Contextual Modal Types for Algebraic Effects and Handlers

NIKITA ZYUZIN, IMDEA Software Institute, Spain and Universidad Politécnica de Madrid, Spain
ALEKSANDAR NANEVSKI, IMDEA Software Institute, Spain

Programming languages with algebraic effects often track the computations' effects using type-and-effect systems. In this paper, we propose to view an algebraic effect theory of a computation as a variable context; consequently, we propose to track algebraic effects of a computation with *contextual modal types*. We develop ECMTT, a novel calculus which tracks algebraic effects by a contextualized variant of the modal $\Box$ (necessity) operator, that it inherits from Contextual Modal Type Theory (CMTT).

Whereas type-and-effect systems add effect annotations on top of a prior programming language, the effect annotations in ECMTT are inherent to the language, as they are managed by programming constructs corresponding to the logical introduction and elimination forms for the $\Box$ modality. Thus, the type-and-effect system of ECMTT is actually just a type system.

Our design obtains the properties of local soundness and completeness, and determines the operational semantics solely by β-reduction, as customary in other logic-based calculi. In this view, effect handlers arise naturally as a witness that one context (i.e., algebraic theory) can be reached from another, generalizing explicit substitutions from CMTT.

To the best of our knowledge, ECMTT is the first system to relate algebraic effects to modal types. We also see it as a step towards providing a correspondence in the style of Curry and Howard that may transfer a number of results from the fields of modal logic and modal type theory to that of algebraic effects.

CCS Concepts: • **Theory of computation → Type theory**; **Modal and temporal logics**; • **Software and its engineering → Abstraction, modeling and modularity**; **Control structures**.

Additional Key Words and Phrases: algebraic effects, effect handlers, modal logic, modal types

ACM Reference Format:
Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual Modal Types for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 5, ICFP, Article 75 (August 2021), 29 pages. https://doi.org/10.1145/3473580

1 INTRODUCTION

Languages with algebraic effects [Plotkin and Power 2002, 2003, 2001] represent effects as calls to operations from algebraic theories. Effect handlers [Pretnar and Plotkin 2013] specify how these operations should be interpreted when the computation using them executes. Together, algebraic effects and handlers provide a flexible way for composing effects in a computation, and for evaluating a computation into a pure value by handling all of its effects.

For example, to combine effects of state and exception in a computation it suffices to merely use the operations from the algebraic theory of state alongside the operations from the algebraic theory of exceptions. The task of interpreting the combination of effects is delegated to the point when the computation executes through a handler for the effects of both theories. This is in contrast to monads, where the semantics is provided upfront, and remains unchanged through execution.

Authors' addresses: Nikita Zyuzin, IMDEA Software Institute, Campus de Montegancedo s/n, Pozuelo de Alarcon, Madrid, 28223, Spain, nikita.zyuzin@imdea.org and Universidad Politécnica de Madrid, Spain; Aleksandar Nanevski, IMDEA Software Institute, Campus de Montegancedo s/n, Pozuelo de Alarcon, Madrid, 28223, Spain, aleks.nanevski@imdea.org.

2475-1421/2021/8-ART75
https://doi.org/10.1145/3473580

Typically, a type system for algebraic effects keeps track of the operations of a program through effect annotations, deriving a type-and-effect system. A call to an operation is annotated with the invoked effect, which propagates to the type of the enclosing computation. Handling removes the effect from the annotation. Crucially, the search for the best type-and-effect system for algebraic effects is still open. For example, the recent work on effect handlers in OCaml [Sivaramakrishnan et al. 2021] highlights the current lack of a design in which effect annotations combine well with advanced language features such as polymorphism, modularity and generativity.

In this paper, we propose an entirely type-based design for a language with algebraic effects. We consider the operations used by a computation as its *effect context*, and track it through the typing mechanisms that we adapt from Contextual Modal Type Theory (CMTT) [Nanevski et al. 2008]. In particular, we derive a novel calculus for algebraic effects rooted in modal logic and modal type theory, which we name Effectful Contextual Modal Type Theory, or ECMTT for short.

CMTT derives from a contextual variant of intuitionistic modal logic S4 through the annotation of the inference rules with proof terms and a computational interpretation in the style of Curry-Howard correspondence. Modal logics in general reason about truth in a universe of possible worlds. In the specific case of S4, the key feature is the propositional constructor $\Box$ (called "necessity", or "box" for short). The proposition $\Box A$ is considered proved in the current world, if we can produce a proof of A in every possible future world [Alechina et al. 2001; Benton et al. 1998; Pfenning and Davies 2001; Simpson 1994]. The derived computational interpretations resulted in type systems for staged computation [Davies and Pfenning 2001] and run-time code generation [Wickline et al. 1998].

The contextual variant further indexes, or grades, the necessity constructor with a context of propositions Ψ. The proposition $[\Psi]A$ is considered proved in the current world if we can produce a proof of A in every possible future world, using *only* the propositions from Ψ as hypotheses. The modality $\Box A$ is recovered when the context Ψ is empty. The computational interpretation transforms propositions to types, and considers the type $[\Psi]A$ as classifying programs of type A that admit free variables from the variable context Ψ, and no others. The typing discipline has obtained calculi for meta-programming with open code [Nanevski and Pfenning 2005], for dynamic binding [Nanevski 2003a], for typed tactics and proof transformations in proof assistants [Pientka 2010; Stampoulis and Shao 2010; Stampoulis 2013], and has explained the notion of meta variable and related optimizations in logical frameworks [Pientka and Pfenning 2003].

ECMTT applies the contextual discipline by taking the context Ψ to be an algebraic theory—a signature of effect operations[1]—locally bound in a computation of type $[\Psi]A$. It allows ECMTT to integrate effects much stronger into the language than a typical type-and-effect system would. A typical type-and-effect system annotates the types of some existing language without changing the terms. In contrast, algebraic theories in ECMTT are manipulated at the level of terms as well. The constructs for context (algebraic theory in our view) binding and instantiation logically correspond to introduction and elimination forms for the modality. We prove that these constructs are in harmony [Pfenning 2009] by establishing their local soundness and completeness. The latter properties ensure that the typing rules for the modality are self-contained and independent of the other language features, and in turn imply the desired modularity of the type system design.

By connecting the disciplines of modal logic and of algebraic effects, we see ECMTT as a step towards transferring the ideas between them. For example, the work on modal type theory has considered polymorphism over contexts as first-class objects [Cave and Pientka 2013; Pientka 2008], which should have direct equivalent in polymorphism over effect theories [Biernacki et al. 2019; Brachthäuser et al. 2020; Hillerström and Lindley 2016; Leijen 2017; Zhang and Myers 2019].

[1]In this paper we consider only *free* algebraic theories that contain operations, but no equations between operations.

Because of the foundation in modal logic, we also expect that ECMTT will admit important logical properties and developments such as: Kripke semantics and normalization by evaluation [Bak 2017; Gratzer et al. 2019; Ilik 2013; Simpson 1994], proof search via cut-free sequent calculus (corresponding computationally to synthesizing programs with algebraic effects), clean equational theory, proof of strong normalization, scaling to dependent types [Ahman 2017; Gratzer et al. 2019], etc. We leave all of these considerations for future work.

1.1 Introducing Contextual Modality

To see how contextual types apply to algebraic effects, consider the algebraic theory of state with a single integer cell. As in related work on algebraic effects and handlers [Bauer 2018; Bauer and Pretnar 2015], this theory consists of the operations $\mathit{get} : \mathit{unit} \longrightarrow \mathit{int}$ and $\mathit{set} : \mathit{int} \longrightarrow \mathit{unit}$. The following program increments the state by 1, and has type *int*.

$$x \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (x+1);\ \mathtt{ret}\ x : \mathit{int}$$

Intuitively, we model this theory as a *variable context* $\mathit{St} \mathrel{\widehat{=}} \mathit{get} : \mathit{unit} \longrightarrow \mathit{int}, \mathit{set} : \mathit{int} \longrightarrow \mathit{unit}$,[2] which is *bound locally* in the scope of the computation, and is also listed in the computation's type:

$$\mathit{incr} \mathrel{\widehat{=}} \mathsf{box}\ \mathit{St}.\ x \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (x+1);\ \mathtt{ret}\ x : [\mathit{St}]\mathit{int} \qquad (1)$$

The term constructor box is inherited from CMTT, and *simultaneously* binds all the variables from the supplied context *St*. In this case, the effect operations *get* and *set* from *St* become available for use within the scope of box.

Operationally, box *thunks* the enclosed programs. The boxed program doesn't execute until explicitly forced, and is considered pure. Semantically, the type $[\mathit{St}]A$ classifies computations that use operations from the theory *St*—*but no other operations*—and return a value of type A upon termination. Of course, in general, we admit an arbitrary variable context (resp. algebraic theory) Ψ to be bound by box and $[\Psi]A$, not just the concrete one *St*. Crucially, the box constructor is the *introduction form* for $[\Psi]A$.

1.2 Context Reachability

In a contextual type system one has to describe how the propositions from one context Ψ, can be proved from the hypotheses in another context Ψ'. When such proofs can be constructed, one says that Ψ is reachable from Ψ', or that Ψ' reaches Ψ. As we discuss in Section 2, CMTT models context reachability using explicit substitutions. In ECMTT, we model context reachability somewhat differently, using *effect handlers*.

For example, the following is one possible handler for the algebraic theory *St*.

$$\begin{aligned} \mathit{handlerSt} \mathrel{\widehat{=}} (&\mathit{get}\,(x,k,z) \to \mathtt{cont}\ k\ z\ z,\\ &\mathit{set}\,(x,k,z) \to \mathtt{cont}\ k\ ()\ x,\\ &\mathit{return}\,(x,z) \to \mathtt{ret}\ (x,z)) \end{aligned}$$

As customary in languages for algebraic effects, an effect handler consists of a number of clauses showing how to interpret each effect operation upon its use in a computation.

In the case of *handlerSt*, the clause for *get* takes x (of type *unit*) as the call parameter, k as the continuation at the call site, and z as the handler's current state at the call site. It then calls k with z and z, thus modeling that *get* returns the current value of the state (the first z), and proceeds to execute handling from the unmodified state (the second z). For *set*, the call parameter x has type *int*, and the handler calls k with $()$ and x, thus modeling that *set* returns the (only) value $()$ of *unit*

[2]As we shall see in Section 3, the technical details of our variable typing will differ somewhat, and will make use of several different typing judgements, but the above types for *get* and *set* in the context *St* are sufficiently approximate for now.

type, and sets x as the new state for the rest of the handling. The *return* clause applies when the handled computation returns a value x and ends in state z. In *handlerSt*, x and z are paired up and returned to the enclosing scope by `ret` (x, z).

This *handlerSt* describes how to transform a program that uses the theory *St* into a program with no effects, and it models state by passing it explicitly via continuation calls. Because its clauses use no operations, we can say that *handlerSt* is a witness for *St* reaching the *empty* algebraic theory (i.e., the empty context). Of course, it's perfectly possible for handler clauses to invoke operations from a non-empty theory as well. We shall see examples of this in Section 3.

1.3 Eliminating Contextual Modality

Following CMTT, we adopt a `let`-style constructor as an elimination form for $[\Psi]A$. To illustrate this form, consider the following program that handles the *St* effects in *incr* using *handlerSt*.[3]

$$\texttt{let box}\ u = \mathit{incr}\ \texttt{in handle}\ u\ \mathit{handlerSt}\ 0$$

The reduction of this program binds the variable u to the term inside box in the definition of *incr* in (1) and proceeds with the scope of `let`, where the term is handled with *handleSt* starting from the initial state 0. This program evaluates to a pair of the produced value and state, $(0, 1)$, as we will illustrate in detail in Section 5.

The reduction for `let-box` is quite different from the one associated with monads and monadic bind. Unlike with monads, the body of *incr* isn't evaluated when *incr* is bound to u, but is evaluated later, when u is handled. In particular, if u itself doesn't appear in the scope of `let-box`, then the computation bound to u is never evaluated.

This kind of `let` construct is associated with comonads, and indeed the $[\Psi]$ modality of CMTT has been assigned a comonadic semantics [Gabbay and Nanevski 2013]. Similar `let` constructs have been considered in other modal calculi for (co)effects [Gaboardi et al. 2016; Nanevski 2003b, 2004; Orchard et al. 2019]. Common to them is the reduction that provides an unevaluated expression to a program term that's equipped for evaluating it; in our case, to a handler matching the effect operations of the expression. We don't consider the categorical semantics or the equational theory for our variant of $[\Psi]A$ here. However, one of the contributions of this paper is observing that the comonadic elimination form applies to handling of algebraic effects as well.

2 REVIEW OF CONTEXTUAL MODAL TYPE THEORY

The typing rules of CMTT rely on a two-context typing judgment from Figure 1. The context Γ contains the variables of the ordinary lambda calculus $x : A$, which we refer to as *value variables*, as they bind values in a call-by-value semantics. The context Δ contains *modal variables* $u :: A[\Psi]$ that are annotated with a type A, but also with a context Ψ of value variables. Such a modal variable stands for a term that may depend on the (value) variables in Ψ, but no other value variables.

The term box $\Psi.\, e$ captures this dependence. It binds Ψ in the scope of e, but also prevents e from using any value variables that may have been declared in the outside context. This is formalized in the rule $\Box I$ where, reading the rule upside-down, the context Γ from the conclusion is removed

[3]In this paper, we don't consider handler variables. Thus, when we write *handlerSt* in the code, the reader should assume that the definition of *handlerSt* is simply spliced in. Similarly for algebraic theories, e.g., *St*. Quantifying over contexts and substitutions has been done in the work on contextual types [Cave and Pientka 2013] and should directly transfer to our setting. We thus forego such addition and focus on the fundamental connection between modalities and algebraic effects.

