stack. As a result, while the left unit and linearity laws hold, the right unit and associativity laws fail. We provide an explicit counter-example in the appendix which is implemented similarly to `abort` and counts the number of stack frames that it encounters.

The root cause of this lawlessness is that problematic programs involve unrestricted manipulations on the stack frame, yet a canonical instance of the exception monad, when executed, should only compute its result -either an exception or a valid value-and react to one of the destructors correspondingly. We henceforth propose a canonicity condition for these instances as shown below, where `M` is an arbitrary computation of type $\texttt{Exn}\ E\ A$.

$$\begin{aligned} M \equiv\ & \mathsf{do}\ x \leftarrow M\,; \\ & \mathsf{match}\ x \\ & \mid \mathsf{inj}_0(e) \Rightarrow\ !\,\texttt{fail}\ e \\ & \mid \mathsf{inj}_1(a) \Rightarrow\ !\,\texttt{return}\ a \end{aligned}$$

We verify in Appendix C that this condition ensures the monad laws hold. This is of course a strong restriction, and essentially ensures the implementor of a `Exn E A` computation only interacts with the monad by calling `fail`, `return`, and `bind`,

3.3 More Example Relative Monads

Next we demonstrate how to adapt several classic example monads. Here we list the type signatures of the monads that one can implement in CBPV. We omit most of the implementations of the monads due to their similarity to the examples of exception monad shown, but they are included in the supplementary material.

We give the types of the two continuation monads `Kont` and `PolyKont` and state monad `State` in Figure 8. The first continuation monad `Kont` is the one with fixed answer type `R`. The intuition is that a stack for `Kont` is an `R` stack with a continuation on top. Since `nR` is fixed, the computation can interact with the stack before invoking the continuation (if ever). For instance, by picking `R` to be `OS`, we can get Zydeco's analog of Haskell's `IO` monad. The second continuation monad `PolyKont` looks similar at first to `Kont` but the `R` parameter is universally quantified. Since the `R` is universally quantified, by parametricity the computation cannot interact with the `R` stack *except* by calling the continuation. For this reason, using parametric reasoning it can be proven that `PolyKont` is equivalent to `Ret`, and so we might call `PolyKont` the church-encoded `Ret` type [Møgelberg and Simpson 2009; Reynolds 1983]. For this reason it is also possible to leave the `Ret` type out of CBPV as a primitive type entirely, since it can be defined as this Church encoding. Finally, the state monad provides a "thread-local" state that can be read and mutated by the computation. Reading off of the stack semantics, the state is conceptually updated "in-place".

$$\begin{aligned} \texttt{def Kont} &\triangleq \lambda\,(R : \mathsf{CTy})\ (A : \mathsf{VTy}).\ \mathsf{Thk}\ (A \to R) \to R \\ \texttt{def PolyKont} &\triangleq \lambda\,A : \mathsf{VTy}.\ \forall\, R : \mathsf{CTy}.\ \mathsf{Thk}\ (A \to R) \to R \\ \texttt{def State} &\triangleq \lambda\,(S : \mathsf{VTy})\ (A : \mathsf{VTy}).\ \forall\, R : \mathsf{CTy}.\ \mathsf{Thk}\ (A \to S \to R) \to S \to R \end{aligned}$$

Fig. 8. The Types of State Monad and Continuation Monads in Zydeco

Next, we turn to a general class of monads, *free monads* generated by a collection of operations, in the sense of algebraic effects [Plotkin and Power 2001]. Free monads give a way to take an arbitrary collection of desired syntactic "operations" and provide a monad `T` that supports those operations. Here each operation is specified by two value types: a domain `A` and codomain `A'`. Then the free monad T should be the minimal monad supporting for each operation a function $\texttt{Thk}(A \to T\ A')$. For instance, a printing operation would be an operation $\texttt{Thk}(\texttt{String} \to T\ \texttt{Unit})$ or a random choice operation would be $\texttt{Thk}(\texttt{Unit} \to T\ \texttt{Bool})$. Typically free monads are constructed as a type of *trees* inductively generated by return as a leaf of the monad, and a node constructor for each operation. This construction could be carried out to construct a monad using recursive value types, and then compose this with Ret to get a relative monad. On the other hand, we can construct this directly as a computation type using a similar Church-encoding trick to what we have seen thus far. For instance, here is the type constructor for a free monad supporting print and random choice operations, which can obviously be generalized to any number of operations:

$$\begin{aligned}
\text{def } \texttt{KPrint} &\triangleq \lambda\, R : \mathsf{CTy}.\ \texttt{Thk}\ (\texttt{String} \to \texttt{Kont}\ R\ \texttt{Unit}) \\
\text{def } \texttt{KFlip} &\triangleq \lambda\, R : \mathsf{CTy}.\ \texttt{Thk}\ (\texttt{Unit} \to \texttt{Kont}\ R\ \texttt{Bool}) \\
\text{def } \texttt{PrintFlip} &\triangleq \lambda\, A.\ \forall\, R : \mathsf{CTy}.\ \texttt{KPrint}\ R \to \texttt{KFlip}\ R \to \texttt{Kont}\ R\ A
\end{aligned}$$

An intuition for this construction is that a computation in the free monad is one that takes in any "interpretation" of the operations as acting on some result type R and an A-continuation for R and returns an R. Again polymorphism ensures that all such a computation can do is perform some sequence of operations and return. We provide the full definition of the monad structure for PrintFlip as well as demonstrate that it supports Print and Flip operations in the appendix.

4 MONADIC BLOCKS AND THE ALGEBRA TRANSLATION

While relative monads provide a nice interface for user-defined effects implemented on the stack, their ergonomics leave a bit to be desired, as writing programs using explicit calls to bind leads to a CPS-like programming style. In Haskell, this is solved using the do notation which allows programming in a kind of embedded CBV sub-language. In a CBPV calculus, there is already a primitive do notation, which works for one particular relative monad: Ret. Then the analog of Haskell's do notation would be a way to *overload* the built-in do to work with an arbitrary relative monad. However, there is a major difference between the do for Ret and the bind for a relative monad: in the bind for a relative monad T, the continuation for the bind has result type T A, but the built-in do allows for the result type of the continuation to be of arbitrary computation type B.

At first look, then, it seems like an overloaded do would need to be much more restrictive than the built-in one. Miraculously, this is not the case! Any relative monad can be used to interpret code written with *unrestricted* CBPV do notation, using the primitive bind when the continuation is of the form T A, but using a different, type-dependent operation when the continuation has a different type B. This type-dependent operation is called an *algebra* of the monad, and we show in this section that *all* type constructors in CBPV extend to algebra constructors, allowing for a type-directed translation of unrestricted CBPV do notation. As an extended example, we provide a modified version of our earlier sum_and_print example from Figure 6 in Figure 16 that uses monadic blocks to introduce an exception effect inside the loop in the example.

4.1 Algebras of Relative Monads

Intuitively, an algebra is a version of bind that works with an output type not necessarily of the form T A. In terms of stacks, an algebra for a type B is a way of composing B stacks with T A continuations. Since algebras generalize bind, they have analogous laws. The only one that does not have an analog is the right unit law, which is specific to monadic result type.

Definition 4.1. Algebras for a monad are given by the type

$$\text{def } \texttt{Algebra} \triangleq \lambda\ (T : \mathsf{VTy} \to \mathsf{CTy})\ (R : \mathsf{CTy}).\ \forall\, A : \mathsf{VTy}.\ \texttt{Thk}\ (T\ A) \to \texttt{Thk}\ (A \to R) \to R$$

An algebra $\texttt{bindA}$ is *lawful* if it satisfies the following equational principles:

(1) **left unit law:**

$$!\,\texttt{bindA}\,A\;\{\,!\,\texttt{return}\,A\,a\}\;f \equiv\,!\,f\;a$$

for any $a : A$ and $f : \texttt{Thk}\;(A \to B)$,

(2) **associativity law:**

$$!\,\texttt{bindA}\,A'\;\{\,!\,\texttt{bind}\,A\,A'\;m\,f\}\;g \equiv\,!\,\texttt{bindA}\,A\,m\;\{\lambda\,x.\,!\,\texttt{bindA}\,A'\;\{\,!\,f\;x\}\;g\}$$

for any $f : \texttt{Thk}\;(A \to T\;A')$, $g : \texttt{Thk}\;(A' \to B)$, and $m : \texttt{Thk}\;(T\;A)$,

(3) **linearity law:**

$$\texttt{do}\;m \leftarrow\,!\,tm\,;\,!\,\texttt{bindA}\,A\,m \equiv\,!\,\texttt{bindA}\,A\;\{\texttt{do}\;m \leftarrow\,!\,tm\,;\,!\,m\}$$

for any $tm : \texttt{Thk}\;(\texttt{Ret}\;(\texttt{Thk}\;(T\;A)))$.

Then as expected, the built-in CBPV do notation ensures that *all* computation types B carry a built-in algebra structure `Algebra Ret B`:

$$\texttt{def alg\_ret} : \forall R.\texttt{Algebra}\,\texttt{Ret}\,R \;\triangleq\; \{\,\Lambda\,R.\,\Lambda\,Z.\,\lambda\,mz\,k.\;\texttt{do}\;z\;\leftarrow\,!\,mz;\;\,!\,k\;z\,\}$$

Additionally, for every monad T we trivially get an algebra structure `Algebra T (T A)` for any A using `bind` of the monad.

$$\texttt{def alg\_mo} : \forall T.\texttt{Monad}\,T \to \forall X.\texttt{Algebra}\,T\,X \;\triangleq\; \{\,\Lambda\,T.\,\lambda\,mo.\,\Lambda\,X.\,\Lambda\,Z.\,\lambda\,mz\,k.\,!\,mo\;.\texttt{bind}\;mz\;k\,\}$$

What's more, each of the type constructors comes with a canonical way of lifting algebra structures, that is, if the arguments are algebras, then the constructed type is as well. Here is the constructor for $\to$, the others are included in the appendix.

$$\begin{aligned}\texttt{def alg\_arrow} :&\forall T.\forall A.\forall B.\texttt{Algebra}\,T\,B \to \texttt{Algebra}\,T\,(A \to B) \;\triangleq\\ &\{\,\Lambda\,T.\,\Lambda\,T\,A\,B.\,\lambda\,alg : \texttt{Thk}\;(\texttt{Algebra}\,T\,B).\\ &\qquad \Lambda\,Z.\,\lambda\,mz\,f.\,\lambda\,a.\,!\,alg\;mz\;\{\,\lambda\,z.\,!\,f\;z\;a\,\}\,\}\end{aligned}$$

This algebra works by popping the A value off of the stack, and then invoking the algebra for B with a continuation that passing A to the original continuation. The algebra lifting for &, Top, ν are simple to define and provided in the appendix. This means complex types like $A \to^{+} B$ we defined earlier can have algebra structures derived in a type-directed manner.

The last computation type connective is $\forall$. It is not the case that arbitrary $\forall$ quantified types can be given algebra structure, for example, how would we construct an algebra for $\forall R.\texttt{Thk}R \to R$? Instead what we can support is quantifiers that quantify over *both* a type, *and* an algebra structure. For example $\forall R.\texttt{Algebra}\;T\;R \to \texttt{Thk}R \to R$, in which case the algebra structure can be defined, using the passed in algebra structure for R. This process becomes more complicated if R is not a computation type, but a higher-kinded type, in which case we need to pass in not an algebra for R, but some kind of "algebra constructor" whose structure is derived from the kind of R. We make this precise in the next section.

4.2 Monadic Blocks

The monadic blocks are introduced in Figure 9 as a new constructor on computations. Given a *closed* computation M, we can construct a computation $\mathcal{M}(M)$ which is like the original M except that it is *polymorphic* in the choice of underlying monad. That is, everywhere the original M used Ret, $\mathcal{M}(M)$ will use the passed in monad T. Additionally, as discussed in the previous section, everywhere M quantified over types, $\mathcal{M}(M)$ must quantify over *both* a type and a corresponding *structure*. This requires that the output of M be translated to $\lfloor\,M\,\rfloor$, which we will describe shortly.

$$\frac{\cdot\,;\,\cdot\ \vdash M : B}{\Delta;\Gamma \vdash \mathcal{M}(M) : \forall\,(T : \mathsf{VTy} \to \mathsf{CTy}).\ \mathsf{Thk}\ (\mathsf{Monad}\ T) \to \lfloor\, B \,\rfloor}\ [\textsc{TyMoBlock}]$$

Fig. 9. Monadic Blocks

It may seem overly limiting to require M to be a *closed* computation. However some limitations on M are necessary: M cannot for instance use arbitrary computation type variables, as it would require that these support algebras of the given monad. In practice, we can work around this restriction by making M a function type, then the translation type $\lfloor\, B \,\rfloor$ will also be a function type, and any type variables that M requires will then require algebra structures as well.

