Reasoning about Effect Interaction by Fusion

ZHIXUAN YANG, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

Effect handlers can be composed by applying them sequentially, each handling some operations and leaving other operations uninterpreted in the syntax tree. However, the semantics of composed handlers can be subtle—it is well known that different orders of composing handlers can lead to drastically different semantics. Determining the desired order of composition is a difficult task.

To alleviate this problem, this paper presents a systematic way of deriving sufficient conditions on handlers for their composite to correctly handle combinations, such as the sum and the tensor, of the effect theories separately handled. These conditions are solely characterised by the clauses for relevant operations of the handlers, and are derived by fusing two handlers into one by a form of fold/build fusion and continuation-passing style transformation.

As case studies, the technique is applied to commutative and distributive interaction of handlers to obtain a series of results about the interaction of common handlers: (a) equations respected by each handler are preserved after handler composition; (b) handling mutable state *before* any handler gives rise to a semantics in which state operations are commutative with any operations from the latter handler; (c) handling the writer effect and mutable state in either order gives rise to a correct handler of the commutative combination of these two theories.

ACM Reference Format:
Zhixuan Yang and Nicolas Wu. 2021. Reasoning about Effect Interaction by Fusion. *Proc. ACM Program. Lang.* 1, ICFP, Article 11 (January 2021), 46 pages.

11

1 INTRODUCTION

Algebraic effects [Plotkin and Power 2002] and their handlers [Plotkin and Pretnar 2009, 2013] are inherently a modular approach to modelling computational effects: algebraic theories of effects *specify* effects and handlers *implement* them. Furthermore, both algebraic theories and handlers are composable in their own right. Algebraic theories can be combined in various ways of specifying the interaction of operations of the sub-theories [Hyland et al. 2006], such as requiring operations from one sub-theory to be commutative with any operation from other theories, giving rise to the combined theory called the *tensor* of the sub-theories. The modularity of effect theories enables programmers to reason about programs involving complex computational effects in a modular way [Gibbons and Hinze 2011]. On the implementation side, effect handlers are composable by running them sequentially, each handling a set of operations in the computation and forwarding other operations.

However, the link between the composability on the specification side (effect theories) and the composability on the implementation side (handlers) has remained elusive. Suppose that two effect theories are combined into a bigger theory by specifying a particular way for their operations to interact. Following the modular methodology of algebraic effects, we would like to handle the combined theory by composing handlers of the sub-theories. However, it is *not* the case that the sequential composition of any handlers of the sub-theories automatically respects the specified interaction. Instead, additional work must be done to prove that the composite handler indeed validates the combined theory. Our goal is to minimise this additional work.

Authors' addresses: Zhixuan Yang, s.yang20@imperial.ac.uk, Department of Computing, Imperial College London, United Kingdom; Nicolas Wu, n.wu@imperial.ac.uk, Department of Computing, Imperial College London, United Kingdom.

2021. 2475-1421/2021/1-ART11 $15.00
https://doi.org/

$$
\begin{array}{l}
h_{ST} = \texttt{handler}\ \{ \\
\quad \texttt{val}\ x \quad \mapsto \texttt{val}\ (\texttt{fun}\ s \mapsto \texttt{val}\ (x, s)), \\
\quad \mathit{get}\ ()\ k \mapsto \texttt{val}\ (\texttt{fun}\ s \mapsto k\ s\ s), \\
\quad \mathit{put}\ s'\ k \mapsto \texttt{val}\ (\texttt{fun}\ s \mapsto k\ ()\ s')\}
\end{array}
\qquad
\begin{array}{l}
h_{ND} = \texttt{handler}\ \{ \\
\quad \texttt{val}\ x \quad \mapsto \texttt{val}\ [x], \\
\quad \mathit{coin}\ ()\ k \mapsto \{\texttt{let}\ x = k\ \mathit{True}\ \texttt{in} \\
\qquad\qquad\quad \texttt{let}\ y = k\ \mathit{False}\ \texttt{in} \\
\qquad\qquad\quad \texttt{val}\ (x + y)\}\}
\end{array}
$$

Fig. 1. Handlers of mutable state and nondeterminism in the language Eff [Bauer and Pretnar 2015].

To illustrate this problem, we use a running example of the theories of mutable state and nondeterminism. The theory *State* of mutable state consists of *operations*, such as *get* and *put* for reading and writing the state, and *equations* that give the properties these two operations obey (listed in full in Example 2.4), such as that the result of a read immediately following a write must be the value just written. The theory *NDet* of nondeterministic choice has one operation *coin* that returns a Boolean value and certain equations specifying *coin* (Example 2.5).