Modal variable context	$\Delta ::= \cdot \mid \Delta, u :: A[\Psi]$
Value variable context	$\Gamma, \Psi ::= \cdot \mid \Gamma, x : A$

$$\frac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}\ \textsc{var} \qquad \frac{u :: A[\Psi] \in \Delta \qquad \Delta; \Gamma \vdash \sigma : \Psi}{\Delta; \Gamma \vdash u\,\langle\sigma\rangle : A}\ \textsc{mvar}$$

$$\frac{\Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x : A.\ e : A \to B} \to I \qquad \frac{\Delta; \Gamma \vdash e_1 : A \to B \qquad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : B} \to E$$

$$\frac{\Delta; \Psi \vdash e : A}{\Delta; \Gamma \vdash \mathsf{box}\ \Psi.\ e : [\Psi]A}\ \Box I \qquad \frac{\Delta; \Gamma \vdash e_1 : [\Psi]A \qquad \Delta, u :: A[\Psi]; \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \mathtt{let}\ \ \mathsf{box}\ u = e_1\ \mathtt{in}\ e_2 : B}\ \Box E$$

$$\frac{\Delta; \Gamma \vdash e_1 : A_1 \quad \cdots \quad \Delta; \Gamma \vdash e_n : A_n}{\Delta; \Gamma \vdash (x_1 \to e_1, \ldots, x_n \to e_n) : x_1 : A_1, \ldots, x_n : A_n}\ \textsc{esub}$$

Fig. 1. Typing rules of CMTT.

from the premise. This means, in particular, that the following term is well-typed[4]

$$\vdash \mathsf{box}\ x : int, y : int.\ (x + y) : [x : int, y : int]\, int$$

On the other hand, a term that, under box, uses value variables other than those bound by box, cannot be ascribed a type

$$\nvdash \lambda z : int.\ \mathsf{box}\ x : int, y : int.\ (x + y + z)$$

In Section 4 we will use a similar rule for ECMTT to capture that a term may contain effect operations from the algebraic theory Ψ, but not from an outside algebraic theory Γ.

However, box doesn't restrict the use of modal variables, as the premise and the conclusion of the $\Box I$ rule share the Δ context. To introduce a modal variable into Δ one uses the

$$\mathtt{let}\ \ \mathsf{box}\ u = e_1\ \mathtt{in}\ e_2$$

construct (rule $\Box E$), which binds u to *the unboxed body* of e_1 in the scope of e_2. For example, eliding the types and commas in iterated variable binding, if $e_1 = \mathsf{box}\ x\ y.\ (x + y)$, then u is bound to $x + y$, which is a term with free variables x and y. In the scope of e_2, u will be declared in Δ as $u :: int[x : int, y : int]$.

Because u may be bound to a term with free variables from its associated context Ψ, one can use u in a program only after providing a definition for *all* of the variables in Ψ. These definitions are given by explicit substitutions (rule ESUB) that guard the occurrences of u (rule MVAR). For example, the following term, in which u is guarded by the explicit substitution $(x \to 5, y \to 2)$, is well-typed and evaluates to 7.

$$\mathtt{let}\ \ \mathsf{box}\ u = \mathsf{box}\ x\ y.\ (x + y)\ \mathtt{in}\ u\,\langle x \to 5, y \to 2\rangle \tag{2}$$

[4]In examples, we freely use standard types such as *int* and product types, integer constants, and functions such as +, ∗, pairing and projections, without declaring them in the syntax and the typing rules. They pose no formal difficulties. We also consider the application of such (pure) expressions to evaluate immediately.

Because box doesn't restrict the use of modal variables, the following term, in which u occurs within a box, is also well-typed, with type $\Box int$, and it evaluates to $\mathsf{box}\ (5+2)$.[5]

$$\mathtt{let\ box}\ u = \mathsf{box}\ x\ y.\ (x+y)\ \mathtt{in}\ \mathsf{box}\ (u\ \langle x \to 5, y \to 2\rangle) \tag{3}$$

As mentioned in Section 1, in contrast to monadic bind, binding $\mathsf{box}\ \Psi.\ e_1$ to a modal variable u

$$\mathtt{let\ box}\ u = \mathsf{box}\ \Psi.\ e_1\ \mathtt{in}\ e_2$$

doesn't by itself cause the evaluation of e_1. Whether e_1 is evaluated depends on the occurrences of u in e_2. For instance, in the example term (3), u is thunked under a box. Thus, upon substituting u with $x+y$, the explicit substitution $(x \to 5, y \to 2)$ executes to produce $\mathsf{box}\ (5+2)$, after which the evaluation stops (i.e., boxed terms are values). In contrast, in the example term (2), u is not thunked, thus the evaluation proceeds for one more step to obtain 7. Moreover, e_2 may contain several occurrences of u, some thunked and some not, and each guarded by a different explicit substitution. For example, the following is a well-typed term

$$\begin{array}{l}\mathtt{let\ box}\ u = \mathsf{box}\ x\ y.\ (x+y)\ \mathtt{in}\\ \quad (u\ \langle x \to 5, y \to 2\rangle,\\ \quad\ \mathsf{box}\ x.\ u\ \langle x \to 3, y \to x\rangle + u\ \langle x \to x^2, y \to u\ \langle x \to 2x, y \to 1\rangle\rangle + x,\\ \quad\ \mathsf{box}\ x\ y.\ u\ \langle x \to y, y \to x\rangle)\end{array}$$

which evaluates to the triple

$$(7, \mathsf{box}\ x.\ (3+x) + (x^2 + (2x+1)) + x, \mathsf{box}\ x\ y.\ y + x)$$

of type $int \times [x : int]\, int \times [x : int, y : int]\, int$.

In general, local soundness for $\Box$ in CMTT is formalized as the β-reduction

$$\mathtt{let\ box}\ u = \mathsf{box}\ \Psi.\ e\ \mathtt{in}\ e' \ \mapsto_\beta\ e'[\Psi.e/\!/u]$$

where $e'[\Psi.e/\!/u]$ is a *modal substitution*, in which e substitutes for u in e', incurring the capture of variables from Ψ. We elide its definition here (it can be found in [Nanevski et al. 2008]), but do emphasize its key property. Namely, upon modally substituting e for u in $u\ \langle\sigma\rangle$, the result is *not* the expression $e\ \langle\sigma\rangle$, but the expression obtained after directly applying the substitution σ to e. We retain this design in ECMTT, where we develop a similar notion of modal substitution, except with context Ψ generalized to an algebraic theory, and explicit substitutions replaced by effect handlers. In particular, handling in ECMTT will apply to terms with free *operations*, which will let us incorporate handling into β-reduction over open terms in Section 5.

We close the review of CMTT by noting that local completeness for $\Box$ in CMTT is formalized as the following η-expansion

$$e : [\Psi]A\ \mapsto_\eta\ \mathtt{let\ box}\ u = e\ \mathtt{in}\ \mathsf{box}\ \Psi.\ u\ \langle \mathtt{id}_\Psi\rangle$$

where $\mathtt{id}_\Psi$ is the identity substitution for Ψ; that is, if $\Psi = x_1 : A_1, \ldots, x_n : A_n$, then $\mathtt{id}_\Psi \mathrel{\widehat{=}} (x_1 \to x_1, \ldots, x_n \to x_n)$. ECMTT will introduce a definition of an identity effect handler and appropriately adapt the above η-expansion.

3 OVERVIEW OF ECMTT BY EXAMPLES

To support algebraic effects and handlers, the syntax of ECMTT (Figure 2) diverges from CMTT in several important aspects which we summarize below, before proceeding to illustrate ECMTT through concrete programming examples.

[5] When binding the empty variable context $\cdot$, we abbreviate $[\cdot]A$ as $\Box A$ and $\mathsf{box}\ \cdot\ .\ e$ as $\mathsf{box}\ e$.

Modal context	Δ	$::= \cdot \mid \Delta, u :: A[\Psi] \mid \Delta, x : A$
Effect context	Γ, Ψ	$::= \cdot \mid \Gamma, op \div A \Rightarrow B \mid \Gamma, k \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} A \overset{S}{\Rightarrow} B$
Types	A, B, C, D, S	$::= P \mid A \rightarrow B \mid [\Psi]A$
Expressions	e	$::= x \mid \lambda x : A.\ e \mid e_1\ e_2 \mid \mathtt{box}\ \Psi.\ c \mid \mathtt{let\ box}\ u = e_1\ \mathtt{in}\ e_2$
Computations	c	$::= \mathtt{ret}\ e \mid x \leftarrow s;\ c \mid \mathtt{let\ box}\ u = e\ \mathtt{in}\ c$
Statements	s	$::= op\ e \mid \mathtt{cont}\ k\ e_1\ e_2 \mid \mathtt{handle}\ u\ [\Theta]\ h\ e$
Handlers	h	$::= return\,(x, z) \rightarrow c \mid h, op\,(x, k, z) \rightarrow c$
Handling sequences	Θ	$::= \bullet \mid \Theta, (h, e, x.\ c)$

Fig. 2. The syntax of ECMTT. In effect contexts, op ranges over algebraic operations, and k over continuations. Ψ ranges over effect contexts that are algebraic theories (i.e., that contain only algebraic operations and no continuations). P ranges over base types, including at least the singleton type *unit* with value $()$.

(1) We introduce a category of (effectful) computations. These are terms that explicitly sequence bindings of algebraic operations and continuation calls, and terminate with a return of a purely-functional result. Our formulation is based on the judgmental presentation of monadic computations by Pfenning and Davies [2001]. The category of expressions is inherited from CMTT, and retains the purely-functional nature.

(2) The context of value variables changes into effect context. It now contains effect operations $\mathrm{op} \div A \Rightarrow B$ and continuations $k \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} A \overset{S}{\Rightarrow} B$, each typed with a special new judgment. The modal type $[\Psi]A$ now classifies computations that use algebraic operations from the effect context Ψ, not purely-functional expressions as in CMTT.

(3) The value variables move to the modal context. The intuition is that values, being purely-functional, can be regarded as computations in the empty effect theory. Thus, the typing $x : A$ can be considered as a special case of the modal typing $x :: A[\cdot]$.

(4) We replace explicit substitutions with handlers, and more generally, handling sequences, as we shall see. Similar to explicit substitutions, an effect handler binds a computation to each algebraic operation in a context Ψ. In each binding, a handler further provides access to the argument x of the algebraic operation, and to the current continuation k, and to its current state z. A handler also provides a *return* binding that applies when handling $\mathtt{ret}\ e$ terms.

3.1 Effect Contexts, Algebraic Theories and Operations

Similarly to CMTT, the box constructor in ECMTT binds all the free variables in the underlying term. But in ECMTT, these free variables are algebraic operations (henceforth, simply *operations*), and the term under the box is a computation, not an expression. For example, we can box a computation that reads and returns the state using the algebraic theory $St \mathrel{\widehat{=}} get \div unit \Rightarrow int, set \div int \Rightarrow unit$ from the introduction, as follows.[6]

$$\mathtt{box}\ St.\ x \leftarrow get\ ();\ \mathtt{ret}\ x$$

Here, we apply the operation *get* on the argument of type *unit*, record the result in x and return x right away. The typing judgement $\div$ for operations is new, and we use it to ensure that an operation can only be invoked in a computation, not in an expression, as expressions are purely-functional. We illustrate this typing discipline in detail in Section 4.

[6]Except, this time, we type the algebraic operations in *St* with the actual variable judgment from ECMTT.

Importantly, box only binds contexts that contain operations (i.e., variables typed by $\div$), even though the new *effect context* Γ in ECMTT can also hold continuation variables (typed by $\dot{\div}$). We refer to the operation-only context as *algebraic theory* and we use Ψ to range over such contexts.

In the above example, *get* () is a *statement*, with *get* an *operation* and () its argument. Formally, statements are used with sequential composition, analogous to monadic bind

$$x \leftarrow s;\ c$$

which binds the result of s to x and continues with c. ECMTT features other forms of statements as well (e.g., continuation application and handling of modal variables), which we discuss later in this section. When the result of the statement is immediately returned, we abbreviate the computation into the statement alone. For example, we abbreviate the above term simply as

$$\mathsf{box}\ St.\ get\ ()$$

Of course, we retain from CMTT that box prevents the use of outside operations. For example, the following term doesn't type check because the inside box declares the empty algebraic theory as current, and rules out calls to the operations from the outside theory *St*.

$$\mathsf{box}\ St.\ \mathsf{box}\ (get\ ())$$

3.2 Modal Context, Value Variables and Handling of Modal Variables

In ECMTT we move the value variables into the modal context Δ, reserving the context Γ for operations and continuations. This implements our intention that modal types in ECMTT track effects, not the use of value variables as in CMTT. Moving value variables into Δ has the additional benefit that they now survive boxing. For example, consider the computation *incr* from the introduction, but this time we want to increment the state not by 1, but by a user-provided integer n. We achieve this by λ-binding n, and then invoking it under box.

$$incr_n\ \widehat{=}\ \lambda n : int.\ \mathsf{box}\ St.\ x \leftarrow get\ ();\ y \leftarrow set\ (x + n);\ \mathtt{ret}\ x$$

The resulting term $incr_n$ is an expression (i.e., it's purely functional), as it's a function whose body, being boxed, is itself an expression. Applying $incr_n$ to an integer reveals the boxed computation.

$$\mathtt{let\ box}\ u = incr_n\ 2\ \mathtt{in}\ \mathtt{handle}\ u\ handlerSt\ 0$$

As in CMTT, `let-box` above eliminates the box, binds the computation to a modal variable, and proceeds to handle the incrementing computation with *handlerSt* from the introduction.[7] Of course, we could have used any other handler for the theory *St*, just like we could guard a modal variable $u :: A[\Psi]$ in CMTT with any explicit substitution σ that defines the variables from Ψ.