Next, we define the type $\lfloor\, B \,\rfloor$ of a monadic block in Figure 10. We call this the "carrier translation" as it takes every type in CBPV to the type of "carriers" for an algebraic structure that will be constructed later. We use dependently typed syntax to give a specification for the well-typedness property of this translation, using curly braces in the style of Agda to denote inferable arguments. The carrier translation takes in a type $\Delta \vdash S : K$ and returns back a type of the same kind, but requiring that a type constructor T (corresponding to the monad) additionally be in scope. For most types, the translation is simply homomorphic and so we elide these for now and include them in the appendix. We show the interesting cases: `Ret` is translated to the provided monad T, and $\forall, \exists$ are translated to require or provide respectively a *structure* $\mathrm{Sig}_K(X)$ for the quantified type. The type $\mathrm{Sig}_K(X)$ is defined next, it defines for each kind a computation type which is the "signature" type for a structure that will be required for all types of that kind, again with a monad T assumed to be in scope. For computation types, that structure is that of an algebra of the monad. For value types, the structure is trivial, so we use `Top`. For function kinds, we need a corresponding "structure constructor": given any structure on the input, we require structure on the output. Additionally, we extend the notion of signature from kinds to type environments: given a type environment Δ, we return a value environment $\mathrm{Sig}(\Delta)$ of thunks of structures satisfying the signatures for the corresponding variables in Δ.

4.3 The Algebra Translation

To implement monadic blocks, we define a translation from CBPV to CBPV that translates all monadic blocks to ordinary CBPV code. This proceeds in two phases: first there is a translation $\lfloor\, M \,\rfloor$ that translates a closed program that does *not* use monadic blocks into one that is polymorphic in the underlying monad. Second, we have a translation $[\![\, M \,]\!]$ that uses $\lfloor - \rfloor$ to translate away all uses of monadic blocks. This is defined homomorphically on all structures except monadic blocks which are translated as

$$[\![\, \mathcal{M}(M) \,]\!] \ := \ \Lambda\,(T : \mathsf{VTy} \to \mathsf{CTy}).\ \lambda\,(mo : \mathsf{Thk}\ (\mathsf{Monad}\ T)).\ \lfloor\, [\![\, M \,]\!] \,\rfloor$$

The term translation for values and computations is given in Figure 11. The type-preservation property of the translation is that given an *open* value/computation, we construct a value/computation whose output type is given by the carrier translation, but under a context extended with a monad T as well as structures for al of the free type variables $\mathrm{Sig}(\Delta)$, with the free value variables translated according to the carrier translation $\lfloor\, \Gamma \,\rfloor$. The term translation elaborates ret to the return of the provided monad and elaborates do-bindings to use the algebra constructed by the structure translation. The $\forall$ and $\exists$ cases are translated to be keeping the thunked structure around when instantiating the type variable X, which is bound to str_X whenever the X is bound.

The structure translation is given in Figure 12. The translation $\mathrm{Str}(S)$ takes a type (constructor) $S : K$ and generates a computation of type $\mathrm{Sig}_K(\lfloor\, S \,\rfloor)$ in the target language, scoped by the

$$\begin{aligned}
\lfloor \cdot \rfloor &: \forall \{K\}\, \{\Delta\} \to (\Delta \vdash \_ : K) \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash \_ : K)\\
\lfloor \mathtt{Ret} \rfloor &:= T\\
\lfloor \forall X : K.\, B \rfloor &:= \forall X : K.\, \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \to \lfloor B \rfloor\\
\lfloor \exists X : K.\, A \rfloor &:= \exists X : K.\, \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \times \lfloor A \rfloor\\
&\ \cdots
\end{aligned}$$

$$\begin{aligned}
\mathrm{Sig} &: \forall \{K\}\, \{\Delta\} \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash K) \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash \mathsf{CTy})\\
\mathrm{Sig}_{\mathsf{CTy}}(B) &:= \mathtt{Algebra}\ T\ B\\
\mathrm{Sig}_{\mathsf{VTy}}(A) &:= \mathtt{Top}\\
\mathrm{Sig}_{K_0 \to K}(S) &:= \forall X : K_0.\, \mathsf{Thk}\ (\mathrm{Sig}_{K_0}(X)) \to \mathrm{Sig}_K(S\ X)
\end{aligned}$$

$$\begin{aligned}
\mathrm{Sig} &: (\Delta : \mathit{TEnv}) \to \Delta \vdash \mathit{VEnv}\\
\mathrm{Sig}(\cdot) &:= \cdot\\
\mathrm{Sig}(\Delta, X : K) &:= \mathrm{Sig}(\Delta), \mathit{str}_X : \mathsf{Thk}\ (\mathrm{Sig}_K(X))
\end{aligned}$$

Fig. 10. Signature Translation and Carrier Translation

additional monad type constructor T and a monad instance *mo*, as well as all of the assumed structures for any free type variables. For value type constructors, the resulting structure is defined trivially. Type operators like Thk or $(\times)$ takes the type arguments and its structure first before generating the algebra for the value or computation type base cases. As for the computation types, we use the lifting of algebras described earlier in this section. In our cases, the parameter $mz : \mathsf{Thk}\ (T\ Z)$ is the thunked monad value, and f is the continuation from Z to the translated type. The $\forall X : K.\, B$ case is a base computation type, so it starts with parameters Z, mz and f as part of the algebra definition, leaving a body of type $\lfloor \forall X : K.\, B \rfloor$, which is $\forall X : K.\, \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \to \lfloor B \rfloor$. After introducing the type quantifier X and its structure str_X into the scope, we can now generate the algebra structure of B, pass in all the arguments we received along the way, and use type application in the continuation. The $\nu\, Y : K.\, B$ case is a bit more complicated, but the same idea applies. We first introduce a fixed point *str* for the generated algebra structure, introduce the algebra parameters, and then construct a term of the type $\lfloor \nu\ Y : K.\, B \rfloor$ by using roll. When we reach the recursive call $\mathrm{Str}(B)$, we've introduced both a abstract type variable Y and a variable str_Y for the the structure of Y. After substituting them with $\nu\, X : K.\, B$ and str_Y accordingly, we can pass in the arguments as usual and finally unroll it in the continuation. At last we have the type variable, type constructor, and type instantiation cases. The type variable case X is a simple lookup of str_X in the value environment. Since $X \in \Delta$ implies $str_X \in \mathrm{Sig}(\Delta)$, str_X is guaranteed to be in the value environment. The definition of the type constructor case satisfies its typing requirement since

$$(\Lambda\, X : K_0.\, \lambda\, str_X.\, \mathrm{Str}(S)) : (\forall\, X : K_0.\, \mathsf{Thk}\ (\mathrm{Sig}_{K_0}(X)) \to \mathrm{Sig}_K(S))$$

holds. Together with the type instantiation case, the structure of X is passed in to the structure of S as function arguments.

5 MONAD TRANSFORMERS

Next, we turn to how to adapt monad *transformers* to relative monad transformers. A monad transformer, defined in Figure 13 is a type $T : (\mathsf{VTy} \to \mathsf{CTy}) \to \mathsf{VTy} \to \mathsf{CTy}$ such that for any

$$\begin{aligned}
\lfloor \cdot \rfloor \;&:\; \forall\,\{A\}\,\{\Delta\}\,\{\Gamma\} \\
&\quad \to (\,\Delta;\Gamma \vdash A\,) \\
&\quad \to (\,T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \\
&\qquad ;\; mo : \mathsf{Thk}\ (\mathtt{Monad}\ T), \mathrm{Sig}(\Delta), \lfloor \Gamma \rfloor \vdash \lfloor A \rfloor\,) \\
\lfloor x \rfloor \;&:=\; x \\
\lfloor (S,V) \rfloor \;&:=\; (\lfloor S \rfloor, (\{\mathrm{Str}(S)\}, \lfloor V \rfloor)) \\
&\ldots
\end{aligned}$$

$$\begin{aligned}
\lfloor \cdot \rfloor \;&:\; \forall\,\{B\}\,\{\Delta\}\,\{\Gamma\} \\
&\quad \to (\,\Delta;\Gamma \vdash B\,) \\
&\quad \to (\,T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \\
&\qquad ;\; mo : \mathsf{Thk}\ (\mathtt{Monad}\ T), \mathrm{Sig}(\Delta), \lfloor \Gamma \rfloor \vdash \lfloor B \rfloor\,) \\
\lfloor \mathsf{let}\ (X,x) = V\ \mathsf{in}\ M \rfloor \;&:=\; \mathsf{let}\ (X,p) = \lfloor V \rfloor\ \mathsf{in}\ \mathsf{let}\ (str_X, x) = p\ \mathsf{in}\ \lfloor M \rfloor \\
\lfloor \mathsf{ret}\ V \rfloor \;&:=\; !\,mo\ \mathtt{.return}\ \lfloor A \rfloor\ \lfloor V \rfloor \\
&\qquad \text{where } \Delta;\Gamma \vdash V : A \\
\lfloor \mathsf{do}\ x \leftarrow M_0\,;M \rfloor \;&:=\; \mathrm{Str}(B)\ \lfloor A \rfloor\ \{\lfloor M_0 \rfloor\}\ \{\lambda\, x.\ \lfloor M \rfloor\} \\
&\qquad \text{where } \Delta;\Gamma \vdash M_0 : \mathtt{Ret}\ A \text{ and } \Delta;\Gamma \vdash M : B \\
\lfloor \Lambda\, X : K.\ M \rfloor \;&:=\; \Lambda\, X : K.\ \lambda\, str_X.\ \lfloor M \rfloor \\
\lfloor M\ S \rfloor \;&:=\; \lfloor M \rfloor\ \lfloor S \rfloor\ \{\mathrm{Str}(S)\} \\
&\ldots
\end{aligned}$$

Fig. 11. Term Translations (Selected)

monad M, $T(M)$ is a monad, and we support a `lift` function that turns embeds $M\ A$ computations into $T\ M\ A$ computations. So a monad transformer is a way to "add" effects to an arbitrary "base" monad M. Monad transformers provide one way to combine multiple effects by "stacking" multiple monad transformers.

In practice many monads in pure languages are known to extend to monad transformers, but there is no automatic method of doing so. The reason that this cannot be automatic is that a monad transformer must be an "effect aware" version of the original monad, but the original monad was written in a pure language. In a CBPV setting, however, all relative monads we define are in a sense "effect aware" in that they all can interact with the "base" effect `Ret`. In fact we can use our monadic blocks to derive *automatically* an implementation of a relative monad transformer from an implementation of a relative monad.

In Figure 14 we show how to use monadic blocks to implement monad transformers. First, we are given a closed monad implementation M : `Monad` S. We take in another monad implementation T as an argument. Then inside the monadic block, we implement the monad structure for M, as well as the lift. Inside the monadic block, implementing lift is just an instance of the fact that *all* computation types are algebras with respect to the `Ret`! Lastly, we must make a minor change: because the definition of monad and lift both quantify over value types, we must instantiate

$$
\begin{array}{rcl}
Str & : & \forall\, \{K\}\ \{\Delta\} \to (\Delta \vdash S : K) \\
& & \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta;\ mo : \mathsf{Thk}\ (\mathsf{Monad}\ T), \mathrm{Sig}(\Delta) \vdash \mathrm{Sig}_K(\lfloor S \rfloor)) \\
\mathrm{Str}(\mathsf{Thk}) & := & \Lambda\, X.\ \lambda\ \_.\ \mathrm{comatch}\ | \\
\mathrm{Str}((\times)) & := & \Lambda\, X.\ \lambda\ \_.\ \Lambda\, Y.\ \lambda\ \_.\ \mathrm{comatch}\ | \\
\dots & & \\
\mathrm{Str}(\mathsf{Ret}) & := & \Lambda\, X.\ \lambda\ \_.\ \Lambda\, Z.\ \lambda\ mz\ (f : \mathsf{Thk}\ (Z \to T\ X)). \\
& & !\ mo\ \ .\mathtt{bind}\ Z\ X\ mz\ f \\
\mathrm{Str}((\to)) & := & \Lambda\, X.\ \lambda\ \_.\ \Lambda\, Y.\ \lambda\ alg_Y.\ \Lambda\, Z.\ \lambda\ mz\ (f : \mathsf{Thk}\ (Z \to \lfloor X \rfloor \to \lfloor Y \rfloor)). \\
& & \lambda\ (x : \lfloor X \rfloor).\ !\ alg_Y\ Z\ mz\ \{\lambda\ z.\ !\ f\ z\ x\} \\
\mathrm{Str}((\&)) & := & \Lambda\, X.\ \lambda\ alg_X.\ \Lambda\, Y.\ \lambda\ alg_Y.\ \Lambda\, Z.\ \lambda\ mz\ (f : \mathsf{Thk}\ (Z \to \lfloor X \rfloor\ \&\ \lfloor Y \rfloor)). \\
& & \mathrm{comatch} \\
& & |\ .0 \Rightarrow\ !\ alg_X\ Z\ mz\ \{\lambda\ z.\ !\ f\ z\ .0\} \\
& & |\ .1 \Rightarrow\ !\ alg_Y\ Z\ mz\ \{\lambda\ z.\ !\ f\ z\ .1\} \\
\mathrm{Str}(\forall\, X : K.\ B) & := & \Lambda\, Z.\ \lambda\ mz\ (f : \mathsf{Thk}\ (Z \to \forall\, X : K.\ \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \to \lfloor B \rfloor)). \\
& & \Lambda\, X : K.\ \lambda\ (str_X : \mathsf{Thk}\ (\mathrm{Sig}_K(X))).\ \mathrm{Str}(B)\ Z\ mz\ \{\lambda\ z.\ !\ f\ z\ X\ str_X\} \\
\mathrm{Str}(\nu\ Y : K.\ B) & := & \mathrm{fix}\ (str : \mathsf{Thk}\ (\mathtt{Algebra}\ T\ (\nu\ Y : K.\ \lfloor B \rfloor))). \\
& & \Lambda\, Z.\ \lambda\ mz\ (f : \mathsf{Thk}\ Z \to \nu\ Y : K.\ \lfloor B \rfloor). \\
& & \quad \mathrm{roll}(\mathrm{Str}(B)[(\nu\ Y : K.\ \lfloor B \rfloor)/Y][str/str_Y]\ Z\ mz\ \{\lambda\ z.\ \mathrm{unroll}(\ !\ f\ z\ )\}) \\
\mathrm{Str}(X) & := & !\ str_X \\
\mathrm{Str}(\lambda\, X : K.\ S) & := & \Lambda\, X.\ \lambda\ str_X.\ \mathrm{Str}(S) \\
\mathrm{Str}(S\ S_0) & := & \mathrm{Str}(S)\ \lfloor S_0 \rfloor\ \{\mathrm{Str}(S_0)\}
\end{array}
$$