An important term that becomes expressible by moving variables into Δ is the following coercion of a value x into a computation in a given theory Ψ, commonly known as *monadic unit*.

$$\lambda x{:}A.\ \mathsf{box}\ \Psi.\ \mathtt{ret}\ x$$

3.3 Handlers

Effect handlers are similar to explicit substitutions in that they replace operation variables with computations that define them. But there are important differences as well. In particular, operation clauses of a handler can manipulate the control flow of the target program by providing access to the current (delimited) continuation of the operation call. This facilitates a wide variety of effects [Pretnar 2015; Pretnar and Plotkin 2013], as we shall illustrate.

[7] While the formal syntax for handling is $\mathtt{handle}\ u\ [\Theta]\ h\ e$, by convention we elide $[\Theta]$ when Θ is empty, as it's here. We explain the need and the use of Θ in Section 3.5.

3.3.1 Return Clause. A handler in ECMTT always has a *return* clause, which applies when the handled computation terminates with $\texttt{ret}\ e$. The clause specifies how the handler processes e, together with the ending handler state of the computation. To illustrate, consider the following simple handler that consists only of a *return* clause that produces a pair of the final result and state:

$$\mathit{simple}\ \widehat{=}\ \mathit{return}\,(x, z) \to \texttt{ret}\ (x, z)$$

This handler handles no operations; thus its theory is empty, and it only applies to computations over the empty theory. For example, we can run *simple* on $\texttt{ret}\ 42$ with initial state 5 as follows.

$$\texttt{let box}\ u = \texttt{box}\,(\texttt{ret}\ 42)\ \texttt{in handle}\ u\ \mathit{simple}\ 5$$

In the *return* clause of *simple*, the $\texttt{ret}$ value of 42 is bound to x and 5 is bound to z to further return $(42, 5)$. If we used a handler with a *return* clause that makes no mention of x and z, e.g.,

$$\mathit{simple}\ \widehat{=}\ \mathit{return}\,(x, z) \to \texttt{ret}\ 7 \tag{4}$$

then the same program returns 7 regardless of the initial state supplied to $\texttt{handle}$ for u.

3.3.2 Updating Handler State. Handler state is modified by invoking continuations in operation clauses with new state values. To illustrate, consider an extension of *simple* with an operation $\mathit{op} \div \mathit{unit} \Rightarrow \mathit{int}$.

$$\begin{aligned}\mathit{simple}^*\ \widehat{=}\ (&\mathit{op}\,(x, k, z) \to \texttt{cont}\ k\ 1\ (z + 4),\\ &\mathit{return}\,(x, z) \to \texttt{ret}\ (x, z))\end{aligned}$$

The handler clause for *op* shows how simple^* interprets occurrences of $\mathit{op}\ e$ in a handled term: it binds the argument e to x, the current continuation up to the enclosing $\texttt{box}$ to k, the current handler state to z, and proceeds with the clause body. In the clause body, the application $\texttt{cont}\ k\ 1\ (z + 4)$ indicates that k continues with 1 as the return value of *op*, and $z + 4$ as the new state. If we apply simple^*, with initial state 5, to $\texttt{ret}\ 42$, then handling returns $(42, 5)$, as before. If we apply simple^* with initial state 5 to the computation that uses *op* non-trivially,

$$\begin{aligned}&\texttt{let box}\ u = \texttt{box}\ \mathit{op} \div \mathit{unit} \Rightarrow \mathit{int}.\\ &\quad y_1 \leftarrow \mathit{op}\,();\ y_2 \leftarrow \mathit{op}\,();\ y_3 \leftarrow \mathit{op}\,();\ \texttt{ret}\ (y_1 + y_2 + y_3)\\ &\ \texttt{in handle}\ u\ \mathit{simple}^*\ 5\end{aligned} \tag{5}$$

then handling returns $(3, 17)$. To see this, consider that each call to *op* is handled by returning 1, and therefore all y_i hold 1. Each call to *op* also increments the initial state by 4. Thus, at the end of handling, when the *return* clause is invoked over $\texttt{ret}\ (y_1 + y_2 + y_3)$, x and z variables of the *return* clause will bind 3 and $5 + 4 + 4 + 4 = 17$, respectively, to produce $(3, 17)$.

We can now revisit the handler for *St* from the introduction.

$$\begin{aligned}\mathit{handlerSt}\ \widehat{=}\ (&\mathit{get}\,(x, k, z) \to \texttt{cont}\ k\ z\ z,\\ &\mathit{set}\,(x, k, z) \to \texttt{cont}\ k\ ()\ x,\\ &\mathit{return}\,(x, z) \to \texttt{ret}\ (x, z))\end{aligned}$$

Because *get* is handled by invoking $\texttt{cont}\ k\ z\ z$, it's apparent that the handler interprets *get* as an operation that returns the value of the current state z and continues the execution without changing this state. On the other hand, $\mathit{set}\,x$ is handled by invoking $\texttt{cont}\ k\ ()\ x$; thus, the handler interprets *set* as returning $()$ and changing the current state to x, as one would expect.

3.3.3 Parametrized Handlers and Function Purity. As shown above, our handlers provide access to handler state, which is shared between handler clauses, and can be updated through continuations. In the parlance of algebraic effects, we thus provide *deep parametrized handlers* [Hillerström et al. 2020]. Most of the other algebraic effect systems, however, don't use parametrized handlers. For example, Eff [Bauer and Pretnar 2015] would encode *handlerSt* as

$$\begin{aligned} \mathit{handlerSt} \;\widehat{=}\; (&\mathit{get}(x,k) \to \lambda z.\, \mathtt{cont}\ k\ z\ z,\\ &\mathit{set}(x,k) \to \lambda z.\, \mathtt{cont}\ k\ ()\ x,\\ &\mathit{return}(x) \to \lambda z.\, \mathtt{ret}\ (x,z)) \end{aligned}$$

where the state z is lambda-bound in each clause. Such a definition wouldn't type check in ECMTT as a continuation call in the clause is a computation and can't be lambda-abstracted directly, because a function body must be pure. The latter, however, is a common and important design aspect of languages that encapsulate effects using types (e.g., Haskell). Thus, parametrized handlers arise naturally as a way to avoid lambda-abstraction over effectful handler clauses, if one is in a setting where effects are encapsulated. The indirect benefit is reducing closure allocation, the original motivation for parametrized handlers [Hillerström et al. 2020].

3.3.4 Uncalled Continuations. If an operation clause of a handler doesn't invoke the continuation k, this aborts the evaluation of the handled term. For example, let us extend simple^* with an operation $\mathit{stop} \div \mathit{unit} \Rightarrow \mathit{int}$ that immediately returns a pair of 42 and the handler state.

$$\begin{aligned} \mathit{simple}^\dagger \;\widehat{=}\; (&\mathit{op}\,(x,k,z) \to \mathtt{cont}\ k\ 1\ (z+4),\\ &\mathit{stop}\,(x,k,z) \to \mathtt{ret}\ (42,z),\\ &\mathit{return}\,(x,z) \to \mathtt{ret}\ (x,z)) \end{aligned}$$

If invoked on the computation from example (5) with the same initial state 5, $\mathit{simple}^\dagger$ returns (3, 17) as before, since (5) doesn't make calls to *stop*. If the handled computation changed the middle call from *op* to *stop* as in

$$y_1 \leftarrow \mathit{op}\,();\ y_2 \leftarrow \mathit{stop}\,();\ y_3 \leftarrow \mathit{op}\,();\ \mathtt{ret}\ (y_1+y_2+y_3) \tag{6}$$

then handling reaches neither the last *op*, nor $\mathtt{ret}\ (y_1 + y_2 + y_3)$. It effectively terminates after handling *stop*, to return (42, 9), as instructed by the *stop* clause of the handler. The state 9 is obtained after the initial state 5 is incremented by 4 through the handling of the first call to *op*.

Eliding continuations makes it possible to handle a theory of exceptions [Pretnar 2015]

$$\mathit{Exn} \;\widehat{=}\; \mathit{raise} \div \mathit{unit} \Rightarrow \bot$$

A handler for *Exn*, such as the following one

$$\begin{aligned} \mathit{handlerExn} \;\widehat{=}\; (&\mathit{raise}\,(x,k,z) \to \mathtt{ret}\ 42,\\ &\mathit{return}\,(x,z) \to \mathtt{ret}\ x) \end{aligned}$$

can't invoke k in the clause for *raise*, because k requires the result of *raise* as an input, which must be a value of the uninhibited type $\bot$. As such an argument can't be provided, the handler must terminate with some value when it encounters *raise*, thus precisely modeling how exceptions are handled in functional programming.

3.3.5 Non-tail Continuation Calls. So far, our example handlers either invoked the continuation variable k as the last computation step (tail call), or did not invoke k at all. In ECMTT, k may be invoked in other ways as well.

To illustrate, consider the following handler for the theory $op, stop \div unit \Rightarrow int$ from before. The handler invokes k in non-tail calls, to count the occurrences of op and $stop$ in the handled term.

$$\begin{aligned} handlerCount \ \widehat{=}\ (&op\,(x,k,z) \to y \leftarrow \texttt{cont}\ k\ 1\ z;\ \texttt{ret}\ (\pi_1\, y+1, \pi_2\, y),\\ &stop\,(x,k,z) \to y \leftarrow \texttt{cont}\ k\ 1\ z;\ \texttt{ret}\ (\pi_1\, y, \pi_2\, y+1),\\ &return\,(x,z) \to \texttt{ret}\ (0,0)) \end{aligned}$$

To explain *handlerCount*, first consider the *return* clause. Return clauses apply when handling values; as values are pure, they can't call operations. Thus the *return* clause of *handlerCount* returns $(0,0)$ to signal that a value contains 0 occurrences of both *op* and *stop*.

Next consider the operation clauses. Each clause executes $\texttt{cont}\ k\ 1\ z$ to invoke the current continuation k—which holds the handled variant of the remaining computation—with argument 1 and current state z.[8] The obtained result, bound to y, is a pair containing the number of uses for *op* and *stop* in the the remaining computation. Each clause then adds 1 for the currently-handled operation to the appropriate projection of y. For example, handling (6) returns $(2,1)$, as expected.

3.3.6 Multiply-called Continuations. A handler clause in ECMTT may also invoke k several times; in the literature such continuations are usually called *multi-shot* [Bruggeman et al. 1996]. This feature is useful for modeling the algebraic theory of *non-determinism*

$$NDet\ \widehat{=}\ choice \div unit \Rightarrow bool$$

in which the operation *choice* provides an unspecified Boolean value, as in the following program.

$$y \leftarrow choice\,();\ \texttt{if}\ y\ \texttt{then ret}\ 4\ \texttt{else ret}\ 5 \tag{7}$$

A handler for *NDet* chooses how to interpret the non-determinism. For example, the handler below enumerates all the possible options for *choice* and collects the respective outputs of the target program into a list. Here $[x]$ is a list with single element x, and ++ is list append.

$$\begin{aligned} handlerNDet\ \widehat{=}\ (&choice\,(x,k,z) \to y_1 \leftarrow \texttt{cont}\ k\ true\ z;\ y_2 \leftarrow \texttt{cont}\ k\ false\ z;\ \texttt{ret}\ (y_1 \mathbin{++} y_2),\\ &return\,(x,z) \to \texttt{ret}\ [x]) \end{aligned}$$

Applying the handler to (7) returns the list $[4,5]$.

3.4 Operations in Handler Clauses

Our example handlers so far used no outside operations, and thus handled into the empty theory. But in ECMTT handlers can handle into other theories as well. For example, the following is an alternative handler for *St* which throws an exception whenever the state is set to 13; thus it uses the operation *raise* and handles *St* into the theory *Exn*.

$$\begin{aligned} handlerExplosiveSt\ \widehat{=}\ (&get\,(x,k,z) \to \texttt{cont}\ k\ z\ z,\\ &set\,(x,k,z) \to \texttt{if}\ x=13\ \texttt{then}\ raise\,()\ \texttt{else cont}\ k\ ()\ x,\\ &return\,(x,z) \to \texttt{ret}\ (x,z)) \end{aligned}$$

Using *handlerExplosiveSt* to handle $incr_n\ 1$ results in a computation from the theory *Exn*. The type system in Section 4 will require us to make the dependence on *Exn* explicit. For example, the following function *explode* takes input m and builds a box thunk with *Exn*. Within it, we use *handlerExplosiveSt* over u with m as the initial state, and return the first projection of the result.

$$\begin{aligned} explode\ \widehat{=}\ \lambda m.\ &\texttt{let box}\ u = incr_n\ 1\ \texttt{in}\\ &\quad \texttt{box}\ Exn.\ x \leftarrow \texttt{handle}\ u\ handlerExplosiveSt\ m;\\ &\qquad \texttt{ret}\ (\pi_1\, x) \end{aligned}$$

[8] Both 1 and z are irrelevant for the execution of the example, but they make the example typecheck.

If we evaluate *explode* 0, we obtain box *Exn*. `ret` 0 as follows. The unboxed body of $incr_n$ 1 is first passed to *handlerExplosiveSt* for handling with initial state 0. This results in the pair $(0, 1)$ indicating that $incr_n$ 1 read the initial state, and returned the value read together with the incremented state. As we shall formalize in Section 5, the binding of the pair to x is immediately reduced, and the first projection is taken, to obtain `ret` 0, which is finally thunked with box *Exn*.

On the other hand, if we evaluate *explode* 12, then we obtain box *Exn*. *raise* (), because the handling for the unboxed body of $incr_n$ 1 with *handlerExplosiveSt* terminates with *raise* upon trying to set the state to 13.

We can proceed with handling by *handlerExn* from initial state (), as in

$$\texttt{let box}\ v = \mathit{explode}\ 0\ \texttt{in}\ \texttt{handle}\ v\ \mathit{handlerExn}\ ()$$

where the *return* clause of *handlerExn* returns 0. Or, if we change the initial state to 12,

$$\texttt{let box}\ v = \mathit{explode}\ 12\ \texttt{in}\ \texttt{handle}\ v\ \mathit{handlerExn}\ () \tag{8}$$

then the *raise* clause of *handlerExn* returns 42.

3.4.1 Identity Handler. Given a theory Ψ, the identity handler $\texttt{id}_\Psi$ handles Ψ into itself, or more generally, into any theory that includes the operations of Ψ. Following the definition of identity substitution of CMTT from Section 2, the identity handler is not an ECMTT primitive, but a meta definition, uniformly given for each Ψ.

$$\begin{aligned}\texttt{id}_\Psi\ \widehat{=}\ (&op_i\,(x, k, z) \to y \leftarrow op_i\ x;\ \texttt{cont}\ k\ y\ z,\\ &return\,(x, z) \to \texttt{ret}\ x)\end{aligned}$$

Each operation op_i of Ψ is handled by invoking it with the same argument x with which op_i is encountered, and passing the obtained result to the awaiting continuation. The *return* clause also just further returns the encountered value. The handler state z is merely propagated by operation clauses, and thus doesn't influence the computation. For convenience, we will thus consider that in $\texttt{id}_\Psi$ the variables z are of type *unit*, and always invoke $\texttt{id}_\Psi$ with initial state (). Using the identity handler, we provide the ECMTT variant of η-expansion for modal types in analogy to that of CMTT in Section 2.

$$e : [\Psi]A\ \mapsto_\eta\ \texttt{let box}\ u = e\ \texttt{in}\ \texttt{box}\ \Psi.\ \texttt{handle}\ u\ \texttt{id}_\Psi\ ()$$

3.4.2 Combining Theories. Suppose that we are given the function *safeDiv* that implements division and raises an exception if the divisor is 0.

$$\mathit{safeDiv}\ \widehat{=}\ \lambda x : \mathit{int}.\ \lambda y : \mathit{int}.\ \texttt{box}\ \mathit{Exn}.\ \texttt{if}\ y = 0\ \texttt{then}\ \mathit{raise}\ ()\ \texttt{else}\ \texttt{ret}\ (x/y)$$

Obviously, the body of *safeDiv* abstracts over the *Exn* theory. Suppose that we now wanted to use *safeDiv* in a program that also operates over state, i.e., over theory *St*. One way to do so would be to handle the calls to *safeDiv* by a handler, such as *handlerExn* for example, that catches the exception and returns a pure value. But identity handlers provide another way as well. We can combine the theories of *St* and *Exn* on the fly into a common theory[9], and then simply delay resolving exceptions to the common theory.

$$\begin{aligned}\mathit{divFromState}\ \widehat{=}\ \texttt{box}\ \mathit{St}, \mathit{Exn}.\ y \leftarrow \mathit{get}\ ();\ &\\ &\texttt{let box}\ u = \mathit{safeDiv}\ 42\ y\ \texttt{in}\\ &\texttt{handle}\ u\ \texttt{id}_{\mathit{Exn}}\ ()\end{aligned}$$

[9]We assume here that the combined theories don't share operation names, and can thus be concatenated without clashes and variable shadowing, as indeed is the case for *St* and *Exn*.

For example, *divFromState* reads the current state into y and invokes *safeDiv* on 42 and y, re-raising an eventual exception by handling with $\mathtt{id}_{Exn}$. The program *divFromState* explicitly boxes over both *St* and *Exn* to signal that the theories are combined into a common context.

3.5 Handling Sequences

In ECMTT, a handler is syntactically always applied to a variable, rather than to an expression, similarly to explicit substitutions in CMTT. The design allows us to express handling as part of modal variable substitution and thus, correspondingly, as part of β-reduction. Unlike in other calculi where handling is applied over closed terms during evaluation, our handling applies over terms with free variables, just like β-reduction. Moreover, β-reduction suffices to implement handling. Indeed, if we allowed applying handlers to general computations, as in

$$\mathtt{handle}\ c\ h\ s$$

then $\mathtt{handle}\ c\ h\ s$ must be a redex, with reductions for it additional to β-reduction.

However, this design causes a problem with handler composition, which occurs in the presence of free variables. To see the issue, consider the following example which slightly reformulates (8).

$$\begin{array}{ll}\mathtt{let\ box}\ v = \mathtt{box}\ \mathit{Exn}.\ (\mathtt{let\ box}\ u = \mathit{incr}_n\ 1\ \mathtt{in} & \\ \qquad\qquad\qquad\qquad\quad x \leftarrow \mathtt{handle}\ u\ \mathit{handlerExplosiveSt}\ 12; & \\ \qquad\qquad\qquad\qquad\quad \mathtt{ret}\ (\pi_1\, x)) & \\ \mathtt{in\ handle}\ v\ \mathit{handlerExn}\ () & \end{array} \tag{9}$$

This term reduces by first binding v to the whole computation under the *Exn* box. Next, the computation v is handled by *handlerExn*, before any reduction in v itself. Therefore we must be able to handle all the subterms of v, including

$$\mathtt{handle}\ u\ \mathit{handlerExplosiveSt}\ 12$$

which already contains handling by *handlerExplosiveSt*. To handle this expression, we can't simply return $\mathtt{handle}\ (\mathtt{handle}\ u\ \mathit{handlerExplosiveSt}\ 12)\ \mathit{hadnlerExn}\ ()$. That isn't syntactically well-formed, as it applies a handler to a general computation, rather than a variable. Intuitively, we would like to return an expression of the form

$$\mathtt{handle}\ u\ (\mathit{handlerExn} \circ \mathit{handlerExplosiveSt})\ 12 \tag{10}$$

which records that eventual substitutions of u must first execute *handlerExplosiveSt* over the substituted term, followed by *handlerExn*.

One may think that composing *handlerExn* and *handlerExplosiveSt* requires simply applying *handlerExn* to all the clauses of *handlerExplosiveSt*; unfortunately, this isn't correct.[10] When applying *handlerExplosiveSt* and *handlerExn* in succession to some computation c, the handling by *handlerExn* requires instantiating its continuation variables with computations obtained after *handlerExplosiveSt* is applied to c. In the above case, c is unknown, with variable u serving as a placeholder. Until u is substituted, we don't have access to continuations necessary to execute *handlerExn*, and thus can't eagerly apply it to the clauses of *handlerExplosiveSt*.

To encode the composition of handlers, we thus introduce the *handling sequence* Θ. The term

$$\mathtt{handle}\ u\ [\Theta]\ h\ e$$

stands for applications of handlers from the list Θ, in sequence from left to right, on a term bound to u, followed by the handler h running on the result, using initial state e. However, Θ can't merely be

[10]Incidentally, such direct approach works in CMTT for composing explicit substitutions [Nanevski et al. 2008], but handlers are much more involved than explicit substitutions.

a list of handlers, as each handler must also be associated with the initial state, and the continuation that's appropriate for it. For example, the immediate β-reduction of example (9) will be

$$\begin{array}{l} \texttt{let box}\ u = incr_n\ 1\ \texttt{in} \\ \quad \texttt{handle}\ u\ [(handlerExplosiveSt, 12, x.\ \texttt{ret}\ (\pi_1\ x))]\ handlerExn\ () \end{array}$$

recording that $\texttt{ret}\ (\pi_1\ x)$ is the immediate continuation of $\texttt{handle}\ u\ handlerExplosiveSt\ 12$ in (9).

Finally, the need for handler sequences arises only when expressing the intermediate results of program reduction, and we don't expect that a user of our system will ever write them by hand. The simplified syntax without Θ suffices for source code examples.

4 TYPING

ECMTT uses the following variable judgements:

$x : A \in \Delta$	values of type A
$u :: A[\Psi] \in \Delta$	computations of type A in an algebraic theory Ψ
$op \div A \Rightarrow B \in \Gamma$	operations with input type A and output type B
$k \mathbin{\dot{\div}} A \overset{S}{\Rightarrow} B \in \Gamma$	continuations with input type A, state type S, and output type B

We also have a separate judgement for each of the syntactic categories:

$\Delta \vdash e : A$	e is an expression of type A
$\Delta; \Gamma \vdash c \div A$	c is a computation of type A
$\Delta; \Gamma \vdash s \div_{\mathsf{s}} A$	s is a statement of type A
$\Delta; \Gamma \vdash h \div A[\Psi] \overset{S}{\Rightarrow} B$	h is a handler for computations of type A in an algebraic theory Ψ that uses state parameter of type S and handles into computations of type B
$\Delta; \Psi \vdash \Theta \div A[\Psi'] \Rrightarrow B$	Θ is a handling sequence for computations of type A in an algebraic theory Ψ' that produces computations of type B in a theory Ψ

We show the typing rules of ECMTT on Figure 3. Let us review their key parts:

(1) The variable typing rules VAR, OP, CONT and MVAR are grouped first. The rule MVAR guards a modal variable u with a handling sequence Θ, handler h, and initial state e. It ensures that the context Ψ of u matches the range theory of Θ, and that the context Ψ' of Θ matches the range theory of h. Thus, the handling of u proceeds by Θ, then h. The judgment for handling sequences formalizes this intermediary role of Θ by using theory Ψ that contains only operations for its effect context, as opposed to the more general effect context Γ that also includes continuations.

(2) Unlike in CMTT, the judgment for expressions elides the context Γ, because expressions can't have effects. On the other hand, the rules dealing with function and modal types are almost unchanged. The only distinctions are that $\Box I$ now boxes computations instead of expressions (introducing an effect context Ψ, not replacing it), and that we also have a rule $\Box E$-COMP that eliminates modal types into computations. The latter is a standard feature of `let`-forms in calculi with multiple judgments [Pfenning and Davies 2001].

(3) The RET rule coerces pure expressions into computations, while the BIND rule sequentially composes a statement s with a computation c. Notice that the term $\texttt{ret}\ e$ is a computation, not a statement. Thus, strictly speaking, the syntax $x \leftarrow \texttt{ret}\ e; c$ is not valid in ECMTT.

$$\frac{x : A \in \Delta}{\Delta \vdash x : A}\ \textsc{var} \qquad \frac{op \div A \Rightarrow B \in \Gamma \qquad \Delta \vdash e : A}{\Delta; \Gamma \vdash op\ e \div_{\mathsf{s}} B}\ \textsc{op}$$

$$\frac{k \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} A \overset{S}{\Longrightarrow} B \in \Gamma \qquad \Delta \vdash e_1 : A \qquad \Delta \vdash e_2 : S}{\Delta; \Gamma \vdash \texttt{cont}\ k\ e_1\ e_2 \div_{\mathsf{s}} B}\ \textsc{cont}$$

$$\frac{u :: A[\Psi] \in \Delta \qquad \Delta; \Psi' \vdash \Theta \div A[\Psi] \Rrightarrow B \qquad \Delta; \Gamma \vdash h \div B[\Psi'] \overset{S}{\Longrightarrow} C \qquad \Delta \vdash e : S}{\Delta; \Gamma \vdash \texttt{handle}\ u\ [\Theta]\ h\ e \div_{\mathsf{s}} C}\ \textsc{mvar}$$

$$\frac{\Delta, x : A \vdash e : B}{\Delta \vdash \lambda x : A.\ e : A \longrightarrow B}\ {\longrightarrow}I \qquad \frac{\Delta \vdash e_1 : A \longrightarrow B \qquad \Delta \vdash e_2 : A}{\Delta \vdash e_1\ e_2 : B}\ {\longrightarrow}E$$

$$\frac{\Delta; \Psi \vdash c \div A}{\Delta \vdash \texttt{box}\ \Psi.\ c : [\Psi]A}\ \Box I \qquad \frac{\Delta \vdash e_1 : [\Psi]A \qquad \Delta, u :: A[\Psi] \vdash e_2 : B}{\Delta \vdash \texttt{let box}\ u = e_1\ \texttt{in}\ e_2 : B}\ \Box E$$

$$\frac{\Delta \vdash e : [\Psi]A \qquad \Delta, u :: A[\Psi]; \Gamma \vdash c \div B}{\Delta; \Gamma \vdash \texttt{let box}\ u = e\ \texttt{in}\ c \div B}\ \Box E\textsc{-comp}$$

$$\frac{\Delta \vdash e : A}{\Delta; \Gamma \vdash \texttt{ret}\ e \div A}\ \textsc{ret} \qquad \frac{\Delta; \Gamma \vdash s \div_{\mathsf{s}} A \qquad \Delta, x : A; \Gamma \vdash c \div B}{\Delta; \Gamma \vdash x \longleftarrow s;\ c \div B}\ \textsc{bind}$$

$$\frac{\Delta, x : A, z : S; \Gamma \vdash c \div C}{\Delta; \Gamma \vdash \mathit{return}\,(x, z) \to c \div A[\cdot] \overset{S}{\Longrightarrow} C}\ \textsc{reth}$$

$$\frac{\Delta; \Gamma \vdash h \div C[\Psi] \overset{S}{\Longrightarrow} C' \qquad \Delta, x : A, z : S; \Gamma, k \mathrel{\overset{\cdot}{\underset{\cdot}{\sim}}} B \overset{S}{\Longrightarrow} C' \vdash c \div C'}{\Delta; \Gamma \vdash h, op\ (x, k, z) \to c \div C[\Psi, op \div A \Rightarrow B] \overset{S}{\Longrightarrow} C'}\ \textsc{oph}$$

$$\frac{\Psi' \subseteq \Psi}{\Delta; \Psi \vdash \bullet \div A[\Psi'] \Rrightarrow A}\ \textsc{emph}$$

$$\frac{\Delta; \Psi' \vdash \Theta \div A[\Psi''] \Rrightarrow B \qquad \Delta; \Psi \vdash h \div B[\Psi'] \overset{S}{\Longrightarrow} C \qquad \Delta \vdash e : S \qquad \Delta, x : C; \Psi \vdash c \div D}{\Delta; \Psi \vdash \Theta, (h, e, x.\ c) \div A[\Psi''] \Rrightarrow D}\ \textsc{seqh}$$

Fig. 3. Typing rules of ECMTT.