Fig. 12. Structure Translation (Selected)

$$
\begin{array}{rcl}
\mathtt{def\ MonadTrans} & \triangleq & \lambda\ (F : (\mathsf{VTy} \to \mathsf{CTy}) \to \mathsf{VTy} \to \mathsf{CTy}).\ \mathrm{codata} \\
& & \quad |\ \mathtt{.monad}\ :\ \forall\, T : \mathsf{VTy} \to \mathsf{CTy}.\ \mathsf{Monad}\ (F\ T) \\
& & \quad |\ \mathtt{.lift}\ :\ \forall\ (T : \mathsf{VTy} \to \mathsf{CTy})\ (A : \mathsf{VTy}).\ \mathsf{Thk}\ (T\ A) \to F\ T\ A
\end{array}
$$

Fig. 13. The Interface of Monad Transformers in Zydeco

these with some trivial structures using a function `removeTriv` which η-expands the monad/lift definitions to fill in these trivial structures.

What are the resulting monad transformers we get in this manner? They are given by our carrier translation in the previous section: any uses of `Ret` are replaced by the argument monad and any uses of universal or existential types are enhanced to require or provide corresponding structures for those types. For instance, if we take our original exception monad `Exn` defined in terms of `Ret` can easily be turned into a monad transformer `ExnT` as `ExnT` $E\ T\ A = T(E + A)$. For more complicated, Church-encoded types such as `ExnK`, we derive a monad transformer that requires the result type to be an algebra of the monad parameter: `ExnKT` $E\ T\ A = \forall R.$`Algebra` $T\ R \to$

```
                          Assume ·; · ⊢ M : Monad S
def motrans′ M ≜ Λ T. λ mo. ! removeTriv {M(
                    let mo_f = {M} in comatch
                      | .monad ⇒ ! mo_f
                      | .lift ⇒ Λ Z. λ mz. do z ← ! mz ; ! mo_f .return Z z
                  ) T mo}
def removeTriv ≜ λ m. comatch
                    | .monad ⇒ comatch
                      | .return A ⇒ ! m .monad .return A triv
                      | .bind A A′ ⇒ ! m .monad .bind A triv A′ triv
                    | .lift A ⇒ ! m .lift A triv
        def triv ≜ {comatch |}
```

Fig. 14. Deriving Monad Transformers From Monads

$\mathsf{Thk}(E \to R) \to \mathsf{Thk}(A \to R) \to R$. The lift for these monads is derived completely mechanically by constructing a complex algebra structure for the monad.

6 FUNDAMENTAL THEOREM OF CBPV RELATIVE MONADS

In this section, we connect our development of relative monads in CBPV to the category-theoretic notion of relative monad, and prove the semantic analog of the algebra translation of the previous section, a theorem we call the *fundamental theorem of CBPV relative monads*. While we have presented the syntactic translation first, we note that we arrived at the mathematical version first and then adapted this to our syntactic calculus.

In order to reduce the mathematical complexity of this section, we will present a less general version of the theorem, one that applies only to models of *simply typed* CBPV, that is without polymorphism (∀/∃), recursive types (ν) or term-level fixed points. More general versions of the theorem are possible, such as a version for extensional dependently typed CBPV models, but the System F_ω style of polymorphism is a bit idiosyncratic to model semantically, since we can quantify over the *data* of relative monads and algebras, but not the *properties* that they satisfy the monad/algebra laws. Handling recursive types properly is difficult for different reasons as it would involve something like enrichment in domains or the use of guarded types/step-indexing to model. Instead of attempting to get the theorem that corresponds most closely to our syntactic presentation, we will instead work on the simply typed case, which matches our syntactic presentation closely and connects easily to existing results in category theory.

6.1 Models of Call-by-push-value

In order to state our fundamental theorem, we first need to define what a model of CBPV is. We will use a simple notion that is seemingly less general than prior work, we discuss the relationship to other models in Section 7. We start by defining a model for only the *judgments* (value/computation types and terms) of CBPV without any of the type connectives.

Definition 6.1. A judgmental CBPV model consists of

(1) A universe of value objects, that is a (large) set $\mathcal{V}_0$ and a function $\mathrm{el} : \mathcal{V}_0 \to \mathit{Set}$.
(2) A category $\mathcal{E}$ of computation objects and linear morphisms

(3) A functor $C : \mathcal{E} \to Set$ of computations.

For any such model, we define a category $\mathcal{V}$ to have $\mathcal{V}_0$ as objects and as morphisms $f \in \mathcal{V}(A, A')$ just functions $f : \mathrm{el}(A) \to \mathrm{el}(A')$. Further we define a profunctor $J : \mathcal{V}^{op} \times \mathcal{E} \to Set$ by $J(A, B) = \mathrm{el}(A) \to C(B)$. We will notate the functorial actions of this profunctor by composition $\circ$.

A judgmental CBPV model can model the basic notions of our CBPV calculus, but none of the type constructors. Value types A are interpreted as value objects $A \in \mathcal{V}_0$, value environments $\Gamma = x_1 : A_1, \ldots$ are interpreted as the Cartesian product $\mathrm{el}(\Gamma) = \mathrm{el}(A_1) \times \cdots$ with the empty context interpreted as a one-element set. Computation types are interpreted as computation objects $B \in \mathcal{E}_0$. Then values $\Gamma \vdash V : A$ are modeled as functions $V : \mathrm{el}(\Gamma) \to \mathrm{el}(A)$, and computations $\Gamma \vdash M : B$ are interpreted as functions $\mathrm{el}(\Gamma) \to C(B)$.

One difference between the syntax and semantics is that the semantics has primitive notions of pure morphism (function between value objects) and linear morphism (morphism between computation objects), whereas the syntax has a more restrictive notion of values and we only give a primitive syntax for stacks in the CES semantics. The correct interpretation is that the morphisms between value objects $\mathcal{V}(A, A')$ correspond in the syntax to *thunkable* terms $x : A \vdash M : \mathtt{Ret}\ A'$ and similarly that computation morphisms $\mathcal{E}(B, B')$ correspond to *linear* terms $z : \mathtt{Thk}\ B \vdash M : B'$, as described in [Munch-Maccagnoni 2014]. In our semantics, we will work with elements of $\mathcal{E}(B, B')$ where in our syntax we worked with a linearity restriction.

In order to model the type and term constructors of CBPV, a judgmental model must come with corresponding object and morphism constructors. One of the convenient features of our choice of model is that most type constructors in the syntax can be expressed as either coproducts or products:

Definition 6.2. (1) Let I be a set and $A : I \to \mathcal{V}_0$ a family of value objects. A coproduct value object is an object $\sum_{i \in I} A(i)$ with a natural isomorphism $\mathcal{V}(\sum_{i \in I} A(i), A') \cong \prod_{i \in I} \mathcal{V}(A(i), A')$.

(2) Let I be a set and $B : I \to \mathcal{E}_0$ a family of computation objects. A product computation object is an object $\&_{i:I} B(i)$ with a natural isomorphism $\mathcal{E}(B', \&_{i \in I} B(i)) \cong \prod_{i \in I} \mathcal{E}(B', B(i))$.

Definition 6.3. A CBPV model is a judgmental CBPV model with the following structures:

(1) (thunk objects) a functor $\mathtt{Thk} : \mathcal{E} \to \mathcal{V}$ and a natural isomorphism $\mathcal{V}(A, \mathtt{Thk}\ B) \cong \mathcal{J}(A, B)$
(2) (returner objects) a functor $\mathtt{Ret} : \mathcal{V} \to \mathcal{E}$ and a natural isomorphism $\mathcal{E}(\mathtt{Ret}\ A, B) \cong \mathcal{J}(A, B)$.
(3) (nullary, binary value coproducts) All coproducts value objects indexed by 0 and 2.
(4) (nullary value product) a value object 1 with $\mathrm{el}(1)$ having one element.
(5) (binary value products) For any $A, A' \in \mathcal{V}_0$, a value coproduct of the constant family $\lambda\_.A' : \mathrm{el}(A) \to \mathcal{V}_0$.
(6) (nullary, binary computation products) All product computation objects indexed by 0 and 2.
(7) (computation function types), For any $A \in \mathcal{V}_0, B \in \mathcal{E}_0$ a computation coproduct of the constant family $\lambda\_.B : \mathrm{el}(A) \to \mathcal{E}_0$.

By composing natural isomorphisms we get $\mathcal{E}(FA, B) \cong \mathcal{V}(A, UB)$, that is, U is a right adjoint functor to F.

6.2 Relative Monads, Algebras and the Fundamental Theorem

Next we recount the standard definition of a relative monad in category theory [Altenkirch et al. 2010].

Definition 6.4. Let $G : \mathcal{X} \to \mathcal{Y}$ be a functor. A monad relative to G consists of

(1) A function $T : \mathcal{X}_0 \to \mathcal{Y}_0$
(2) For each $x \in \mathcal{X}_0$, a *unit* $\eta_x : \mathcal{Y}(Gx, Tx)$

(3) For each morphism $f \in \mathcal{Y}(Gx, Tx')$, an *extension* $f^\dagger \in \mathcal{Y}(Tx, Tx')$
(4) Satisfying $\eta_x^\dagger = \mathrm{id}$, $f^\dagger \circ \eta = f$ and $(f^\dagger \circ g)^\dagger = f^\dagger \circ g^\dagger$ for all f, g

How is this related to our syntactic definition in Section 3? Our syntactic definition corresponds to the following semantic definition:

Definition 6.5. A *CBPV relative monad* consists of

(1) A function $T : \mathcal{V}_0 \to \mathcal{E}_0$
(2) For each $A \in \mathcal{V}_0$, an element $\eta_A : J\,A\,(TA)$
(3) For each $A, A' \in \mathcal{V}_0$ a function $-^\dagger : J\,A\,(TA') \to \mathcal{E}(TA, TA')$
(4) Satisfying $\eta^\dagger = \mathrm{id}$, $j^\dagger \circ \eta = j$ and $(j^\dagger \circ k)^\dagger = j^\dagger \circ k^\dagger$ for all j, k.

This has a fairly direct correspondence to our syntactic notion. First, the syntactic version of return has type $\mathsf{Thk}(A \to TA')$, which in the model is an element of $\mathsf{Thk}(A \to TA')$. But

$$\mathsf{Thk}(\&_{:\mathrm{el}(A)} TA') \cong \mathrm{el}(A) \to \mathrm{el}(\mathsf{Thk}(TA')) \cong \mathcal{V}(A, \mathsf{Thk}(T\,A')) \cong \mathcal{J}(A, TA')$$

Where the first equivalence follows because Thk is a right adjoint and therefor preserves products. The $-^\dagger$ operation then corresponds to the bind operation with arguments reordered, with the linearity assumption of the bind operation instead encoded by requiring that the $-^\dagger$ operation construct a morphism in $\mathcal{E}$. Note that because we are working with a model where $\mathcal{V}$ is a subcategory of sets, this notion of monads is automatically "strong" in that we can define a version of $-^\dagger$ that works in an arbitrary context $(\mathrm{el}(\Gamma) \to \mathcal{J}(A, TA')) \to (\mathrm{el}(\Gamma) \to \mathcal{E}(TA, TA')$ as simply $f^\dagger(\gamma) = (f(\gamma))^\dagger$.

How then does a CBPV relative monad relate to the standard notion? The definitions seem quite similar, the only difference is that a CBPV relative monad uses the profunctor J whereas the standard notion uses morphisms out of Gx. This formulation in terms of a profunctor has been identified in prior work [Arkor and McDermott 2023; Levy 2019]. Then we need only observe that by assumption we have a natural isomorphism $J(A, B) \cong \mathcal{E}(FA, B)$ and then the following is a straightforward consequence.