This is not a restriction in programming, as the intended behavior can be introduced as a syntactic sugar via boxing.[11]

[11] One solution is $x \leftarrow \texttt{ret}\ e; c \ \widehat{=}\ \texttt{let box}\ u = \texttt{box}\ (\texttt{ret}\ e)\ \texttt{in}\ x \leftarrow \texttt{handle}\ u\ \texttt{id}.\ (); c$, and there are others.

(4) The rules RETH and OPH deal with building handlers. RETH is the base case, giving a handler for the empty theory $\cdot$ which thus contains only the *return* clause. OPH is the inductive case, adding a clause for a new operation op to a handler h, thus extending h's theory Ψ with $op \div A \Rightarrow B$. We add new clauses to the right, but retain the syntax from Section 3, where the *return* clause is presented as the last in a handler.

(5) The rules EMPH and SEQH deal with building handling sequences. EMPH gives an empty sequence, and SEQH adds a handler clause to Θ, ensuring proper sequencing, i.e., that the theory Ψ' of the added handler h matches the effect context of Θ.

4.1 Examples

We illustrate the typing rules of ECMTT by showing a few example typing judgments and derivations. We assume the theories of state and exceptions from Section 3.

$$St \mathrel{\widehat{=}} get \div unit \Rightarrow int, set \div int \Rightarrow unit$$
$$Exn \mathrel{\widehat{=}} raise \div unit \Rightarrow \bot$$

(1) $\vdash \mathsf{box}\ St.\ get\ () : [St]\,int$
(2) $\vdash \lambda n.\,\mathsf{box}\ St.\ (x \leftarrow get\ ();\ y \leftarrow set\ (x+n);\ \mathtt{ret}\ x) : int \rightarrow [St]\,int$
(3) $\nvdash \mathsf{box}\ St.\ \mathtt{ret}\ (\mathsf{box}\ (get\ ()))$
(4) $\vdash handlerSt \div int[St] \overset{int}{\Longrightarrow} int \times int$
(5) $\vdash handlerExn \div int[Exn] \overset{unit}{\Longrightarrow} int$
(6) $Exn \vdash handlerExplosiveSt \div int[St] \overset{int}{\Longrightarrow} int \times int$
(7) $Exn \vdash (handlerExplosiveSt, 12, x.\ \mathtt{ret}\ (\pi_1\ x)) \div int[St] \Rrightarrow int$
(8) $u :: int[St]; \cdot \vdash \mathtt{handle}\ u\ [(handlerExplosiveSt, 12, x.\ \mathtt{ret}\ (\pi_1\ x))]\ handlerExn\ () \div int$
(9) If $\Psi \subseteq \Psi'$, then $\Psi' \vdash \mathtt{id}_\Psi \div A[\Psi] \overset{unit}{\Longrightarrow} A$
(10) If $\Psi \subseteq \Psi'$, then $\vdash \lambda e.\,\mathtt{let}\ \mathsf{box}\ u = e\ \mathtt{in}\ \mathsf{box}\ \Psi'.\ \mathtt{handle}\ u\ \mathtt{id}_\Psi\ () : [\Psi]A \rightarrow [\Psi']A$
(11) $\vdash \lambda x.\,\mathsf{box}\ \Psi.\ \mathtt{ret}\ x : A \rightarrow [\Psi]A$
(12)
$$\begin{aligned} \vdash \lambda f.\,\lambda x.\,\mathtt{let}\ & \mathsf{box}\ u = f \\ & \mathsf{box}\ v = x \\ \mathtt{in}\ \mathsf{box}\ \Psi.\ & a \leftarrow \mathtt{handle}\ u\ \mathtt{id}_\Psi\ (); \\ & b \leftarrow \mathtt{handle}\ v\ \mathtt{id}_\Psi\ (); \\ & \mathtt{ret}\ (a\ b) : [\Psi](A \rightarrow B) \rightarrow [\Psi]A \rightarrow [\Psi]B \end{aligned}$$
(13)
$$\begin{aligned} \vdash \lambda x.\,& \mathtt{let}\ \mathsf{box}\ u = x\ \mathtt{in} \\ & \mathsf{box}\ \Psi.\ a \leftarrow \mathtt{handle}\ u\ \mathtt{id}_\Psi\ (); \\ & \quad \mathtt{let}\ \mathsf{box}\ v = a\ \mathtt{in}\ \mathtt{handle}\ v\ \mathtt{id}_\Psi\ () : [\Psi][\Psi]A \rightarrow [\Psi]A \end{aligned}$$

4.1.1 Typing Computations. Derivation (1) shows that we can invoke an operation declared in the current theory. The derivation expands the syntactic sugar whereby the computation consisting of a statement s actually abbreviates $x \leftarrow s; \mathtt{ret}\ x$.

$$\dfrac{\dfrac{\dfrac{get \div unit \Rightarrow int \in St \qquad \dfrac{}{St \vdash () : unit}}{St \vdash get\ () \div_s int}\,\textsc{op} \qquad \dfrac{\dfrac{}{x : int \vdash x : int}\,\textsc{var}}{x : int; St \vdash \mathtt{ret}\ x \div int}\,\textsc{ret}}{St \vdash x \leftarrow get\ (); \mathtt{ret}\ x \div int}\,\textsc{bind}}{\vdash \mathsf{box}\ St.\ get\ () : [St]\,int}\,\Box I$$

On the other hand, we can try to derive (3) as follows

$$\dfrac{\dfrac{\dfrac{\nvdash \mathit{get}\ () \div \mathit{int}}{\mathit{St} \nvdash \mathsf{box}\ (\mathit{get}\ ()) : [\cdot]\,\mathit{int}}\ \Box I}{\mathit{St} \nvdash \mathtt{ret}\ (\mathsf{box}\ (\mathit{get}\ ())) \div [\cdot]\,\mathit{int}}\ \textsc{ret}}{\nvdash \mathsf{box}\ \mathit{St}.\ \mathtt{ret}\ (\mathsf{box}\ (\mathit{get}\ ())) : [\mathit{St}][\cdot]\,\mathit{int}}\ \Box I$$

but the derivation tree cannot be completed because *get* doesn't appear in the effect context available at the top.

4.1.2 Typing Handlers. Next consider the derivations (4-6). In the handler typing judgment, the declared components $A[\Psi] \overset{S}{\Longrightarrow} B$ should be understood as an input that guides the rest of derivation, and determines the types of the variables bound in operation clauses. For example, let us show why (4) holds. Recall the definition of *handlerSt*.

$$\begin{aligned}\vdash (&\mathit{get}\,(x,k,z) \to \mathtt{cont}\ k\ z\ z,\\ &\mathit{set}\,(x,k,z) \to \mathtt{cont}\ k\ ()\ x,\\ &\mathit{return}\,(x,z) \to \mathtt{ret}\ (x,z)) \div \mathit{int}[\mathit{St}] \overset{\mathit{int}}{\Longrightarrow} \mathit{int} \times \mathit{int}\end{aligned}$$

in the typing of the *return* clause we have to assume $x : \mathit{int}$, as *int* is the declared input type of the handler, and $z : \mathit{int}$, as *int* is the declared type of handler state. The clause returns a pair of type $\mathit{int} \times \mathit{int}$, which matches the result type of the handler, by rule RETH.

Similarly, in the typing of the *get* clause, we must assume

- $x : \mathit{unit}$, as *unit* is the input type of *get* in *St*
- $k \div \mathit{int} \overset{\mathit{int}}{\Longrightarrow} \mathit{int} \times \mathit{int}$. The first *int* is the result type of *get* in *St*. The second *int* is the declared type of handler state. The result type of k matches the declared result type of the handler.
- $z : \mathit{int}$ because *int* is the declared type of handler state.

Under these assumptions, `cont` $k\ z\ z$ has type $\mathit{int} \times \mathit{int}$ by rule CONT, matching the result type of the handler by rule OPH.

4.1.3 Typing Handling Sequences. The starting point for derivation (7) is the judgment (6) of *handlerExplosiveSt*. In sequence (7), we provide 12 as the initial state, thus matching the type *int* of handler state of *handlerExplosiveSt*. We also provide continuation $x.\ \mathtt{ret}\ (\pi_1\ x)$, which modifies the output type $\mathit{int} \times \mathit{int}$ of *handlerExplosiveSt* into the output type *int* of the sequence (7).

This modification of the output type of (7) into *int* is essential to typecheck (8). By the typing rule OPH, the output type of a sequence must match the input type of the extending handler, and in our case, *int* is *handlerExn* input type by derivation (5).

4.1.4 Generic Modal Typing Derivations. Derivation (9) is easy to establish by induction on Ψ. Then derivation (10) shows that we can coerce a computation from theory Ψ into a larger theory Ψ'. In the special case when $\Psi' = \Psi$, the derivation establishes the validity of η-expansion for modal types, i.e., local soundness. We show derivation (10) in some detail below, assuming derivation (9) and weakening in both contexts. We also elide the first steps involving λ, as they are standard.

$$\dfrac{\Delta \vdash e : [\Psi]A \qquad \dfrac{\dfrac{\Delta, u :: A[\Psi]; \Psi' \vdash \mathtt{id}_\Psi \div A[\Psi] \overset{\mathit{unit}}{\Longrightarrow} A \qquad \dfrac{}{\Delta, u :: A[\Psi] \vdash () : \mathit{unit}}}{\Delta, u :: A[\Psi]; \Psi' \vdash \mathtt{handle}\ u\ \mathtt{id}_\Psi\ () \div A}\ \textsc{mvar}}{\Delta, u :: A[\Psi] \vdash \mathsf{box}\ \Psi'.\ \mathtt{handle}\ u\ \mathtt{id}_\Psi\ () : [\Psi']A}\ \Box I}{\Delta \vdash \mathtt{let}\ \mathsf{box}\ u = e\ \mathsf{in}\ \mathsf{box}\ \Psi'.\ \mathtt{handle}\ u\ \mathtt{id}_\Psi\ () : [\Psi']A}\ \Box E$$

Derivations (11-13) show that the modal type $[\Psi]A$ satisfies the usual functions required of monads in programming. Derivation (11) shows that a value of type A can be coerced into a computation of type $[\Psi]A$. Derivation (12) shows that a function application is stable under algebraic theory; that is, if a function and the argument share the algebraic theory, so will the result. This is property K of modal logic. Finally, derivation (13) shows that iterated modalities can be coalesced. This is property $C4$ of modal logic.

5 REDUCTIONS AND SUBSIDIARY OPERATIONS

As in CMTT, β-reduction of modal types is the key notion in ECMTT. This, together with β-reduction on function types, is the basis for the ECMTT operational semantics (Figure 6).

$$\begin{aligned} \mathtt{let\ box}\, u = \mathtt{box}\ \Psi.\ c\ \mathtt{in}\ c' &\mapsto_\beta c'[\Psi.c//u] \\ \mathtt{let\ box}\, u = \mathtt{box}\ \Psi.\ c\ \mathtt{in}\ e &\mapsto_\beta e[\Psi.c//u] \end{aligned}$$

The reduction depends on *modal substitutions* $c'[\Psi.c//u]$ (resp. $e[\Psi.c//u]$), which substitute computation c for a modal variable u in a computation c' (resp. expression e), with Ψ indicating the operators bound in c. We define modal substitution in the Appendix A of the extended version [Zyuzin and Nanevski 2021], and in this section illustrate it on an example, along with a number of subsidiary operations (Figure 4) that modal substitution invokes. These operations are:

- *Monadic substitution* $\{c'/x\}c$ sequentially composes c' before c, using variable x as the connection. The definition follows closely a similar notion from Pfenning and Davies [2001].
- *Continuation substitution* $c[x.y.\ c' \curvearrowleft k]$ replaces the continuation variable k in c with a computation c' that binds variables x and y.
- *Handling* $\langle c \rangle_e^h$ applies the handler h over computation c, using e as the current state.
- *Handling sequencing* $(\!| c \mid \Theta |\!)$ applies the handling sequence Θ to the computation c.
- *Expression substitutions* $c[e/x]$ and $e'[e/x]$ replace an expression e for variable x into computation c and expression e' respectively. The last two are standard notions, so we use them without an explicit definition.

5.1 Example

We illustrate all these notions by tracing the reduction steps of the following example. We present the steps on Figure 5 and we also comment on them below.

$$\mathtt{let\ box}\ u = \mathit{incr}_n\ 1\ \mathtt{in}\ \mathtt{handle}\ u\ \mathit{handlerSt}\ 0$$

In the first step (11), we unfold the definition of incr_n 1 and apply the β-reduction to reveal a modal substitution for the variable u into $(x' \leftarrow \mathtt{handle}\ u\ \mathit{handlerSt}\ 0;\ \mathtt{ret}\ x')$. The latter represents the scope of the initial `let-box` term, expanding the syntactic sugar for statements.

In the next step (12), the modal substitution applies handling with *handlerSt* with given state over the application of the empty handling sequence on u. This proceeds by the following characteristic definition case for modal substitution.

$$(x' \leftarrow \mathtt{handle}\ u\ [\Theta]\ h\ e;\ c')[\Psi.c//u] = \{\langle (\!| c \mid \Theta[\Psi.c//u] |\!) \rangle_{e[\Psi.c//u]}^{h[\Psi.c//u]} / x'\}(c'[\Psi.c//u])$$

In other words, we first substitute c for u in all the subterms. As customary, we ensure that c doesn't contain free occurrences of variable x' (thus incurring capture), by α-renaming x' if necessary. In the expression (11), u doesn't occur in the subterms, thus the modal substitutions are vacuous, and we directly obtain (12).

In the next step (13), the empty handling sequencing immediately resolves according to the definition in Figure 4d, and proceeds to handling with *handlerSt*.

$$\begin{aligned}
\{\mathtt{ret}\ e/x\}c &= c[e/x]\\
\{y \leftarrow s;\ c'/x\}c &= y \leftarrow s;\ \{c'/x\}c\\
\{\mathtt{let\ box}\ u = e\ \mathtt{in}\ c'/x\}c &= \mathtt{let\ box}\ u = e\ \mathtt{in}\ \{c'/x\}c
\end{aligned}$$

(a) Monadic substitution.

$$\begin{aligned}
(\mathtt{ret}\ e)[x.y.\ c \wr k] &= \mathtt{ret}\ e\\
(x' \leftarrow op\ e;\ c')[x.y.\ c \wr k] &= x' \leftarrow op\ e;\ c'[x.y.\ c \wr k]\\
(x' \leftarrow \mathtt{cont}\ k\ e_1\ e_2;\ c')[x.y.\ c \wr k] &= \{c[e_1/x][e_2/y]/x'\}c'\\
(x' \leftarrow \mathtt{cont}\ k'\ e_1\ e_2;\ c')[x.y.\ c \wr k] &= x' \leftarrow \mathtt{cont}\ k'\ e_1\ e_2;\ c'[x.y.\ c \wr k] \qquad \text{when } k \neq k'\\
(\mathtt{let\ box}\ u = e\ \mathtt{in}\ c')[x.y.\ c \wr k] &= \mathtt{let\ box}\ u = e\ \mathtt{in}\ c'[x.y.\ c \wr k]\\
(x' \leftarrow \mathtt{handle}\ u\ [\Theta]\ h\ e;\ c')[x.y.\ c \wr k] &= x' \leftarrow \mathtt{handle}\ u\ [\Theta]\ h[x.y.\ c \wr k]\ e;\ c'[x.y.\ c \wr k]
\end{aligned}$$

$$\begin{aligned}
(\mathit{return}\ (x', z) \to c')[x.y.\ c \wr k] &= \mathit{return}\ (x', z) \to c'[x.y.\ c \wr k]\\
(h, op\ (x', k', z) \to c')[x.y.\ c \wr k] &= h[x.y.\ c \wr k], op\ (x', k', z) \to c'[x.y.\ c \wr k]
\end{aligned}$$

(b) Continuation substitution.

$$\begin{aligned}
\langle \mathtt{ret}\ e' \rangle_e^h &= c[e'/x][e/z] \quad \text{where } \mathit{return}\ (x, z) \to c \in h\\
\langle y \leftarrow op\ e';\ c' \rangle_e^h &= c_{op}[e/z][e'/x'][y.z'.\ \langle c' \rangle_{z'}^h \wr k] \quad \text{where } (op\ (x', k, z) \to c_{op}) \in h\\
\langle \mathtt{let\ box}\ u = e'\ \mathtt{in}\ c' \rangle_e^h &= \mathtt{let\ box}\ u = e'\ \mathtt{in}\ \langle c' \rangle_e^h\\
\langle x' \leftarrow \mathtt{handle}\ u\ [\Theta]\ h'\ e';\ c' \rangle_e^h &= x \leftarrow \mathtt{handle}\ u\ [\Theta, (h', e', x'.\ c')]\ h\ e;\ \mathtt{ret}\ x
\end{aligned}$$

(c) Handling.

$$\begin{aligned}
(\!| c' \mid \bullet |\!) &= c'\\
(\!| c' \mid \Theta, (h, e, x.\ c) |\!) &= \{\langle (\!| c' \mid \Theta |\!) \rangle_e^h / x\}c
\end{aligned}$$

(d) Handling sequencing.

Fig. 4. Definition of operations auxiliary to modal substitution. The definitions are recursive but well-founded (hence, also terminating) as each operation either makes recursive calls involving strictly smaller subterms, or invokes an operation that has been defined ahead of it in the figure. As conventional, we assume that all the bound variables are α-renamed to avoid capture by the operations.

Handling by *handlerSt* in (13) results in the computation $\mathtt{ret}\ (0, 1)$, as we shall illustrate shortly. In the next step (14), this computation is monadically substituted for x' in $\mathtt{ret}\ x'$ to get the final reduction result of $\mathtt{ret}\ (0, 1)$. This follows by the definition of the monadic substitution in Figure 4a. Monadic substitution $\{c'/x\}c$ sequentially precomposes c' before c. In the special case when c' is $\mathtt{ret}\ e$, it simply substitutes e for x in c.

In step (15) we return to the step (13) of applying the handler *handlerSt*. We proceed according to the definition of handling from Figure 4c. In particular, we obtain the expression (16) by substituting the following into the get clause of *handlerSt*:

- The supplied initial state 0 is substituted for the state variable z.

$$\begin{aligned}
&\mathtt{let\ box}\ u = \mathit{incr}_n\ 1\ \mathtt{in}\ \mathtt{handle}\ u\ \mathit{handlerSt}\ 0 \ \mapsto_\beta \\
&\qquad (x' \leftarrow \mathtt{handle}\ u\ \mathit{handlerSt}\ 0;\ \mathtt{ret}\ x')[\mathit{St}.y \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y /\!/ u] && (11)\\
&\quad = \{\langle (\!|\, y \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y \mid \bullet |\!) \rangle_0^{\mathit{handlerSt}} / x'\}\mathtt{ret}\ x' && (12)\\
&\quad = \{\langle y \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y \rangle_0^{\mathit{handlerSt}} / x'\}\mathtt{ret}\ x' && (13)\\
&\quad = \{\mathtt{ret}\ (0,1)/x'\}\mathtt{ret}\ x' = \mathtt{ret}\ (0,1) && (14)
\end{aligned}$$

$$\begin{aligned}
&\langle y \leftarrow \mathit{get}\ ();\ \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y \rangle_0^{\mathit{handlerSt}} && (15)\\
&\quad = (x' \leftarrow \mathtt{cont}\ k\ z\ z;\ \mathtt{ret}\ x')[0/z][()/x][y.z'.\ \langle \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y \rangle_{z'}^{\mathit{handlerSt}} \hookleftarrow k] && (16)\\
&\quad = (x' \leftarrow \mathtt{cont}\ k\ 0\ 0;\ \mathtt{ret}\ x')[y.z'.\ \mathtt{ret}\ (y, y+1) \hookleftarrow k] && (17)\\
&\quad = \{(\mathtt{ret}\ (y, y+1))[0/y][0/z']/x'\}\mathtt{ret}\ x' = \{(\mathtt{ret}\ (0, 0+1))/x'\}\mathtt{ret}\ x' && (18)\\
&\quad = (\mathtt{ret}\ x')[(0, 0+1)/x'] = \mathtt{ret}\ (0, 0+1) = \mathtt{ret}\ (0,1) && (19)
\end{aligned}$$

$$\begin{aligned}
&\langle \_ \leftarrow \mathit{set}\ (y+1);\ \mathtt{ret}\ y \rangle_{z'}^{\mathit{handlerSt}} && (20)\\
&\quad = (x' \leftarrow \mathtt{cont}\ k\ ()\ x;\ \mathtt{ret}\ x')[z'/z][(y+1)/x][\_.z''.\ \langle \mathtt{ret}\ y \rangle_{z''}^{\mathit{handlerSt}} \hookleftarrow k] && (21)\\
&\quad = (x' \leftarrow \mathtt{cont}\ k\ ()\ (y+1);\ \mathtt{ret}\ x')[\_.z''.\ \mathtt{ret}\ (y, z'') \hookleftarrow k] && (22)\\
&\quad = \{(\mathtt{ret}\ (y, z''))[(y+1)/z'']/x'\}\mathtt{ret}\ x' = \{(\mathtt{ret}\ (y, y+1))/x'\}\mathtt{ret}\ x' && (23)\\
&\quad = (\mathtt{ret}\ x')[(y, y+1)/x'] = \mathtt{ret}\ (y, y+1) && (24)
\end{aligned}$$

$$\langle \mathtt{ret}\ y \rangle_{z''}^{\mathit{handlerSt}} = (\mathtt{ret}\ (x,z))[y/x][z''/z] = \mathtt{ret}\ (y, z'') \qquad (25)$$

Fig. 5. Process of reduction for $\mathtt{let\ box}\ u = \mathit{incr}_n\ 1\ \mathtt{in}\ \mathtt{handle}\ u\ \mathit{handlerSt}\ 0$.

- The input argument () of *get* is substituted for the variable x in the handler clause for *get*.
- The *handled* remainder of the initial computation is substituted for the continuation variable k, according to continuation substitution from Figure 4b.

The last component above, namely the handling of the remainder of the computation is depicted in lines (20-25) in Figure 5. We do not comment on it in any detail as it is similar to the rest of the example, but we just note that it obtains the result $\mathtt{ret}\ (y, y+1)$ in the scope of variables y and z'. Thus, it illustrates that all of our subsidiary operations work over expressions with free variables.

We next execute the substitutions over z and x to derive step (17). Next, to obtain (18), we proceed with continuation substitution whereby the arguments 0 and 0 of k in the main expression are replaced for y and z' respectively in the expression being substituted for k. The remainder of the reduction is then easy to complete.

6 SOUNDNESS

In this section we present the basic theoretical properties of ECMTT, leading to the proofs of local soundness (i.e., that β-reduction is well typed), local completeness (i.e., that η-expansion is well typed), as well as the type soundness theorems (i.e., progress and preservation) for the operational semantics from Figure 6. The bulk of the development involves lemmas about the subsidiary operations from Section 5. These lemmas all have the form reminiscent of substitution principles, as

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_2 \mapsto e_2'}{v\ e_2 \mapsto v\ e_2'} \qquad \frac{}{(\lambda x{:}A.\ e)\ v \mapsto e[v/x]}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let box}\ u = e_1\ \texttt{in}\ e_2 \mapsto \texttt{let box}\ u = e_1'\ \texttt{in}\ e_2} \qquad \frac{}{\texttt{let box}\ u = \texttt{box}\ \Psi.\ c\ \texttt{in}\ e \mapsto e[\Psi.c//u]}$$

$$\frac{e \mapsto e'}{\texttt{ret}\ e \mapsto \texttt{ret}\ e'}$$

$$\frac{e \mapsto e'}{\texttt{let box}\ u = e\ \texttt{in}\ c \mapsto \texttt{let box}\ u = e'\ \texttt{in}\ c} \qquad \frac{}{\texttt{let box}\ u = \texttt{box}\ \Psi.\ c_1\ \texttt{in}\ c_2 \mapsto c_2[\Psi.c_1//u]}$$

Fig. 6. Call-by-value operational semantics. The semantics considers closed terms (no free variables) in the empty effect theory (no unhandled operations). Consequently, it only evaluates computations of the form ret e and let box $u = e$ in c, as other cases involve free variables or operations. The rules include only those for β-reduction, and for enforcing call-by-value/left-to-right evaluation order. The values of expression kind (ranged over by v) are $\lambda e : A.\ e'$ and box $\Psi.\ c$. The value of computation kind is ret v.

they describe how a variable or an effect operation can be replaced in a term. Appendix B of the extended version [Zyuzin and Nanevski 2021] contains a more detailed presentation of the proofs.

6.1 Structural Properties

To avoid writing out all 20 combinations of our variable and term judgements, we state weakening with generic binding forms. We use $x_\Delta : J_{var}$ to stand over variable bindings in Δ, namely $x : A$ and $u :: A[\Psi]$. $x_\Gamma : J_{var}$ ranges over Γ variable bindings $op \div A \Rightarrow B$ and $k \dotdiv A \overset{B}{\Longrightarrow} C$. The generic term judgement $\Delta; \Gamma \vdash t : J_{term}$ ranges over the judgements for $e : A$ (in this case we implicitly ignore Γ), $c \div A$, $s \div_{\mathsf{s}} A$, $h \div A[\Psi] \overset{S}{\Longrightarrow} B$, and $\Theta \div A[\Psi] \Rrightarrow B$. We also use the generic term judgement to state the principles governing the subsidiary operations from Section 5.

Lemma 6.1 (Weakening). *If* $\Delta; \Gamma \vdash t : J_{term}$, *then*

(1) $\Delta, x_\Delta : J_{var}; \Gamma \vdash t : J_{term}$.
(2) $\Delta; \Gamma, x_\Gamma : J_{var} \vdash t : J_{term}$.

Lemma 6.2 (Expression substitution principle). *If* $\Delta \vdash e : A$ *and* $\Delta, x : A; \Gamma \vdash t : J_{term}$, *then* $\Delta; \Gamma \vdash t[e/x] : J_{term}$.

Lemma 6.3 (Monadic substitution principle). *If* $\Delta; \Gamma \vdash c \div A$ *and* $\Delta, x : A; \Gamma \vdash c' \div B$, *then* $\Delta; \Gamma \vdash \{c/x\}c' \div B$.

Lemma 6.4 (Continuation substitution principle). *If* $\Delta, x : A_1, y : A_2; \Gamma \vdash c' \div A_3$, *then*

(1) *If* $\Delta; \Gamma, k \dotdiv A_1 \overset{A_2}{\Longrightarrow} A_3 \vdash c \div B$, *then* $\Delta; \Gamma \vdash c[x.y.\ c' \rightsquigarrow k] \div B$.

(2) *If* $\Delta; \Gamma, k \dotdiv A_1 \overset{A_2}{\Longrightarrow} A_3 \vdash h \div C[\Psi] \overset{S}{\Longrightarrow} C'$, *then* $\Delta; \Gamma \vdash h[x.y.\ c' \rightsquigarrow k] \div C[\Psi] \overset{S}{\Longrightarrow} C'$.

Lemma 6.5 (Handling principle). *If* $\Delta; \Psi \vdash c \div A$, *and* $\Delta; \Gamma \vdash h \div A[\Psi] \overset{B}{\Longrightarrow} C$, *and* $\Delta \vdash e : B$, *then* $\Delta; \Gamma \vdash \langle c \rangle_e^h \div C$.