Lemma 6.1. *A CBPV relative monad is equivalent to a monad relative to* `Ret`

This equivalence allows us to adapt standard notions of algebra and homomorphism to the CBPV setting.

Definition 6.6. An *algebra* for a CBPV monad T consists of

(1) A carrier object $B \in \mathcal{E}_0$
(2) For every A a function $-^{\dagger_B} : J\,A\,B \to \mathcal{E}(TA, B)$
(3) Satisfying $j^{\dagger_B} \circ \eta = j$ and $(j^{\dagger_B} \circ k)^{\dagger_B} = j^{\dagger_B} \circ k^\dagger$

Given two algebras $(B, -^{\dagger_B})$ and $(B', -^{\dagger'_B})$, an algebra *homomorphism* is a morphism $\phi : \mathcal{E}(B, B')$ that preserves the extension operation $\phi(j^{\dagger_B}) = \phi(j)^{\dagger'_B}$. This defines a category $Alg(T)$ with a forgetful functor $U : Alg(T) \to \mathcal{E}$ that takes the carrier object of an algebra and the underling morphism of a homomorphism.

Then the fundamental theorem of relative monads says that this category of algebras can replace $\mathcal{E}$ and serve as a new model of CBPV. To prove this we need only show that this category of algebras can model the connectives of CBPV. Modeling the connectives on value objects comes essentially for free as we will use the same notion of value object. For the returner object, we use the *free algebra*, which is the semantic analog of the fact that every monad, when instantiated is an algebra for itself:

Definition 6.7. For a value object A, the free algebra is the T-algebra on TA defined by the $-^{\dagger}$ of the monad itself. This has the property that algebra homomorphisms $Alg(T)((TA, -x^{\dagger}), (B, -^{\dagger_B}))$ are naturally isomorphic to morphisms $J\,A\,B$.

The remaining computation object constructions, function objects and nullary and binary products, are all given by computation products, and the fact that $Alg(T)$ has these products follows from the following standard property of algebras of relative monads:

Lemma 6.2. *The forgetful functor $U : Alg(T) \to \mathcal{E}$ creates limits.*

Theorem 6.3 (Fundamental Theorem of CBPV Relative Monads). *Given a CBPV relative monad T, we construct a new CBPV model with*

(1) *$\mathcal{V}$ as in the original model, $\mathcal{E} = Alg(T)$ and $C = U \circ C$*
(2) *$\mathsf{Thk} = \mathsf{Thk} \circ U$, other value object connectives as in the original model*
(3) `Ret` *given by the free algebra, other computation connectives given by Lemma 6.2*

Our algebra translation is essentially a syntactic version of this construction: interpreting the syntax of CBPV into a new model where computation types carry an algebra structure.

To appreciate this theorem, let's consider two similar settings where the analogous theorem fails. First, consider the situation in Moggi's original work: a strong monad on a Cartesian closed category. This is a "pure" λ calculus with a monad on it. We can take the category of algebras of this monad, but this will not generally give back a Cartesian closed category. To get a Cartesian closed category of algebras the monad would need to strongly preserve Cartesian products $T(A \times A') \cong TA \times TA'$ which of the common examples in effects, is satisfied by the reader monad but is not satisfied by non-trivial exception, writer, state or continuation monads [Kock 1971].

Secondly, consider the enriched effect calculus, which adds a coproduct of computation types $\oplus$ and a "tensor" $A \oslash B$ to CBPV. Since these connectives are *coproducts* rather than products, the fact that the forgetful functor U creates limits does not help. In general, constructing even coproducts in a category is more complex, and involves *coequalizers*, which is a form of *quotient* typeAdamek and Koubek [1980].

6.3 Constructing Relative Monad Transformers from Relative Monads

Next, we analyze the semantic content of our construction of monad transformers from monads. Fundamentally there is something *syntactic* about this theorem: it says that any *definable* relative monad in CBPV extends to a definable monad transformer. But there is a semantic component to this construction, which is that we construct the transformer definition by interpreting the monad not in the original model, but in the algebra model. First, we provide some preliminary definitions:

Definition 6.8. A *monad homomorphism h* from $(T, \eta, -^{\dagger})$ to $(T', \eta', -^{\dagger'})$ consists of a natural transformation $h_A : \mathcal{E}(TA, T'A)$ that preserves the monad structure in that $h_A(\eta_A) = \eta'_A$ and $h_A(j^{\dagger}) = j^{\dagger'}$.

A *monad transformer* on a model of CBPV consists of a function F : RMonad $\to$ RMonad and a function lift taking any monad T to a monad homomorphism $\text{lift}_M : T \to F(T)$.

A *generic monad* $\tilde{T}$ is a function that takes any CBPV model to a relative monad on that model.

The idea of the monad transformer theorem is as follows: any syntactically definable CBPV relative monad, such as those we have defined in Section 3, gives rise to a relative monad in *any* model of CBPV. To turn a definable relative monad into a monad transformer, we interpret that monad in the category of algebras for the monad being transformed. We then observe that this gives rise to another monad in the original model with a monad homomorphism corresponding to lift. Note that even this does not capture the full strength of the syntactic version of our theorem, which provides a *definable* monad transformer from a definable monad.

Construction 6.1. *Let T be a relative monad on a CBPV model and let S be a relative monad on the algebra model. Then we can define a relative monad US on the original model. Further, any monad transformer $\phi : S \to S'$ extends to a monad transformer $U\phi : US \to US'$*

Proof. Define $US(A) = U(S(A))$, $\eta_{US} = \eta_S$ and $f^{\dagger}_{US} = U(f^{\dagger}S)$. Define $U\phi_A = U(\phi_A)$. □

Construction 6.2. *For any model of CBPV, the returner objects* $\texttt{Ret}$ *define a relative monad with the extension operator $-^{\dagger} : J(A, \texttt{Ret}\ A') \to \mathcal{E}(\texttt{Ret}\ A, \texttt{Ret}\ A')$ given by the natural isomorphism and the unit $\eta : J(A, \texttt{Ret}\ A)$ given by the unique element such that $\eta^{\dagger} = id$.*

This is the initial *monad in that for any other monad T there is a unique monad homomorphism from* $\texttt{Ret}$ *to T defined by $h(f) = (f \circ \eta_{\texttt{Ret}})^{\dagger}_{T}$.*

Theorem 6.4. *Any generic monad N gives rise to a monad transformer on any CBPV model.*

Proof. Let M be a monad on the given CBPV model. Interpret the generic monad N in the model of algebras of M. By Construction 6.2, there is a monad homomorphism from F to N in this model. By Construction 6.1, we can construct form this a monad homomorphism from M to UN. □

7 DISCUSSION AND FUTURE WORK

Programming with CBPV. Our Zydeco language and semantics is based on Levy's original CBPV syntax and stack-machine semantics [Levy 2001]. Downen and Ariola [Downen and Ariola 2018] further discussed how different calling conventions can be modeled as conservative extensions to Levy's CBPV through the shifts between positive and negative types, which is similar to our use of CBPV for encoding stack representations. Binder et al. [Binder et al. 2022] introduced the concept of data and codata symmetry and transformations back and forth between the two, named defunctionalization and refunctionalization.

Categorical models of CBPV. Our notion of CBPV model, in which values form a subcategory of sets is seemingly much more restrictive than Levy's original formulation of adjunction models or Curien, Fiore and Munch-Maccagnoni's (CFMM) simplification, which both model the value types in an arbitrary Cartesian category[Curien et al. 2016; Levy 2003]. In particular it would seem that the syntax of CBPV would not form a model in this sense. In CFMM's formulation the computation types are then modeled as a category enriched in presheaves on the category of values. But any CFMM model can be viewed as one of our models by interpreting "set" using the *internal language* of presheaves over value types. Then the value types do form a universe (the representable presheaves), and the category of computation types and linear morphisms then can be viewed as an ordinary category in this internal language. Then as long as we stick to constructive arguments (which we have done in this work), any mathematics that we do can be interpreted in presheaf categories. We refer interested readers to existing resources on the interpretation of mathematics in the internal language of presheaves [Maietti 2005].

Applications to Verified Compilation. In this work, for concreteness we have focused on a stack-based calculus, but much of our interest in this topic is on potential applications to verified compilation, where low-level representation decisions such as the representation of continuations and stack frames must be formalized and code verified to use this representation. We argue that there is not quite as large a gap between the CBPV-based calculus we have used here and low-level compiler IRs. Firstly, the CBPV calculus is closely related to common functional compiler IRs such as ANF and CPS [Appel 1992; Flanagan et al. 1993], and additionally to SSA form, the most commonly used compiler IR in industry today. The most thorough result about this relationship proves an isomorphism between focused CBPV terms and SSA control-flow graphs [Garbuzov et al. 2018]. Additionally, polarized calculi similar to CBPV have been used in compilation [Downen 2024; Downen et al. 2016] . However these intermediate representations typically abstract from explicit stack manipulation, and so CBPV is somewhat more general in its ability to describe stack structure in its type system. In this regard it is more similar to explicit stack-manipulating IRs such as the Stack-based Typed Assembly Language (STAL) [Morrisett et al. 2003]. STAL includes, like CBPV, a second "kind" of type, called stack types, and indeed their work includes examples of implementing exceptions using stack types that inspired some of our examples in Zydeco. For instance, they give encodings of double-barreled continuations that, other than specifying additional details such as the representation of closures and precisely which registers are used correspond directly to our encoding in CBPV. One feature that we have identified in our work that is absent from STAL is *recursive* computation types, in order to encode stack-walking.

Alternative Implementations of the Algebra Translation. The algebra translation for monadic blocks is carefully crafted to work exactly with the simple F_ω-style of CBPV calculus that we have presented here, but we discuss other programming language features would allow for different implementations. If our language supported *sigma kinds*, we could unify the signature and carrier translations by translating the kind CTy to the sigma kind $\sum_{B:\mathsf{CTy}} \texttt{ThkAlg}\ T\ B$. This would have the benefit that value types would not require any structure to be defined for them as we could simply translate VTy to VTy without requiring any structure, removing the need for the trivial `Top` structures to be defined or passed around. A downside is that to support full overloadability we would need to translate these sigma kinds as well, and that the type system as a whole would be more complex. It also may be possible to use Haskell-style typeclasses for algebras of a relative monad and use a metaprogramming tool like Haskell's `deriving` mechanism to implement our algebra translation.

An avenue for future work would be to extend our algebra translation to dependently typed CBPV, which would allow us to build the monad and algebras into the types. This should be closely related to the "weaning translation" for ∂-CBPV [Pédrot and Tabareau 2017, 2019]. In particular, the weaning translation is an adaptation of the algebra semantics of CBPV for an ordinary monad to the setting of intensional type theory, so our relative monad algebra semantics could possibly be combined to generalize this to a "relative weaning translation".

7.1 Relative Comonads

An obvious direction for future work would be to consider relative *comonads* in CBPV, which would dually be type constructors CTy $\rightarrow$ VTy with extra structure, with `Thk` being the prototypical example in the same way that `Ret` is the prototypical relative monad. In the same way that relative monads abstract the structure of continuations implemented on the stack to allow for additional "effects", relative comonads would abstract the structure of *closures* to allow for additional "coeffects" such as accessing metadata for the closure or availability of a destructor for an object. However it seems unlikely that it would support "comonadic blocks" via a "coalgebra" translation, since this would mean the category of coalgebras would be Cartesian, where in general it would likely only be a monoidal category.

8 DATA-AVAILABILITY STATEMENT

This paper comes with a complete implementation of the Zydeco language, which closely resembles the CBPV$_\omega^{\forall,\exists,\nu}$ calculus. The Zydeco language consists of a parser, a bidirectional type checker, and a stack-based environment-capturing interpreter. The implementation is written in Rust and is accompanied by functional examples of monadic blocks and algebra translation.

REFERENCES

Jiri Adamek and Vaclav Koubek. 1980. Are colimits of algebras simple to construct? *Journal of Algebra* 66, 1 (1980), 226–250. https://doi.org/10.1016/0021-8693(80)90122-2

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2005. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theor. Comput. Sci.* 342, 1 (2005), 149–172. https://doi.org/10.1016/J.TCS.2005.06.008

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2010. Monads Need Not Be Endofunctors. In *Foundations of Software Science and Computational Structures*. 297–311.

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

Nathanael Arkor and Dylan McDermott. 2023. The formal theory of relative monads. arXiv:2302.14014 [math.CT]

David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2022. Data-Codata Symmetry and its Interaction with Evaluation Order. https://doi.org/10.48550/arXiv.2211.13004 arXiv:2211.13004 [cs].