Lemma 6.6 (Handling sequencing principle). *If* $\Delta;\Psi \vdash c \div A$ *and* $\Delta;\Psi' \vdash \Theta \div A[\Psi] \Rrightarrow B$, *then* $\Delta;\Psi' \vdash (\!| c \mid \Theta |\!) \div B$.

Lemma 6.7 (Modal substitution principle). *If* $\Delta;\Psi \vdash c \div A$ *and* $\Delta, u :: A[\Psi];\Gamma \vdash t : J_{term}$, *then* $\Delta;\Gamma \vdash t[\Psi.c//u] : J_{term}$.

Lemma 6.8 (Identity handler). $\Delta;\Psi \vdash id_\Psi \div A[\Psi] \xRightarrow{unit} A$.

6.2 Main Theorems

Theorem 6.9 (Local soundness). *If* $\Delta;\Psi \vdash c \div A$, *then the following β-reductions are well typed.*

(1) *If* $\Delta;\Gamma \vdash \texttt{let box}\ u = \texttt{box}\ \Psi.\ c\ \texttt{in}\ c' \div B$, *then* $\Delta;\Gamma \vdash c'[\Psi.c//u] \div B$.
(2) *If* $\Delta \vdash \texttt{let box}\ u = \texttt{box}\ \Psi.\ c\ \texttt{in}\ e' : B$, *then* $\Delta \vdash e'[\Psi.c//u] : B$.

Theorem 6.10 (Local completeness). *If* $\Delta \vdash e : [\Psi]A$, *then the η-expansion of e is well-typed, i.e.* $\Delta \vdash \texttt{let box}\ u = e\ \texttt{in box}\ \Psi.\ \texttt{handle}\ u\ id_\Psi\ () : [\Psi]A$.

Theorem 6.11 (Preservation on expressions). *If* $\vdash e : A$ *and* $e \mapsto e'$, *then* $\vdash e' : A$.

Theorem 6.12 (Preservation on computations). *If* $\vdash c \div A$ *and* $c \mapsto c'$, *then* $\vdash c' : A$.

Theorem 6.13 (Progress on expressions). *If* $\vdash e : A$, *then either (1) e is a value, or (2) there exists* e' *s.t.* $e \mapsto e'$.

Theorem 6.14 (Progress on computations). *If* $\vdash c \div A$, *then either (1) c is* $\texttt{ret}\ v$ *where v is a value, or (2) there exists* c' *s.t.* $c \mapsto c'$.

The above theorems directly imply *effect safety*: programs with no effects (empty context Γ), can't incur unhandled effect operation during execution. Indeed, preservation implies that execution, which is always attempted on terms with no effect operations, can't reach a computation that will use one. In turn, progress implies that a computation with no operations in Γ can always make a step; in particular, it can't get stuck on an unhandled operation.

7 EXTENSIONS TO ECMTT

7.1 Evaluation

In any calculus that tracks effects in types, one wants to relate the category of computations with no effects to the purely functional expressions. Similarly, in ECMTT we also would like to have a correspondence between the types A and $\Box A$. In Section 4 we presented a function (monadic unit) of type $A \to \Box A$ that realizes one side of the correspondence. To establish the other, we need a function of type $\Box A \to A$, known in modal logic as axiom T or reflexivity [Blackburn et al. 2001], as counit for the □ comonad in categorical semantics [Bierman and de Paiva 2000], and as the *eval* function in modal type systems [Davies and Pfenning 2001].

However, currently it's impossible to express in ECMTT, because modal types can only be handled within the computation judgement. That is, we can always handle $\Box A$ into A, but only within the scope of a box term. Thus, we can write a function for $\Box A \to \Box A$

$$\lambda x : \Box A.\ \texttt{box let box}\ u = x\ \texttt{in handle}\ u\ id.\ () : \Box A \to \Box A$$

but not for $\Box A \to A$. To support the latter, we need a way to convert computations in an empty theory into expressions. Thus we present an extension of ECMTT with a primitive `eval` expression.

We extend the language with a new expression $\texttt{eval}\ [\Theta]\ u$ that, similarly to handling, holds a handling sequence Θ applied to u (our extensions are summarized on Figure 7). For `eval`, the typing rule restricts Θ to typecheck in an empty effect context which ensures that that the last

$$
\begin{aligned}
e &::= \cdots \mid \texttt{eval}\ [\Theta]\ u \mid \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c\ \mathsf{in}\ e \\
c &::= \cdots \mid \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c_1\ \mathsf{in}\ c_2
\end{aligned}
$$

(a) Syntax

$$\frac{(u :: A[\Psi]) \in \Delta \qquad \Delta; \cdot \vdash \Theta \div A[\Psi] \Rrightarrow B}{\Delta \vdash \texttt{eval}\ [\Theta]\ u : B}\ \textsc{eval}$$

$$\frac{\Delta, f : A \to [\Psi]B, x : A; \Psi \vdash c \div B \qquad \Delta, f : A \to [\Psi]B \vdash e : C}{\Delta \vdash \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c\ \mathsf{in}\ e : C}\ \textsc{fix}$$

$$\frac{\Delta, f : A \to [\Psi]B, x : A; \Psi \vdash c_1 \div B \qquad \Delta, f : A \to [\Psi]B; \Gamma \vdash c_2 \div C}{\Delta; \Gamma \vdash \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c_1\ \mathsf{in}\ c_2 \div C}\ \textsc{fix-comp}$$

(b) Typing

$$
\begin{aligned}
(\texttt{eval}\ [\Theta]\ u)[\Psi.c//u] &= [\![(\!| c \mid \Theta[\Psi.c//u] |\!)]\!] \\
[\![\texttt{ret}\ e]\!] &= e \\
[\![x \leftarrow \texttt{handle}\ u\ [\Theta]\ h\ e;\ c]\!] &= \texttt{eval}\ [\Theta, (h, e, x.\ c)]\ u \\
[\![\texttt{let box}\ u = e\ \mathsf{in}\ c]\!] &= \texttt{let box}\ u = e\ \mathsf{in}\ [\![c]\!]
\end{aligned}
$$

(c) Modal substitution and evaluation definition

$$\frac{}{\texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c\ \mathsf{in}\ e \mapsto e[\lambda x : A.\ \mathsf{box}\ \Psi.\ \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c\ \mathsf{in}\ c/f]}$$

$$\frac{}{\texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c_1\ \mathsf{in}\ c_2 \mapsto c_2[\lambda x : A.\ \mathsf{box}\ \Psi.\ \texttt{let fix}\ f\ (x:A) = \mathsf{box}\ \Psi.\ c_1\ \mathsf{in}\ c_1/f]}$$

(d) Operational semantics for `let-fix`

Fig. 7. Extension of ECMTT with `eval` and `let-fix`.

clause of Θ will not use any operation. Thus the computation produced after applying this handling sequence will also be operation-free. This allows us to define the evaluation on empty effect context computations and we additionally extend the modal substitution with the corresponding clause.

We establish the soundness of this addition by proving the following lemma:

Lemma 7.1 (Eval principle). *If* $\Delta; \cdot \vdash c \div A$, *then* $\Delta \vdash [\![c]\!] : A$.

Finally, this extension allows us to define *eval* function in ECMTT and so obtain the other side of the correspondence between A and $\Box A$:

$$eval_f \ \widehat{=}\ \lambda x : \Box A.\ \texttt{let box}\ u = x\ \mathsf{in}\ \texttt{eval}\ [\bullet]\ u : \Box A \to A$$

The addition of *eval* is relatively cheap and doesn't affect any of the definitions from Figure 4; the modal substitution is the only definition that we change. Accordingly, the changes to the soundness proofs are also minimal. The modal substitution principle for *eval* readily follows from Lemmas 7.1 and 6.6. And the progress and preservation theorems do not need to consider this case explicitly because `eval` expression is not a closed term.

7.2 Fixed Points

Currently, there is no way to write loops in ECMTT. In that it's more similar to a system for logical inference than a programming language. We can however add loops through recursion to ECMTT. We introduce a fixed point operator `let-fix` and update operational semantics accordingly (changes are summarized in Figure 7). The idea is that a recursive function f should have the type $A \longrightarrow [\Psi]B$, allowing it to use effect operations from Ψ. As customary, the reduction in operational semantics substitutes f with its definition.

Thus `let-fix` allows us to write programs that use recursion, for example a program that computes 3! and reduces to 6 as we expect:

$$\begin{array}{l} \texttt{let fix}\ \mathit{fact}\ (n : \mathit{int}) = \texttt{box}\ \cdot . \\ \quad \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\mathit{eval}_f\ (\mathit{fact}\ (n-1)))\ \texttt{in} \\ \mathit{eval}_f\ (\mathit{fact}\ 3) \end{array}$$

In Appendix A of the extended version [Zyuzin and Nanevski 2021] we present the definitions for all our subsidiary operations extended with `eval` and `let-fix`, and we also include these cases in the proofs in Appendix B of the extended version.

8 RELATED AND FUTURE WORK

Type-and-effect Systems for Algebraic Effects. Many languages for algebraic effects and handlers include effect systems to track operations. Among them are λ_{eff} [Kammar et al. 2013], which puts collection of available operations E in the typing judgement for computations $\Gamma \vdash_E M : C$, similarly as we do with effect contexts, and Eff [Bauer and Pretnar 2014, 2015; Karachalias et al. 2020], Links [Hillerström and Lindley 2016, 2018; Hillerström et al. 2020], Helium [Biernacki et al. 2019], Frank [Convent et al. 2020], Koka [Leijen 2014], and Effekt [Brachthäuser et al. 2020] which put the annotation on the computation's type.

In Eff, the type $A!\Delta$ specifies that a computation might invoke operations from Δ, which is a collection of operations. Links, Helium, and Koka rely on a more involved effect annotation called effect rows, although their primary role—to track the use of operations—is similar to the one in Eff. Links for example similarly specifies their computation type as $A!E$ but there E is an effect row of a computation. Notably, effect row languages support a notion of row polymorphism, that allows a programmer to write computations polymorphic over effect theories.

Effekt and the system of Zhang and Myers [2019], also rely on a capability-passing style for tracking effects. In this style, effects are capabilities that handlers provide to the code within their handling scope. In contrast, the operations in ECMTT are *variables*, which are bound by `box`, and handled by the handler associated with the modal variable in the corresponding `let-box`. This distinction should be important for a future extension of ECMTT with effect polymorphism, as we discuss below.

ECMTT further differs from all the above languages in the key property that it manipulates effect annotations explicitly through introduction and elimination forms of the modal type $[\Psi]A$. Thus, the ECMTT type-and-effect system is in fact just a type system, as the effect annotations are built directly into the modal types. This leads to an operational semantics determined solely by β-reduction, although β-reduction itself now includes handling. More importantly, we can consider the modal (effectful) types as propositions, thus, in the future, obtaining a Curry-Howard style interpretation for algebraic effects and handlers.

On the other hand, the above languages support more advanced features in handlers than ECMTT, for example shallow [Hillerström and Lindley 2018] and recursive [Bauer and Pretnar

2015] handlers, or asynchronous effects, as in $\lambda_{æ}$ [Ahman and Pretnar 2021]. We leave considering these features for ECMTT for future work.

Scaling to Dependent Types. Ahman [2017] develops a dependently typed calculus for algebraic effects based on call-by-push value approach, called EMLTT. In this paper we don't consider dependent types. However, the foundation of ECMTT in modal logic should offer some conceptual simplification when scaling to dependent types, as we can build on the recent work on modal type theories. For example, the modal foundation led us to define handling as part of β-reduction on open terms. In EMLTT, handling also works over open terms, but is governed by several definitional equalities which describe how handling reduces. In contrast, ECMTT requires only a single β-reduction of `let-box` which encompasses handling.

CMTT itself has been lifted to dependent types [Nanevski et al. 2008], albeit in a somewhat restricted form, with abstraction over modal variables, but no explicit modality. More recently, Gratzer et al. [2019] have developed a variant of Martin-Löf type theory with a non-contextual $\Box$ modality, while using an alternative elimination form `unbox` [Davies and Pfenning 2001] instead of `let-box` to avoid commuting conversions. We might consider a similar approach in future work.

Currently, ECMTT admits only free algebraic theories; i.e. those with no equations between operations. Luksic and Pretnar [2020] add support for equations in Eff through the extension of the annotations to $A!\Delta/\mathcal{E}$, where $\mathcal{E}$ stands for the set of equations. Ahman [2017] supports equations and also treats them as a special annotation over the computation types. We expect that the scaling to dependent types will immediately allow us to support equations, or any arbitrary propositions over operations. Indeed, in dependent type theories, propositions are merely types, and thus can easily be added to a context Ψ representing an algebraic theory in a modal type.

Effect Polymorphism and Abstraction Safety. The work related to Beluga [Cave and Pientka 2013; Pientka 2008, 2010] has extended CMTT, among other features, with abstraction over first-class contexts and explicit substitutions. In the future, we plan to incorporate similar extensions to ECMTT and apply them to algebraic effects. These should provide new and effective solutions to the problem of unintended capture of operations in the presence of effect polymorphism.

To describe the problem, suppose we have a handler that handles only a single operation *op*. We want to apply this handler to an effect polymorphic computation that has an operation signature $[E, op]$, where E is a variable that stands for some set of effects. After handling, E is supposed to remain in the signature of the resulting computation, while *op* is to be stripped away by handling. The problem of unintended capture is to ensure that E remains in the signature after handling, even if we instantiate E with *op* itself.

Current solutions in the field of algebraic effects rely on a coupling of operations with their intended handlers through lexical scoping [Biernacki et al. 2020; Brachthäuser et al. 2020; Zhang and Myers 2019]. They are implemented by making handlers emit a capability, or a label, that is then passed to the code in the handler's scope and is attached to each operation call so that handlers can dynamically distinguish the operations they know of from others.