Pierre-Louis Curien, Marcelo P. Fiore, and Guillaume Munch-Maccagnoni. 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 44–56. https://doi.org/10.1145/2837614.2837652

Paul Downen. 2024. Call-by-Unboxed-Value. *Proc. ACM Program. Lang.* 8, ICFP, Article 265 (Aug. 2024), 35 pages. https://doi.org/10.1145/3674654

Paul Downen and Zena M. Ariola. 2018. Beyond Polarity: Towards a Multi-Discipline Intermediate Language with Sharing. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 119)*, Dan R. Ghica and Achim Jung (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:23. https://doi.org/10.4230/LIPIcs.CSL.2018.21

Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. 2016. Sequent calculus as a compiler intermediate language. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 74–88. https://doi.org/10.1145/2951913.2951931

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2014. The enriched effect calculus: syntax and semantics. *J. Log. Comput.* 24, 3 (2014), 615–654. https://doi.org/10.1093/LOGCOM/EXS025

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. 2018. Structural Operational Semantics for Control Flow Graph Machines. *CoRR* abs/1805.05400 (2018). arXiv:1805.05400 http://arxiv.org/abs/1805.05400

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph. D. Dissertation. Université Paris Diderot.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Anders Kock. 1971. Bilinearity and Cartesian Closed Monads. *Math. Scand.* 29, 2 (1971), 161–174. https://doi.org/10.7146/math.scand.a-11042

Paul Blain Levy. 2001. *Call-by-Push-Value*. Ph. D. Dissertation. Queen Mary, University of London, London, UK.

Paul Blain Levy. 2003. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science* 69 (2003), 248–271. https://doi.org/10.1016/S1571-0661(04)80568-1 CTCS'02, Category Theory and Computer Science.

Paul Blain Levy. 2017. Contextual isomorphisms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 400–414. https://doi.org/10.1145/3009837.3009898

Paul Blain Levy. 2019. What is a Monoid? (2019). https://conferences.inf.ed.ac.uk/ct2019/slides/83.pdf Category Theory 2019.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, San Francisco, California, United States, 333–343. https://doi.org/10.1145/199448.199528

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514. https://doi.org/10.1145/3009837.3009897

Maria Emilia Maietti. 2005. Modular correspondence between dependent type theories and categories including pretopoi and topoi. *Math. Struct. Comput. Sci.* 15, 6 (2005), 1089–1149. https://doi.org/10.1017/S0960129505004962

Rasmus Ejlers Møgelberg and Alex Simpson. 2009. Relational Parametricity for Computational Effects. *Log. Methods Comput. Sci.* 5, 3 (2009). http://arxiv.org/abs/0906.5488

Eugenio Moggi. 1991. Notions of computation and monads. *Inform. And Computation* 93, 1 (1991).

J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. 2003. Stack-based typed assembly language. *J. Funct. Program.* 13, 5 (2003), 957–959. https://doi.org/10.1017/S0956796802004446

Guillaume Munch-Maccagnoni. 2014. Models of a Non-associative Composition. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). 396–410.

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *IEEE Symposium on Logic in Computer Science (LICS), Reykjavik, Iceland*.

Pierre-Marie Pédrot and Nicolas Tabareau. 2019. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.* 4, POPL, Article 58 (dec 2019), 28 pages. https://doi.org/10.1145/3371126

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, 71–84. https://doi.org/10.1145/158511.158524

Gordon D. Plotkin and John Power. 2001. Semantics for Algebraic Operations. In *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 45)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference on - ACM '72*, Vol. 2. ACM Press, Boston, Massachusetts, United States, 717–740. https://doi.org/10.1145/800194.805852

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

Hayo Thielecke. 2001. Comparing Control Constructs by Double-barrelled CPS Transforms. *Electronic Notes in Theoretical Computer Science* 45 (2001), 433–447. https://doi.org/10.1016/S1571-0661(04)80974-5 MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, 61–78. https://doi.org/10.1145/91556.91592

Mitchell Wand. 1980. Continuation-Based Program Transformation Strategies. *J. ACM* 27, 1 (1980), 164–180. https://doi.org/10.1145/322169.322183

```
def read_and_stack : Thk (Thk (Int →* OS) → OS)
      ≜ { fix loop. λ s.
          ! read_line_as_int { λ i?. match i?
          | inj₀(_) ⇒ ! s .done
          | inj₁(i) ⇒ ! loop { ! s .more i } } }
def sum_and_print : Thk (Int → Int →* OS)
      ≜ { fix loop. λ sum. comatch
           | .done  ⇒ ! write_int sum { ! halt }
           | .more i  ⇒  do j ← ! add sum i ; ! loop j }
def main ≜ ! read_and_stack { ! sum_and_print 0 }
```

Fig. 15. Zydeco Examples read_and_stack and sum_and_print

A ZYDECO EXAMPLES

A.1 Read Integers, Stack Them Up, Sum Them, and Print the Result

The motivation of this pair of example is to show how to construct and consume a stack of type `Int` $\to^*$ `OS`. The following Zydeco program in Figure 15 reads integers from standard input, pushes them to the stack one by one, sums them while popping, and prints the result before halting. The function `read_and_stack` reads integers from standard input and stacks them up, which **constructs** the `Int` $\to^*$ `OS` typed stack, or conversely, **destructs** a computation of the same type. The function `sum_and_print` sums the integers and prints the result, consuming the stack, yet producing a computation of type `Int` $\to^*$ `OS`. The main function `main` invokes both functions and instantiates the sum with 0.

A.2 Revisiting `sum_and_print`

As an example of a non-trivial use of monadic blocks, let's consider what would happen if we wanted to update the `sum_and_print` function to use an overflow-safe version of add, called `add_of` whose result is not a `Ret Int` but an `Exn String Int`. Without monadic blocks it's unclear how to change the code: before we were using the built-in do notation with a complex computation type. But with monadic blocks we can re-use the exact same implementation of the loop as long as we provide an *algebra* structure for the base type `OS`. The full code is given in Figure 16, we've achieved the goal without changing any original code by wrapping it inside a monadic block and provide the implementation of the overloaded types and functions through funciton arguments. In this example we use a large number of lambdas to "import" values from outside the monadic block into the monadic world. The most important of these is the implementation of `alg_exn_os`, which gives an `OS` algebra of the exception monad by using a runtime panic.

B MONAD IMPLEMENTATIONS

We demonstrate an example of the free monad in Zydeco, defining the operations as user interface and handlers as implementations, and show how to use use them in an example program.

```
def sum_and_print : Thk (Int → Int →* OS)
        ≜ { 𝓜(
              Λ Int : VTy.
              Λ OS : CTy.
              λ add : Thk (Int → Int → Ret Int).
              λ write_int : Thk (Int → Thk OS → OS).
              λ halt : Thk OS.
              fix loop. λ sum. comatch
              | .done  ⇒ ! write_int sum { ! halt }
              | .more i  ⇒  do j ← ! add sum i ; ! loop j
            ) (Exn String) { ! mexn String }
             Int {comatch |} OS  alg_exn_os
             add_of  write_int  halt }
def alg_exn_os  :  Thk (Algebra (Exn String) OS)
      ≜  { Λ A. λ m k.
             do a? ← ! m ; match a?
               | inj₀(s) ⇒ ! panic s
               | inj₁(a) ⇒ ! k a }
def add_of  :  Thk (Int → Int → (Exn String Int))
def panic  :  Thk (String → OS)
```

Fig. 16. Polymorphic sum_and_print via the Monadic Block

```
def KPrint  ≜  λ R : CTy. Thk (String → Kont R Unit)
 def KFlip  ≜  λ R : CTy. Thk (Unit → Kont R Bool)
def PrintFlip  ≜  λ A. ∀ R : CTy. KPrint R → KFlip R → Kont R A

    def mfree  ≜  { comatch
                     | .return  ⇒  Λ A. λ a.
                            Λ R. λ print flip. λ k. ! k a
                     | .bind  ⇒  Λ A A′. λ m f.
                            Λ R. λ print flip.
                               λ k′. ! m R print flip
                                  { λ a. ! f a R print flip k′ } }
```

Fig. 17. An Example of The Free Monad in Zydeco

```
    def print ≜ { $\lambda\, s.\, \Lambda\, R.\, \lambda\, print\ flip.\ !\ print\ s$ }
     def flip ≜ { $\lambda\, \_.\, \Lambda\, R.\, \lambda\, print\ flip.\ !\ flip\ ()$ }
 def print_os ≜ { $\lambda\, s.\, \lambda\, k.$ ! write_line $s$ { ! $k$ () } }
  def flip_os ≜ { $\lambda\, \_.\, \lambda\, k.$ ! random_int {
                    $\lambda\, i.$ do $j \leftarrow$ ! mod $i$ 2 ; do $b \leftarrow$ ! int_eq $j$ 0 ; ! $k$ $b$} }
```

Fig. 18. Operations and Handlers for a Free Monad

```
def main ≜ { let $m$ = {
              ! mfree .bind Bool Unit { ! flip ()} { $\lambda\, b.$
              ! if (PrintFlip Unit) $b$ { ! print "+"} { ! print "-"}
              }} in ! $m$ OS print_os flip_os {$\lambda\, \_.$ ! halt }}
```

Fig. 19. An Example Program Using The Free Monad in Zydeco

C MONAD LAWS

We provide a counter-example to the right unit monad law for the defunctionalized continuation monad in Figure 20.

```
def count_kont ≜ { fix count_kont.
    $\Lambda\ (E : \mathsf{VTy})\ (A : \mathsf{VTy}).\ \lambda\ (i : \mathsf{Int})\ (fi : \mathsf{Thk}\ (\mathsf{Int} \to \mathsf{ExnDe}\ E\ A))$. comatch
        | .try  ⇒ $\Lambda\, E'.\, \lambda\, ke.$ ! count_kont $E'$ $A$ $i$ { $\lambda\, i.$ ! $fi$ $i$ .try $E'$ $ke$ }
        | .kont ⇒ $\Lambda\, A'.\, \lambda\, ka.$ do $i'$ ← ! add $i$ 1;
            ! count_kont $E$ $A'$ $i'$ { $\lambda\, i.$ ! $fi$ $i$ .kont $A'$ $ka$ }
        | .done ⇒ ! $fi$ $i$ .done }

  def ExnCounter ≜ ExnDe Unit Int
        def bench ≜ { $\lambda$ ($impl$ : Thk (Thk ExnCounter → ExnCounter)).
                      do $i?$ ← ! impl {
                        ! count_kont Unit Int 0
                        { ! mexnde Unit .return Int }
                      } .done;
                      match $i?$ | $\mathsf{inj}_0(e)$ ⇒ ret (−1) | $\mathsf{inj}_1(i)$ ⇒ ret $i$ }
```

Fig. 20. Counter example counter-example

We verify the monad laws on the exception monads satisfying the canonicity condition. The right unit law is satisfied without the extra conditions:

$$\begin{aligned}
& !\,\mathtt{bind}\,A\,A'\,\{!\,\mathtt{return}\,A\,a\}\,f\\
\equiv\ & !\,\mathtt{return}\,A\,a\,.\mathtt{kont}\,A'\,f\\
\equiv\ & !\,f\,a
\end{aligned}$$

And the left unit law can be verified if we apply the canonicity condition on $!\,m$.

$$\begin{aligned}
& !\,\mathtt{bind}\,A\,A\,m\,\{\,!\,\mathtt{return}\,A\}\\
\equiv\ & !\,m\,.\mathtt{kont}\,A\,\{\,!\,\mathtt{return}\,A\}\\
\equiv\ & \mathsf{do}\,x \leftarrow !\,m\,;\\
& \mathsf{match}\,x\\
& \mid \mathsf{inj}_0(e) \rightarrow !\,\mathtt{fail}\,e\\
& \mid \mathsf{inj}_1(a) \rightarrow !\,\mathtt{return}\,A\,a\\
& .\mathtt{kont}\,A\,\{\,!\,\mathtt{return}\,A\}\\
\equiv\ & \mathsf{do}\,x \leftarrow !\,m\,;\\
& \mathsf{match}\,x\\
& \mid \mathsf{inj}_0(e) \rightarrow !\,\mathtt{fail}\,e\,.\mathtt{kont}\,A\,\{\,!\,\mathtt{return}\,A\}\\
& \mid \mathsf{inj}_1(a) \rightarrow !\,\mathtt{return}\,A\,a\,.\mathtt{kont}\,A\,\{\,!\,\mathtt{return}\,A\}\\
\equiv\ & \mathsf{do}\,x \leftarrow !\,m\,;\\
& \mathsf{match}\,x\\
& \mid \mathsf{inj}_0(e) \rightarrow !\,\mathtt{fail}\,e\\
& \mid \mathsf{inj}_1(a) \rightarrow !\,\mathtt{return}\,A\,a\\
\equiv\ & !\,m
\end{aligned}$$

As for the associativity law, we observe

$$\begin{aligned}
!\,\mathtt{bind}\,A'\,A''\,\{\,!\,\mathtt{bind}\,A\,A'\,m\,f\}\,g\ &\equiv !\,m\,.\mathtt{kont}\,A'\,f\,.\mathtt{kont}\,A''\,g\\
!\,\mathtt{bind}\,A\,A'\,m\,\{\lambda\,x.\,!\,\mathtt{bind}\,A'\,A''\,\{\,!\,f\,x\}\,g\}\ &\equiv !\,m\,.\mathtt{kont}\,A''\,\{\lambda\,x.\,!\,f\,x\,.\mathtt{kont}\,A''\,g\}
\end{aligned}$$

By applying the canonicity condition on $!\,m$ and using the same reasoning as in the left unit law, we have for the left side

$$\begin{aligned}
& !\,m\,.\mathtt{kont}\,A'\,f\,.\mathtt{kont}\,A''\,g\\
\equiv\ & \mathsf{do}\,x \leftarrow !\,m\,;\\
& \mathsf{match}\,x\\
& \mid \mathsf{inj}_0(e) \rightarrow !\,\mathtt{fail}\,e\\
& \mid \mathsf{inj}_1(a) \rightarrow !\,\mathtt{return}\,A\,a\,.\mathtt{kont}\,A'\,f\,.\mathtt{kont}\,A''\,g\\
\equiv\ & \mathsf{do}\,x \leftarrow !\,m\,;\\
& \mathsf{match}\,x\\
& \mid \mathsf{inj}_0(e) \rightarrow !\,\mathtt{fail}\,e\\
& \mid \mathsf{inj}_1(a) \rightarrow !\,f\,a\,.\mathtt{kont}\,A''\,g\\
& \mathit{end}
\end{aligned}$$

$$
\begin{aligned}
\mathsf{Sig} &: \forall \{K\} \{\Delta\} \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash K) \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash \mathsf{CTy}) \\
\mathsf{Sig}_{\mathsf{VTy}}(A) &:= \mathtt{Top} \\
\mathsf{Sig}_{\mathsf{CTy}}(B) &:= \mathtt{Algebra}\ T\ B \\
\mathsf{Sig}_{K_0 \to K}(S) &:= \forall X : K_0.\ \mathsf{Thk}\ (\mathsf{Sig}_{K_0}(X)) \to \mathsf{Sig}_K(S\ X)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Sig} &: (\Delta : TEnv) \to \Delta \vdash VEnv \\
\mathsf{Sig}(\cdot) &:= \cdot \\
\mathsf{Sig}(\Delta, X : K) &:= \mathsf{Sig}(\Delta), str_X : \mathsf{Thk}\ (\mathsf{Sig}_K(X))
\end{aligned}
$$

Fig. 21. Signature Translation and Type Environment Signature Translation

and the right side

$$
\begin{aligned}
&\ !\ m\ \mathtt{.kont}\ A''\ \{\lambda\, x.\ !\ f\ x\ \mathtt{.kont}\ A''\ g\} \\
\equiv\ & \mathsf{do}\ x \leftarrow\ !\ m\ ; \\
& \mathsf{match}\ x \\
& \mid \mathsf{inj}_0(e) \to\ !\ \mathtt{fail}\ e \\
& \mid \mathsf{inj}_1(a) \to\ !\ \mathtt{return}\ A\ a\ \mathtt{.kont}\ A''\ \{\lambda\, x.\ !\ f\ x\ \mathtt{.kont}\ A''\ g\} \\
\equiv\ & \mathsf{do}\ x \leftarrow\ !\ m\ ; \\
& \mathsf{match}\ x \\
& \mid \mathsf{inj}_0(e) \to\ !\ \mathtt{fail}\ e \\
& \mid \mathsf{inj}_1(a) \to\ !\ f\ a\ \mathtt{.kont}\ A''\ g \\
& \mathit{end}
\end{aligned}
$$

Finally, the linearity law trivially holds because M is syntactically restricted to be used for only once in the computation. The only surrounding computation of M is a do-binding, which preserves linearity. Therefore, we conclude that all four monad laws are satisfied with the canonicity property.

D ALGEBRA TRANSLATION

We give the full algebra translation composed of seven translations.

(1) Signature Translation: see Figure 21.
(2) Type Environment Signature Translation: see Figure 21.
(3) Carrier Translation: see Figure 22.
(4) Value Environment Translation: see Figure 22.
(5) Structure Translation: see Figure 23.
(6) Term Translations: see Figure 24 and Figure 25.
(7) Monadic Block Translation: see Figure 26.

$$\begin{aligned}
\lfloor \cdot \rfloor &: \forall \{K\}\, \{\Delta\} \to (\Delta \vdash K) \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \vdash K) \\
\lfloor \mathsf{Thk} \rfloor &:= \mathsf{Thk} \\
\lfloor \mathtt{Unit} \rfloor &:= \mathtt{Unit} \\
\lfloor (\times) \rfloor &:= (\times) \\
\lfloor (+) \rfloor &:= (+) \\
\lfloor \exists\, X : K.\, A \rfloor &:= \exists\, X : K.\, \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \times \lfloor A \rfloor \\
\lfloor \mathtt{Ret} \rfloor &:= T \\
\lfloor (\to) \rfloor &:= (\to) \\
\lfloor \mathsf{Top} \rfloor &:= \mathsf{Top} \\
\lfloor (\&) \rfloor &:= (\&) \\
\lfloor \forall\, X : K.\, B \rfloor &:= \forall\, X : K.\, \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \to \lfloor B \rfloor \\
\lfloor X \rfloor &:= X \\
\lfloor \lambda\, X : K.\, S \rfloor &:= \lambda\, X : K.\, \lfloor S \rfloor \\
\lfloor S\, S_0 \rfloor &:= \lfloor S \rfloor\, \lfloor S_0 \rfloor \\
\lfloor \nu\, Y : K.\, B \rfloor &:= \nu\, Y : K.\, \lfloor B \rfloor
\end{aligned}$$

$$\begin{aligned}
\lfloor \cdot \rfloor &: \mathit{VEnv} \to \mathit{VEnv} \\
\lfloor \cdot \rfloor &:= \cdot \\
\lfloor \Gamma, x : A \rfloor &:= \lfloor \Gamma \rfloor, x : \lfloor A \rfloor
\end{aligned}$$

Fig. 22. Carrier Translation and Value Environment Translation

$$
\begin{array}{rcl}
\mathit{Str} & : & \forall\, \{K\}\; \{\Delta\} \to (\Delta \vdash S : K) \\
& & \to (T : \mathsf{VTy} \to \mathsf{CTy}, \Delta;\; \mathit{mo} : \mathsf{Thk}\ (\mathtt{Monad}\ T), \mathrm{Sig}(\Delta) \vdash \mathrm{Sig}_K(\lfloor S \rfloor)) \\
\mathrm{Str}(\mathsf{Thk}) & := & \Lambda\, X.\, \lambda\, \_.\ \mathsf{comatch}\ | \\
\mathrm{Str}(\mathtt{Unit}) & := & \mathsf{comatch}\ | \\
\mathrm{Str}((\times)) & := & \Lambda\, X.\, \lambda\, \_.\, \Lambda\, Y.\, \lambda\, \_.\ \mathsf{comatch}\ | \\
\mathrm{Str}((+)) & := & \Lambda\, X.\, \lambda\, \_.\, \Lambda\, Y.\, \lambda\, \_.\ \mathsf{comatch}\ | \\
\mathrm{Str}(\exists\, X : K.\, A) & := & \mathsf{comatch}\ | \\
\mathrm{Str}(\mathtt{Ret}) & := & \Lambda\, X.\, \lambda\, \_.\, \Lambda\, Z.\, \lambda\, \mathit{mz}\ (f : \mathsf{Thk}\ (Z \to T\ X)). \\
& & \quad !\, \mathit{mo}\ \mathtt{.bind}\ Z\ X\ \mathit{mz}\ f \\
\mathrm{Str}((\to)) & := & \Lambda\, X.\, \lambda\, \_.\, \Lambda\, Y.\, \lambda\, \mathit{alg}_Y.\, \Lambda\, Z.\, \lambda\, \mathit{mz}\ (f : \mathsf{Thk}\ (Z \to \lfloor X \rfloor \to \lfloor Y \rfloor)). \\
& & \quad \lambda\ (x : \lfloor X \rfloor).\ !\, \mathit{alg}_Y\ Z\ \mathit{mz}\ \{\lambda\, z.\ !\, f\ z\ x\} \\
\mathrm{Str}((\&)) & := & \Lambda\, X.\, \lambda\, \mathit{alg}_X.\, \Lambda\, Y.\, \lambda\, \mathit{alg}_Y.\, \Lambda\, Z.\, \lambda\, \mathit{mz}\ (f : \mathsf{Thk}\ (Z \to \lfloor X \rfloor \,\&\, \lfloor Y \rfloor)). \\
& & \quad \mathsf{comatch} \\
& & \quad\ |\ .0 \Rightarrow\ !\, \mathit{alg}_X\ Z\ \mathit{mz}\ \{\lambda\, z.\ !\, f\ z\ .0\} \\
& & \quad\ |\ .1 \Rightarrow\ !\, \mathit{alg}_Y\ Z\ \mathit{mz}\ \{\lambda\, z.\ !\, f\ z\ .1\} \\
\mathrm{Str}(\forall\, X : K.\, B) & := & \Lambda\, Z.\, \lambda\, \mathit{mz}\ (f : \mathsf{Thk}\ (Z \to \forall\, X : K.\ \mathsf{Thk}\ (\mathrm{Sig}_K(X)) \to \lfloor B \rfloor)). \\
& & \quad \Lambda\, X : K.\, \lambda\ (\mathit{str}_X : \mathsf{Thk}\ (\mathrm{Sig}_K(X))).\ \mathrm{Str}(B)\ Z\ \mathit{mz}\ \{\lambda\, z.\ !\, f\ z\ X\ \mathit{str}_X\} \\
\mathrm{Str}(\nu\, Y : K.\, B) & := & \mathsf{fix}\ (\mathit{str} : \mathsf{Thk}\ (\mathtt{Algebra}\ T\ (\nu\, Y : K.\ \lfloor B \rfloor))). \\
& & \quad \Lambda\, Z.\, \lambda\, \mathit{mz}\ (f : \mathsf{Thk}\ Z \to \nu\, Y : K.\ \lfloor B \rfloor). \\
& & \qquad \mathsf{roll}(\mathrm{Str}(B)[(\nu\, Y : K.\ \lfloor B \rfloor)/Y][\mathit{str}/\mathit{str}_Y]\ Z\ \mathit{mz}\ \{\lambda\, z.\ \mathsf{unroll}(\ !\, f\ z\ )\}) \\
\mathrm{Str}(X) & := & !\, \mathit{str}_X \\
\mathrm{Str}(\lambda\, X : K.\, S) & := & \Lambda\, X.\, \lambda\, \mathit{str}_X.\ \mathrm{Str}(S) \\
\mathrm{Str}(S\ S_0) & := & \mathrm{Str}(S)\ \lfloor S_0 \rfloor\ \{\mathrm{Str}(S_0)\}
\end{array}
$$

Fig. 23. Structure Translation

$$
\begin{array}{rcl}
\lfloor \cdot \rfloor & : & \forall\, \{A\}\; \{\Delta\}\; \{\Gamma\} \\
& & \to (\ \Delta; \Gamma \vdash A\ ) \\
& & \to (\ T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \\
& & \qquad ;\ \mathit{mo} : \mathsf{Thk}\ (\mathtt{Monad}\ T), \mathrm{Sig}(\Delta), \lfloor \Gamma \rfloor \vdash \lfloor A \rfloor\ ) \\
\lfloor x \rfloor & := & x \\
\lfloor \{M\} \rfloor & := & \{\lfloor M \rfloor\} \\
\lfloor () \rfloor & := & () \\
\lfloor (V_0, V_1) \rfloor & := & (\lfloor V_0 \rfloor, \lfloor V_1 \rfloor) \\
\lfloor \mathsf{inj}_i(V) \rfloor & := & \mathsf{inj}_i(\lfloor V \rfloor) \\
\lfloor (S, V) \rfloor & := & (\lfloor S \rfloor, (\{\mathrm{Str}(S)\}, \lfloor V \rfloor))
\end{array}
$$