In the field of contextual modal type theory, the problem of unintended capture of variables in the presence of context polymorphism was already considered by Pientka [2008] in Beluga. Suppose we extended ECMTT following this approach, and let us sketch how that problem would look in the extended system where we can bind contexts, and what solution we plan to borrow:

$$\begin{array}{l}\lambda\psi : ctx.\ \lambda f : unit \to [\psi, op \div int \Rightarrow int]\, int.\ \texttt{box}\ \psi.\\ \quad \texttt{let box}\ u = f\ ()\ \texttt{in}\\ \quad \texttt{handle}\ u\ (id_{\psi}, op\,(x, k, z) \to \texttt{cont}\ k\ (x + 1)\ z)\ ()\end{array}$$

The example program above binds some concrete context to ψ, takes a function f that produces a computation in the theory (ψ, op), then runs f, and handles only op, propagating any eventual operations from ψ through the identity handler id_{ψ}. The problem of unintended capture appears if we consider instantiating ψ with some context that itself contains $op \div int \Rightarrow int$, as this operation will now be handled rather than propagated, contrary to the intended semantics.

One can see that the example is problematic already from the types, as the substitution of ψ causes the problem in the context of f, which now contains two instances of op, one shadowing the other. To avoid the shadowing, and thus also the unintended capture, Beluga locally α-renames the variables in the context instantiating ψ. As this context stands for the (unknown) bound variables in the type and body of f, α-renaming them doesn't change the semantics. The renaming occurs in the type of f but also in the index of the identity handler id_{ψ}. Thus, once α-renamed, the occurrence of op arising out of ψ is given to the identity handler for propagation, as intended, rather than being captured by the op clause in the handler for u. We will follow this approach in extending ECMTT with effect polymorphism. We will also consider how one efficiently finds the nearest handler in a handling sequence that treats a given effect in a non-trivial way (i.e., not by simply propagating it via identity handler) [Schuster et al. 2020; Xie et al. 2020].

Relationship to Comonads and Modal Logic and Calculi. As illustrated in Section 5, our formulation of ECMTT employs the `let box` $u = e_1$ `in` c_2 constructor. When e_1 itself is a value—thus, by typing, necessarily of the form box $\Psi.\ c_1$—the reduction modally substitutes c_1 for u in c_2. However, whether c_1 evaluates depends on the occurrences of u in c_2. This is in contrast to the elimination rule for monads (i.e., the monadic bind), where the bound computation immediately executes.

This kind of reduction is characteristic of comonadic calculi. For example, CMTT was given comonadic semantics in [Gabbay and Nanevski 2013]. More generally, comonadic calculi for coeffects [Gaboardi et al. 2016; Orchard et al. 2019] also employ constructs with similar behavior, inspired by the elimination rule for exponentials in linear λ-calculus. However, to the best of our knowledge, these calculi haven't been applied to algebraic effects and handling. On a related note, CMTT has been applied to staged computation and meta programming [Davies and Pfenning 2001; Nanevski and Pfenning 2005], and recently it was proposed that staging could be useful for modular treatment of algebraic effects [Poulsen et al. 2021; Schuster et al. 2020; Wei et al. 2020], albeit also without using modal logic.

Nanevski [2004] presents modal calculi similar to CMTT, where the graded $\Box$ modality tracks effects that depend on the execution environment but don't change it. These are handleable effects; examples include exceptions and delimited continuations [Nanevski 2003b], and dynamic binding [Nanevski 2003a]. In contrast, ECMTT supports algebraic effects whose execution may change the environment (e.g., witness the theory St of state in Section 3), and requires a significantly more general notion of handling. Another distinction is that ECMTT models effects simply as contexts of operations, whereas loc. cit. use much more involved freshness and binding disciplines inspired by Nominal logic [Pitts 2003].

Moreover, as we show in the examples (11-13) in Section 4, the modality $[\Psi]A$ exhibits the functions with the types that are in programming usually associated with monads. Furthermore, the typing $\Box A \to A$ of the comonadic counit becomes available with the addition of the evaluation construct in Section 7. In the future we will study the equational theory of ECMTT, with the goal of clarifying the exact categorical nature of our modality. A useful step in this direction will be the work on categorical semantics of operations in scope [Piróg et al. 2018].

Another fruitful research direction is suggested by Wu et al. [2014] in Haskell. It illustrates that the practical use of algebraic effects requires operations with types which are higher-order, meaning that the operation can be parametrized by a computation, or even a handler. The former is already possible in ECMTT, as our operations can range over modal types. The latter will require internalization of effect handlers. While we have a judgment for handlers, we don't currently have a type for them that can be combined with other types. In CMTT this corresponds to internalizing the judgment for explicit substitutions, which has been done in Beluga by Cave and Pientka [2013].

We also plan to study how graded $\diamond$ modality (called "possibility", or "diamond") can be used in ECMTT. Nanevski [2003a] proposed that the proof terms for $\langle\Psi\rangle$ correspond to installing a default handler for a number of effect names. Afterwards, these effects do not need to be explicitly handled, as the default handler applies. We expect that similar behavior will usefully extend to algebraic effects in ECMTT.

Finally, the type system of ECMTT makes sense as a logic as well, i.e., when one erases the terms and just considers the types as propositions. In the future we plan to study this logic (e.g., its Kripke semantics, its sequent calculus, etc.) and derive correspondence with ECMTT in the style of Curry and Howard.

9 CONCLUSION

We have presented the design of ECMTT, a novel contextual modal calculus for algebraic effects and handlers. We start from the idea that an algebraic theory can be represented as a variable context, and apply the graded modal necessity type $[\Psi]A$ to classify computations of type A that may invoke effects described by the algebraic theory (equivalently, context) Ψ. The notion of handling naturally arises as a way to transform Ψ into another theory. To the best of our knowledge, this is the first calculus that relates algebraic effects to modal types.

ECMTT is organized around β-reduction and η-expansion for its type constructors. Defining these requires developing interesting technical concepts such as identity handlers and modal substitutions that encompass handling. We illustrated the system on a number of examples, and established the basic soundness properties.

ACKNOWLEDGMENTS

We are grateful to Miëtek Bak, Alexander Gryzlov, and Patrick Cousot for their comments and discussions on the earlier presentation of the work. We thank the anonymous reviewers from ICFP'21 PC and our shepherd Brigitte Pientka for their feedback, which significantly helped us to understand and describe our contribution better.

This research was partially supported by the Spanish MICINN projects BOSCO (PGC2018-102210-B-I00) and the European Research Council project Mathador (ERC2016-COG-724464). Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Danel Ahman. 2017. Handling fibred algebraic effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (2017). https://doi.org/10.1145/3158095

Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434305

Natasha Alechina, Michael Mendler, Valeria de Paiva, and Eike Ritter. 2001. Categorical and Kripke semantics for Constructive S4 modal logic. In *International Workshop on Computer Science Logic, (CSL).* 292–307. https://doi.org/10.1007/3-540-44802-0_21

Miëtek Bak. 2017. Introspective Kripke models and normalisation by evaluation for the $\lambda^{\Box}$-calculus. In *Workshop on Intuitionistic Modal Logic and Applications (IMLA).* https://github.com/mietek/imla2017/blob/master/doc/imla2017.pdf

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? *arXiv preprint* (2018). https://arxiv.org/abs/1807.05923v2

Andrej Bauer and Matija Pretnar. 2014. An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

P. N. Benton, G. M. Bierman, and V.C.V de Paiva. 1998. Computational types from a logical perspective. *Journal of Functional Programming* 8, 2 (1998), 177–193. https://doi.org/10.1017/S0956796898002998

Gavin M. Bierman and Valeria de Paiva. 2000. On an Intuitionistic Modal Logic. *Studia Logica* 65, 3 (2000), 383–416. https://doi.org/10.1023/A:1005291931660

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6. https://doi.org/10.1145/3290319

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Patrick Blackburn, Maarten de Rijke, and Yde Venema. 2001. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science, Vol. 53. Cambridge University Press. https://doi.org/10.1017/CBO9781107050884

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428194

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing control in the presence of one-shot continuations. In *Conference on Programming Language Design and Implementation (PLDI)*. 99–107. https://doi.org/10.1145/231379.231395

Andrew Cave and Brigitte Pientka. 2013. First-class substitutions in contextual type theory. In *International Workshop on Logical Frameworks & Meta-languages: Theory & Practice (LFMTP)*. 15–24. https://doi.org/10.1145/2503887.2503889

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Murdoch James Gabbay and Aleksandar Nanevski. 2013. Denotation of contextual modal type theory (CMTT): Syntax and meta-programming. *J. Appl. Log.* 11, 1 (2013), 1–29. https://doi.org/10.1016/j.jal.2012.07.002

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. *Proc. ACM Program. Lang.* 51, ICFP (2016), 476–489. https://doi.org/10.1145/3022670.2951939

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.* 3, ICFP (2019), 107:1–107:29. https://doi.org/10.1145/3341711

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Workshop on Type-Driven Development (TyDe@ICFP)*. 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström and Sam Lindley. 2018. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems (APLAS)*. 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *Journal of Functional Programming* 30 (2020). https://doi.org/10.1017/S0956796820000040

Danko Ilik. 2013. Continuation-passing style models complete for intuitionistic logic. *Annals of Pure and Applied Logic* 164, 6 (2013), 651–662. https://doi.org/10.1016/j.apal.2012.05.003

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. *SIGPLAN Not.* 48, 9 (2013), 145–158. https://doi.org/10.1145/2544174.2500590

Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. 2020. Explicit effect subtyping. *J. Funct. Program.* 30 (2020), e15. https://doi.org/10.1017/S0956796820000131

Daan Leijen. 2014. Koka: programming with row polymorphic effect types. In *Workshop on Mathematically Structured Functional Programming (MSFP@ETAPS)*. 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Symposium on Principles of Programming Languages (POPL)*. 486–499. https://doi.org/10.1145/3093333.3009872

Ziga Luksic and Matija Pretnar. 2020. Local algebraic effect theories. *J. Funct. Program.* 30 (2020), e13. https://doi.org/10.1017/S0956796819000212

Aleksandar Nanevski. 2003a. From dynamic binding to state via modal possibility. In *Symposium on Principles and Practice of Declarative Programming (PPDP)*. 207–218. https://doi.org/10.1145/888251.888271

Aleksandar Nanevski. 2003b. *A modal calculus for effect handling*. Technical Report CMU-CS-03-149. School of Computer Science, Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-149.pdf

Aleksandar Nanevski. 2004. *Functional programming with names and necessity*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/2004/CMU-CS-04-151.pdf

Aleksandar Nanevski and Frank Pfenning. 2005. Staged computation with names and necessity. *Journal of Functional Programming* 15, 6 (2005), 837–891. https://doi.org/10.1017/S095679680500568X

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic* 9, 3 (2008), 23. https://doi.org/10.1145/1352582.1352591

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (2019), 30 pages. https://doi.org/10.1145/3341714

Frank Pfenning. 2009. Lecture notes on harmony. https://www.cs.cmu.edu/~fp/courses/15317-f09/lectures/03-harmony.pdf

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Symposium on Principles of Programming Languages (POPL)*. 371–382. https://doi.org/10.1145/1328438.1328483

Brigitte Pientka. 2010. Beluga: programming with dependent types, contextual data, and contexts. In *International Symposium on Functional and Logic Programming (FLOPS)*. 1–12. https://doi.org/10.1007/978-3-642-12251-4_1

Brigitte Pientka and Frank Pfenning. 2003. Optimizing higher-order pattern unification. In *International Conference on Automated Deduction (CADE)*. 473–487. https://doi.org/10.1007/978-3-540-45085-6_40

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and semantics for operations with scopes. In *Symposium on Logic in Computer Science (LICS)*. 809–818. https://doi.org/10.1145/3209108.3209166

Andrew M. Pitts. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation* 186, 2 (2003), 165–193. https://doi.org/10.1016/S0890-5401(03)00138-X

Gordon Plotkin and John Power. 2002. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and John Power. 2001. Adequacy for algebraic effects. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 1–24. https://doi.org/10.1007/11548133_3

Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. 2021. Staged effects and handlers for modular languages with abstraction. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*. http://casperbp.net/store/staged-effects-and-handlers.pdf

Matija Pretnar. 2015. An introduction to algebraic effects and handlers (invited paper). In *Conference on the Mathematical Foundations of Programming Semantics (MFPS)*. 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Matija Pretnar and Gordon D Plotkin. 2013. Handling algebraic effects. *Logical Methods in Computer Science* 9 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. ACM Program. Lang.* 4, ICFP (2020), 93:1–93:28. https://doi.org/10.1145/3408975

Alex K. Simpson. 1994. *The proof theory and semantics of intuitionistic modal logic*. Ph.D. Dissertation. University of Edinburgh. http://hdl.handle.net/1842/407

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454039

Antonis Stampoulis and Zhong Shao. 2010. VeriML: typed computation of logical terms inside a language with effects. In *International Conference on Functional Programming (ICFP)*. 333–344. https://doi.org/10.1145/1863543.1863591

Antonios Michael Stampoulis. 2013. *VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants*. Ph.D. Dissertation. Yale University. http://flint.cs.yale.edu/flint/publications/ams-thesis.pdf

Guannan Wei, Oliver Bracevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428232

Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. 1998. Modal types as staging specifications for run-time code generation. *Comput. Surveys* 30, 3es (1998). https://doi.org/10.1145/289121.289129

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Symposium on Haskell*. 1–12. https://doi.org/10.1145/2633357.2633358

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. 4, ICFP, Article 99 (2020). https://doi.org/10.1145/3408981

Yizhou Zhang and Andrew C Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290318

Nikita Zyuzin and Aleksandar Nanevski. 2021. Contextual modal types for algebraic effects and handlers. arXiv:2103.02976 [cs.PL] https://arxiv.org/abs/2103.02976