Fig. 24. Term Translation (Value)

$$
\begin{aligned}
\lfloor \cdot \rfloor \ &: \ \forall\, \{B\}\ \{\Delta\}\ \{\Gamma\} \\
&\quad \to (\ \Delta;\Gamma \vdash B\ ) \\
&\quad \to (\ T : \mathsf{VTy} \to \mathsf{CTy}, \Delta \\
&\qquad ;\ mo : \mathsf{Thk}\ (\mathtt{Monad}\ T), \mathrm{Sig}(\Delta), \lfloor \Gamma \rfloor \vdash \lfloor B \rfloor\ ) \\
\lfloor\, !\, V \rfloor &:= !\, \lfloor V \rfloor \\
\lfloor \mathsf{let}\ (x_0, x_1) = V\ \mathsf{in}\ M \rfloor &:= \mathsf{let}\ (x_0, x_1) = \lfloor V \rfloor\ \mathsf{in}\ \lfloor M \rfloor \\
\lfloor \mathsf{match}\ V \mid \mathsf{inj}_0(x_0) \Rightarrow M_0 \mid \mathsf{inj}_1(x_1) \Rightarrow M_1 \rfloor &:= \mathsf{match}\ \lfloor V \rfloor \\
&\quad \mid \mathsf{inj}_0(x_0) \Rightarrow \lfloor M_0 \rfloor \\
&\quad \mid \mathsf{inj}_1(x_1) \Rightarrow \lfloor M_1 \rfloor \\
\lfloor \mathsf{let}\ (X, x) = V\ \mathsf{in}\ M \rfloor &:= \mathsf{let}\ (X, p) = \lfloor V \rfloor\ \mathsf{in}\ \mathsf{let}\ (str_X, x) = p\ \mathsf{in}\ \lfloor M \rfloor \\
\lfloor \mathsf{ret}\ V \rfloor &:= !\, mo\ \mathtt{.return}\ \lfloor A \rfloor\ \lfloor V \rfloor \\
&\qquad \text{where } \Delta;\Gamma \vdash V : A \\
\lfloor \mathsf{do}\ x \leftarrow M_0\ ; M \rfloor &:= \mathrm{Str}(B)\ \lfloor A \rfloor\ \{\lfloor M_0 \rfloor\}\ \{\lambda\, x.\ \lfloor M \rfloor\} \\
&\qquad \text{where } \Delta;\Gamma \vdash M_0 : \mathtt{Ret}\ A \text{ and } \Delta;\Gamma \vdash M : B \\
\lfloor \lambda\, x : A.\ M \rfloor &:= \lambda\, x : \lfloor A \rfloor.\ \lfloor M \rfloor \\
\lfloor M\ V \rfloor &:= \lfloor M \rfloor\ \lfloor V \rfloor \\
\lfloor \mathsf{comatch} \mid \rfloor &:= \mathsf{comatch} \mid \\
\lfloor \mathsf{comatch} \mid .0 \Rightarrow M_0 \mid .1 \Rightarrow M_1 \rfloor &:= \mathsf{comatch} \\
&\quad \mid .0 \Rightarrow \lfloor M_0 \rfloor \\
&\quad \mid .1 \Rightarrow \lfloor M_1 \rfloor \\
\lfloor M\, .i \rfloor &:= \lfloor M \rfloor\, .i \\
\lfloor \Lambda\, X : K.\ M \rfloor &:= \Lambda\, X : K.\ \lambda\, str_X.\ \lfloor M \rfloor \\
\lfloor M\ S \rfloor &:= \lfloor M \rfloor\ \lfloor S \rfloor\ \{\mathrm{Str}(S)\} \\
\lfloor \mathsf{roll}(M) \rfloor &:= \mathsf{roll}(\lfloor M \rfloor) \\
\lfloor \mathsf{unroll}(M) \rfloor &:= \mathsf{unroll}(\lfloor M \rfloor) \\
\lfloor \mathsf{fix}\ x.\ M \rfloor &:= \mathsf{fix}\ x.\ \lfloor M \rfloor
\end{aligned}
$$

Fig. 25. Term Translation (Computation)

$$
\begin{aligned}
[\![\cdot]\!] &: \forall \{A\}\, \{\Delta\}\, \{\Gamma\} \to (\Delta; \Gamma \vdash A) \to (\Delta; \Gamma \vdash A) \\
[\![x]\!] &:= x \\
[\![\{M\}]\!] &:= \{[\![M]\!]\} \\
[\![()]\!] &:= () \\
[\![(V_0, V_1)]\!] &:= ([\![V_0]\!], [\![V_1]\!]) \\
[\![\mathsf{inj}_i(V)]\!] &:= \mathsf{inj}_i([\![V]\!]) \\
[\![(S, V)]\!] &:= (S, [\![V]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![\cdot]\!] &: \forall \{B\}\, \{\Delta\}\, \{\Gamma\} \to (\Delta; \Gamma \vdash B) \to (\Delta; \Gamma \vdash B) \\
[\![\mathcal{M}(M)]\!] &:= \Lambda\, (T : \mathsf{VTy} \to \mathsf{CTy}). \\
&\qquad \lambda\, (mo : \mathsf{Thk}\ (\mathsf{Monad}\ T)).\ \lfloor\, [\![M]\!]\, \rfloor \\
[\![!\, V]\!] &:= !\, [\![V]\!] \\
[\![\mathsf{let}\ (x_0, x_1) = V\ \mathsf{in}\ M]\!] &:= \mathsf{let}\ (x_0, x_1) = [\![V]\!]\ \mathsf{in}\ [\![M]\!] \\
[\![\mathsf{match}\ V \mid \mathsf{inj}_0(x_0) \Rightarrow M_0 \mid \mathsf{inj}_1(x_1) \Rightarrow M_1]\!] &:= \mathsf{match}\ [\![V]\!] \\
&\quad \mid \mathsf{inj}_0(x_0) \Rightarrow [\![M_0]\!] \\
&\quad \mid \mathsf{inj}_1(x_1) \Rightarrow [\![M_1]\!] \\
[\![\mathsf{let}\ (X, x) = V\ \mathsf{in}\ M]\!] &:= \mathsf{let}\ (X, x) = [\![V]\!]\ \mathsf{in}\ [\![M]\!] \\
[\![\mathsf{ret}\ V]\!] &:= \mathsf{ret}\ [\![V]\!] \\
[\![\mathsf{do}\ x \leftarrow M_0\ ;\ M]\!] &:= \mathsf{do}\ x \leftarrow [\![M_0]\!]\ ;\ [\![M]\!] \\
[\![\lambda\, x.\, M]\!] &:= \lambda\, x.\, [\![M]\!] \\
[\![M\ V]\!] &:= [\![M]\!]\, [\![V]\!] \\
[\![\mathsf{comatch} \mid]\!] &:= \mathsf{comatch} \mid \\
[\![\mathsf{comatch} \mid .0 \Rightarrow M_0 \mid .1 \Rightarrow M_1]\!] &:= \mathsf{comatch} \\
&\quad \mid .0 \Rightarrow [\![M_0]\!] \\
&\quad \mid .1 \Rightarrow [\![M_1]\!] \\
[\![M\ .i]\!] &:= [\![M]\!]\ .i \\
[\![\Lambda\, X : K.\, M]\!] &:= \Lambda\, X.\, [\![M]\!] \\
[\![M\ S]\!] &:= [\![M]\!]\ S \\
[\![\mathsf{roll}(M)]\!] &:= \mathsf{roll}([\![M]\!]) \\
[\![\mathsf{unroll}(M)]\!] &:= \mathsf{unroll}([\![M]\!]) \\
[\![\mathsf{fix}\ x.\, M]\!] &:= \mathsf{fix}\ x.\, [\![M]\!]
\end{aligned}
$$

Fig. 26. Monadic Block Translation

E ALGEBRAS GENERATED BY THE ALGEBRA TRANSLATION IS LAWFUL

In this section we show that the algebra structures generated by the algebra translation obey the algebra laws. The translation in concern is the term translation, defined in Figure 24 and Figure 25. In order to prove all algebras we construct are lawful, we need to generalize the inductive hypothesis and prove all structures satisfy some "laws":

(1) For computation types, the structure is an algebra and should obey the algebra laws;
(2) For value types, the structure is $\mathtt{Top}$ and so no laws need to be verified.
(3) For type constructors $K \to K'$, we need to verify that when instantiated with a lawful structure for K, we produce a lawful structure for K'.

We first talk about value types and type constructors. The value types trivially obey the laws since there is no law to obey. For type constructors, since the lawful structure is passed in through a function parameter, we only care about the cases when a computation type is returned. Therefore, we only need to verify that the laws hold for computation types.

Lemma E.1. *The algebra generated by the algebra translation is lawful under CBPV $\beta\eta$ equality. Namely, the following equational principles hold for all type environment Δ, monad type constructor $\Delta \vdash T : VTy \to CTy$, and computation type $\Delta \vdash B : CTy$:*

(1) ***left unit law:***

$$Str(B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f \equiv\ !\ f\ a$$

for any $a : A$ and $f : \mathtt{Thk}\ (A \to Sig_{CTy}(B))$,

(2) ***associativity law:***

$$Str(B)\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g \equiv Str(B)\ A\ m\ \{\lambda\, x.\ Str(B)\ A'\ \{\ !\ f\ x\}\ g\}$$

for any $f : \mathtt{Thk}\ (A \to T\ A')$, $g : \mathtt{Thk}\ (A' \to Sig_{CTy}(B))$, and $m : \mathtt{Thk}\ (T\ A)$,

(3) ***linearity law:***

$$do\ m \leftarrow\ !\ tm\ ;\ Str(B)\ A\ m \equiv Str(B)\ A\ \{do\ m \leftarrow\ !\ tm\ ;\ !\ m\}$$

for any $tm : \mathtt{Thk}\ (\mathtt{Ret}\ (\mathtt{Thk}\ (T\ A)))$.

Proof. By induction on the cases of structure translation.
We start from the left unit law:

(1) **Case** $\mathrm{Str}(\mathtt{Ret}\ A_0)$ for $A_0 : \mathrm{VTy}$: by applying the left unit law of monad T

$$\begin{aligned}&\mathrm{Str}(\mathtt{Ret}\ A_0)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\ \equiv\ &!\ \mathtt{bind}\ A\ A_0\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\ \equiv\ &!\ f\ a\end{aligned}$$

(2) **Case** $\mathrm{Str}(A_0 \to B)$ for $A_0 : \mathrm{VTy}$ and $B : \mathrm{CTy}$: by induction hypothesis on B

$$\begin{aligned}&\mathrm{Str}(A_0 \to B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\ \equiv\ &\lambda\, a_0.\ \mathrm{Str}(A_0 \to B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\ a_0\\ \equiv\ &\lambda\, a_0.\ \mathrm{Str}(B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ \{\lambda\, x.\ !\ f\ x\ a_0\}\\ \equiv\ &\lambda\, a_0.\ (\lambda\, x.\ !\ f\ x\ a_0)\ a\\ \equiv\ &\lambda\, a_0.\ !\ f\ a\ a_0\\ \equiv\ &!\ f\ a\end{aligned}$$

(3) **Case** $\mathrm{Str}(B_0 \,\&\, B_1)$ for $B_0 : \mathsf{CTy}$ and $B_1 : \mathsf{CTy}$: by induction hypothesis on B_0 and B_1

$$\begin{aligned}
&\mathrm{Str}(B_0 \,\&\, B_1)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &\mathsf{comatch}\\
&\mid .0 \Rightarrow \mathrm{Str}(B_0)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ \{\lambda\, x.\,!\ f\ x\ .0\}\\
&\mid .1 \Rightarrow \mathrm{Str}(B_1)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ \{\lambda\, x.\,!\ f\ x\ .1\}\\
\equiv\ &\mathsf{comatch}\\
&\mid .0 \Rightarrow\ !\ f\ a\ .0\\
&\mid .1 \Rightarrow\ !\ f\ a\ .1\\
\equiv\ &\ !\ f\ a
\end{aligned}$$

(4) **Case** $\mathrm{Str}(\forall\, X : K.\, B)$ for all K and $B : \mathsf{CTy}$: by induction hypothesis on B

$$\begin{aligned}
&\mathrm{Str}(\forall\, X : K.\, B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &\Lambda\, X.\,\lambda\, str_X.\ \mathrm{Str}(B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ \{\lambda\, x.\,!\ f\ x\ X\ str_X\}\\
\equiv\ &\Lambda\, X.\,\lambda\, str_X.\,!\ f\ a\ X\ str_X\\
\equiv\ &\ !\ f\ a
\end{aligned}$$

(5) **Case** $\mathrm{Str}(\nu\ Y : \mathsf{CTy}.\, B)$ for $B : \mathsf{CTy}$:
let str_ν be $\{\mathrm{Str}(\nu\ Y : \mathsf{CTy}.\, B)\}$
by induction hypothesis on B

$$\begin{aligned}
&\mathrm{Str}(\nu\ Y : \mathsf{CTy}.\, B)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &\mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ \{\lambda\, x.\ \mathsf{unroll}(\ !\ f\ x\ )\})\\
\equiv\ &\mathsf{roll}((\lambda\, x.\ \mathsf{unroll}(\ !\ f\ x\ ))\ a)\\
\equiv\ &\mathsf{roll}(\mathsf{unroll}(\ !\ f\ a\ ))\\
\equiv\ &\ !\ f\ a
\end{aligned}$$

(6) **Case** $\mathrm{Str}(X)$ for $X : \mathsf{CTy}$: since the structure is from the value environment, it's lawful by the induction hypothesis.

(7) **Case** $\mathrm{Str}((\lambda\, X : K.\, B)\ S)$ for $S : K$ and $B : \mathsf{CTy}$: by induction hypothesis on S and B

$$\begin{aligned}
&\mathrm{Str}((\lambda\, X : K.\, B)\ S)\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &(\Lambda\, X.\,\lambda\, str_X.\ \mathrm{Str}(B))\ \lfloor\, S\,\rfloor\ \{\mathrm{Str}(S)\}\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &\mathrm{Str}(B)[\lfloor\, S\,\rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A\ \{\ !\ \mathtt{return}\ A\ a\}\ f\\
\equiv\ &\ !\ f\ a
\end{aligned}$$

Then we move on to the associativity law:

(1) **Case** $\mathrm{Str}(\mathtt{Ret}\ A_0)$ for all $A_0 : \mathsf{VTy}$: by applying the associativity law of monad T

$$\begin{aligned}
&\mathrm{Str}(\mathtt{Ret}\ A_0)\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g\\
\equiv\ &\ !\ \mathtt{bind}\ A'\ A_0\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g\\
\equiv\ &\ !\ \mathtt{bind}\ A\ A_0\ m\ \{\lambda\, x.\,!\ \mathtt{bind}\ A'\ A_0\ \{\ !\ f\ x\}\ g\}\\
\equiv\ &\ !\ \mathtt{bind}\ A\ A_0\ m\ \{\lambda\, x.\ \mathrm{Str}(\mathtt{Ret}\ A_0)\ A'\ \{\ !\ f\ x\}\ g\}\\
\equiv\ &\mathrm{Str}(\mathtt{Ret}\ A_0)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(\mathtt{Ret}\ A_0)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

(2) **Case** $\mathrm{Str}(A_0 \to B)$ for $A_0 : \mathsf{VTy}$ and $B : \mathsf{CTy}$: by induction hypothesis on B

$$\begin{aligned}
&\mathrm{Str}(A_0 \to B)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ \{\lambda\, y.\ !\ g\ y\ z\} \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B)\ A'\ \{\ !\ f\ x\}\ \{\lambda\, y.\ !\ g\ y\ z\}\} \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ (\lambda\, w.\ \mathrm{Str}(B)\ A'\ \{\ !\ f\ x\}\ \{\lambda\, y.\ !\ g\ y\ w\})\ z\} \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(A_0 \to B)\ A'\ \{\ !\ f\ x\}\ g\ z\} \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, w.\ \mathrm{Str}(A_0 \to B)\ A'\ \{\ !\ f\ w\}\ g\ z\} \\
\equiv\ &\lambda\, z.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, w.\ (\lambda\, x.\ \mathrm{Str}(A_0 \to B)\ A'\ \{\ !\ f\ x\}\ g)\ w\ z\} \\
\equiv\ &\mathrm{Str}(A_0 \to B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(A_0 \to B)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

(3) **Case** $\mathrm{Str}(B_0 \,\&\, B_1)$ for $B_0 : \mathsf{CTy}$ and $B_1 : \mathsf{CTy}$: by induction hypothesis on B_0 and B_1

$$\begin{aligned}
&\mathrm{Str}(B_0 \,\&\, B_1)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ &\mathsf{comatch} \\
&\ \mid .0 \Rightarrow \mathrm{Str}(B_0)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ \{\lambda\, x.\ !\ g\ x\ .0\} \\
&\ \mid .1 \Rightarrow \mathrm{Str}(B_1)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ \{\lambda\, x.\ !\ g\ x\ .1\} \\
\equiv\ &\mathsf{comatch} \\
&\ \mid .0 \Rightarrow \mathrm{Str}(B_0)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B_0)\ A'\ \{\ !\ f\ x\}\ \{\lambda\, x.\ !\ g\ x\ .0\}\} \\
&\ \mid .1 \Rightarrow \mathrm{Str}(B_1)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B_1)\ A'\ \{\ !\ f\ x\}\ \{\lambda\, x.\ !\ g\ x\ .1\}\} \\
\equiv\ &\mathsf{comatch} \\
&\ \mid .0 \Rightarrow \mathrm{Str}(B_0)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B_0 \,\&\, B_1)\ A'\ \{\ !\ f\ x\}\ g\ .0\} \\
&\ \mid .1 \Rightarrow \mathrm{Str}(B_1)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B_0 \,\&\, B_1)\ A'\ \{\ !\ f\ x\}\ g\ .1\} \\
\equiv\ &\mathrm{Str}(B_0 \,\&\, B_1)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B_0 \,\&\, B_1)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

(4) **Case** $\mathrm{Str}(\forall\, X : K.\ B)$ for all K and $B : \mathsf{CTy}$: by induction hypothesis on B

$$\begin{aligned}
&\mathrm{Str}(\forall\, X : K.\ B)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ &\Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B)\ A'\ \{\ !\ \mathsf{bind}\ A\ A'\ m\ f\}\ \{\lambda\, z.\ !\ g\ z\ X\ str_X\} \\
\equiv\ &\Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B)\ A'\ \{\ !\ f\ x\}\ \{\lambda\, z.\ !\ g\ z\ X\ str_X\}\} \\
\equiv\ &\Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(\forall\, X : K.\ B)\ A'\ \{\ !\ f\ x\}\ g\ X\ str_X\} \\
\equiv\ &\mathrm{Str}(\forall\, X : K.\ B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(\forall\, X : K.\ B)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

(5) **Case** $\mathrm{Str}(\nu\, Y : \mathsf{CTy}.\ B)$ for $B : \mathsf{CTy}$:
let B_ν be $\nu\, Y : \mathsf{CTy}.\ \lfloor\ B\ \rfloor$
and str_ν be $\{\mathrm{Str}(\nu\, Y : \mathsf{CTy}.\ B)\}$

by induction hypothesis on B

$$\begin{aligned}
&\mathrm{Str}(\nu\, Y : \mathsf{CTy}.\, B)\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ & \mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ \{\lambda\, x.\ \mathsf{unroll}(\ !\ g\ x)\}) \\
\equiv\ & \mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ m\ \{\lambda\, x. \\
&\quad \mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A'\ \{\ !\ f\ x\}\ \{\lambda\, y.\ \mathsf{unroll}(\ !\ g\ y)\}\}) \\
\equiv\ & \mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ m\ \{\lambda\, x.\ \mathsf{unroll}(\\
&\quad \mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A'\ \{\ !\ f\ x\}\ \{\lambda\, y.\ \mathsf{unroll}(\ !\ g\ y)\}))\}) \\
\equiv\ & \mathsf{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ m\ \{\lambda\, x.\ \mathsf{unroll}(\mathrm{Str}(\nu\, Y : \mathsf{CTy}.\, B)\ A'\ \{\ !\ f\ x\}\ g)\}) \\
\equiv\ & \mathrm{Str}(\nu\, Y : \mathsf{CTy}.\, B)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(\nu\, Y : \mathsf{CTy}.\, B)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

(6) **Case** $\mathrm{Str}(X)$ for $X : \mathsf{CTy}$: since the structure is from the value environment, it's lawful by the induction hypothesis.

(7) **Case** $\mathrm{Str}((\lambda\, X : K.\, B)\ S)$ for $S : K$ and $B : \mathsf{CTy}$: by induction hypothesis on S and B

$$\begin{aligned}
&\mathrm{Str}((\lambda\, X : K.\, B)\ S)\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ & (\Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B))\ \lfloor\ S\ \rfloor\ \{\mathrm{Str}(S)\}\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ & \mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A'\ \{\ !\ \mathtt{bind}\ A\ A'\ m\ f\}\ g \\
\equiv\ & \mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A\ m\ \{\lambda\, x.\ \mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A'\ \{\ !\ f\ x\}\ g\} \\
\equiv\ & \mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A\ m\ \{\lambda\, x.\ \mathrm{Str}((\lambda\, X : K.\, B)\ S)\ A'\ \{\ !\ f\ x\}\ g\} \\
\equiv\ & \mathrm{Str}((\lambda\, X : K.\, B)\ S)\ A\ m\ \{\lambda\, x.\ \mathrm{Str}((\lambda\, X : K.\, B)\ S)\ A'\ \{\ !\ f\ x\}\ g\}
\end{aligned}$$

Finally, we prove the linear bind law:

(1) **Case** $\mathrm{Str}(\mathtt{Ret}\ A_0)$ for all $A_0 : \mathsf{VTy}$: by applying the linear bind law of monad T

$$\begin{aligned}
&\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(\mathtt{Ret}\ A_0)\ A\ m \\
\equiv\ & \mathsf{do}\ m \leftarrow\ !\ tm\ ;\ !\ \mathtt{bind}\ A\ A_0\ m \\
\equiv\ & \ !\ \mathtt{bind}\ A\ A_0\ \{\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\} \\
\equiv\ & \mathrm{Str}(\mathtt{Ret}\ A_0)\ A\ \{\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

(2) **Case** $\mathrm{Str}(A_0 \to B)$ for $A_0 : \mathsf{VTy}$ and $B : \mathsf{CTy}$: by induction hypothesis on B

$$\begin{aligned}
&\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(A_0 \to B)\ A\ m \\
\equiv\ & \mathsf{do}\ m \leftarrow\ !\ tm\ ;\ \lambda\, f\ a.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ a\} \\
\equiv\ & \lambda\, f\ a.\ \mathsf{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ a\} \\
\equiv\ & \lambda\, f\ a.\ \mathrm{Str}(B)\ A\ \{\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\ \{\lambda\, x.\ !\ f\ x\ a\} \\
\equiv\ & \mathrm{Str}(B)\ A\ \{\mathsf{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

(3) **Case** $\mathrm{Str}(B_0\ \&\ B_1)$ for $B_0 : \mathrm{CTy}$ and $B_1 : \mathrm{CTy}$: by induction hypothesis on B_0 and B_1

$$\begin{aligned}
&\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B_0\ \&\ B_1)\ A\ m\\
\equiv\ &\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \lambda\, f.\ \mathrm{comatch}\\
&\mid .0 \Rightarrow \mathrm{Str}(B_0)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ .0\}\\
&\mid .1 \Rightarrow \mathrm{Str}(B_1)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ .1\}\\
\equiv\ &\lambda\, f.\ \mathrm{comatch}\\
&\mid .0 \Rightarrow \mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B_0)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ .0\}\\
&\mid .1 \Rightarrow \mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B_1)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ .1\}\\
\equiv\ &\lambda\, f.\ \mathrm{comatch}\\
&\mid .0 \Rightarrow \mathrm{Str}(B_0)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\ \{\lambda\, x.\ !\ f\ x\ .0\}\\
&\mid .1 \Rightarrow \mathrm{Str}(B_1)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\ \{\lambda\, x.\ !\ f\ x\ .1\}\\
\equiv\ &\mathrm{Str}(B_0\ \&\ B_1)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

(4) **Case** $\mathrm{Str}(\forall\, X : K.\ B)$ for all K and $B : \mathrm{CTy}$: by induction hypothesis on B

$$\begin{aligned}
&\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(\forall\, X : K.\ B)\ A\ m\\
\equiv\ &\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \lambda\, f.\ \Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ X\ str_X\}\\
\equiv\ &\lambda\, f.\ \Lambda\, X.\ \lambda\, str_X.\ \mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B)\ A\ m\ \{\lambda\, x.\ !\ f\ x\ X\ str_X\}\\
\equiv\ &\lambda\, f.\ \Lambda\, X.\ \lambda\, str_X.\ \mathrm{Str}(B)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\ \{\lambda\, x.\ !\ f\ x\ X\ str_X\}\\
\equiv\ &\mathrm{Str}(B)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

(5) **Case** $\mathrm{Str}(\nu\, Y : \mathrm{CTy}.\ B)$ for $B : \mathrm{CTy}$:
let B_ν be $\nu\, Y : \mathrm{CTy}.\ \lfloor\ B\ \rfloor$
and str_ν be $\{\mathrm{Str}(\nu\, Y : \mathrm{CTy}.\ B)\}$
by induction hypothesis on B

$$\begin{aligned}
&\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(\nu\, Y : \mathrm{CTy}.\ B)\ A\ m\\
\equiv\ &\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \lambda\, f.\ \mathrm{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ m\ \{\lambda\, x.\ \mathrm{unroll}(\ !\ f\ x\ )\})\\
\equiv\ &\lambda\, f.\ \mathrm{roll}(\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ m\ \{\lambda\, x.\ \mathrm{unroll}(\ !\ f\ x\ )\})\\
\equiv\ &\lambda\, f.\ \mathrm{roll}(\mathrm{Str}(B)[B_\nu/Y][str_\nu/str_Y]\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\ \{\lambda\, x.\ \mathrm{unroll}(\ !\ f\ x\ )\})\\
\equiv\ &\mathrm{Str}(\nu\, Y : \mathrm{CTy}.\ B)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

(6) **Case** $\mathrm{Str}(X)$ for $X : \mathrm{CTy}$: since the structure is from the value environment, it's lawful by the induction hypothesis.

(7) **Case** $\mathrm{Str}((\lambda\, X : K.\ B)\ S)$ for $S : K$ and $B : \mathrm{CTy}$: by induction hypothesis on S and B

$$\begin{aligned}
&\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}((\lambda\, X : K.\ B)\ S)\ A\ m\\
\equiv\ &\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ \mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A\ m\\
\equiv\ &\mathrm{Str}(B)[\lfloor\ S\ \rfloor/X][\{\mathrm{Str}(S)\}/str_X]\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}\\
\equiv\ &\mathrm{Str}((\lambda\, X : K.\ B)\ S)\ A\ \{\mathrm{do}\ m \leftarrow\ !\ tm\ ;\ !\ m\}
\end{aligned}$$

□