These two theories can be separately handled by two handlers h_{ST} and h_{ND} respectively (Figure 1 shows an implementation of them in Eff [Bauer and Pretnar 2015]). The semantics of any handler h is a function *handle* h that applies the handler on computations, i.e. terms built from effectful operations and pure values, producing terms with unhandled operations. Handlers of *State* and *NDet* can be sequentially composed to handle both stateful and nondeterministic operations in a computation M, in the order that either mutable state gets handled first

$$\mathit{handle}\ h_{ND}\ (\mathit{handle}\ h_{ST}\ M) \tag{HStNd}$$

or nondeterminism gets handled first

$$\mathit{handle}\ h_{ST}\ (\mathit{handle}\ h_{ND}\ M) \tag{HNdSt}$$

On the specification side, the theories *State* and *NDet* can be composed into one single theory too. One desirable combination is the *commutative tensor*, or simply *tensor*, of the theories *State* and *NDet*—the theory with all the operations and equations from *State* and *NDet* and additionally equations stating that any operation from *State* is commutative with any operation from *NDet*:

$$\mathbf{do}\ \{b \leftarrow \mathit{coin}\ (); \mathit{put}\ s; k\ b\} = \mathbf{do}\ \{\mathit{put}\ s; b \leftarrow \mathit{coin}\ (); k\ b\} \tag{1}$$

$$\mathbf{do}\ \{b \leftarrow \mathit{coin}\ (); s \leftarrow \mathit{get}\ (); k\ b\ s\} = \mathbf{do}\ \{s \leftarrow \mathit{get}\ (); b \leftarrow \mathit{coin}\ (); k\ b\ s\} \tag{2}$$

Although both handlers and theories are composable, the problem is that the composabilities of handlers and theories are *not* automatically connected. Supposing that the tensor is the desired semantics of combining state and nondeterminism in an application, the programmer needs to pick one from HStNd and HNdSt and prove that it indeed validates all the equations of the tensor. Furthermore, to make the equations useful in reasoning or optimisation, one usually wants to prove that they are *term congruences* under the composite handler—the equation can be applied to transform terms in *any context* under the handler [Kiselyov et al. 2021].

The conventional way to show a composite handler respecting some combination of effect theories is equational reasoning with the *induction principle on computations* [Plotkin and Pretnar 2008]. For example, if one wants to show that the composite handler HStNd validates equation (1) of the tensor, one needs to do an induction on the computation $k\ b$, which is a free variable in the equation. The base case for $k\ b$ is a pure computation returning some value, and the inductive case is $k\ b = \mathbf{do}\ \{a \leftarrow \mathit{op}\ p; k'\ a\}$, where some operation op is invoked and then it acts as some computation k'. In either case, the proof obligation is to show that applying the handler HStNd to the both sides of (1) gives rise to equivalent computations, which can be established by careful calculation. Additionally, if one wants to show that the equation is a term congruence, an additional induction on the context where the equation is used is required. In practice, proving a composite handler respecting some combination of theories in this way can be laborious for several reasons:

- Equations proved to be respected by sub-handlers needs to be re-established for composite handlers because in general, the composite handler does not necessarily respect the equations respected by the sub-handlers (shown later in Example 3.4).
- One needs to explicitly prove that equations respected by a composite handler are term congruences under the handler since it is not true in general [Kiselyov et al. 2021].
- Some ways of combining effect theories create a large number of equations of the same form. For example, the tensor generates quadratically many commutativity equations in the number of operations of the sub-theories, but the common structure in these commutativity equations are not exploited.

1.1 Overview

The aim of this paper is to develop techniques for proving the correctness of composite handlers with respect to combinations of effect theories in a more manageable way. For a class of handlers that are *modular*, as characterised by Schrijvers et al. [2019], given any combination of effect theories, we present a systematic way to devise conditions on handlers so that their composite correctly handles this combination of sub-theories.

To provide a taste of the results developed in the paper, consider the running example in Figure 1. We can show that both handlers are modular and Theorem 5.5 states that any equations respected by modular handlers separately are still respected by their composite. Thus, without any further work, we immediately know that `HStNd` and `HNdSt` respect the equations from *State* and *NDet* because these equations are respected by h_{ST} and h_{ND} separately. Furthermore, if one aims to prove that the composite handler `HStNd` validates the tensor of state and nondeterminism, Theorem 6.1 implies that `HStNd` respects the commutativity equations (1, 2) if each clause c_1 of h_{ST} (in a sense made clear in Section 5.3) and each clause c_2 of h_{ND} satisfy the following equation

$$\begin{aligned} & c_1\ p_1\ (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k\ a_1\ a_2\ s\ q)) \\ =\ & \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to c_1\ p_1\ (\lambda a_1 \to k\ a_1\ a_2)\ s\ q) \end{aligned} \tag{3}$$

for any p_1, p_2, k (made clear in Section 6). The two clauses of h_{ST} are the following functions c_{put} and c_{get} and the only clause of h_{ND} is c_{coin}:

$$c_{put}\ p_1\ k = \lambda s \to k\ ()\ p_1 \qquad c_{get}\ ()\ k = \lambda s \to k\ s\ s$$
$$c_{coin}\ ()\ k = \mathbf{do}\ \{x \leftarrow k\ \mathit{True}; y \leftarrow k\ \mathit{False}; \mathit{return}\ (x + y)\}$$

Then it is straightforward calculation to check that condition (3) is satisfied for each $c_1 \in \{c_{put}, c_{get}\}$ and $c_2 = c_{coin}$. For example, if $c_1 = c_{put}$, then

$$\begin{aligned} & c_1\ p_1\ (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k\ a_1\ a_2\ s\ q)) && \{\text{ definition of } c_{put}\ \} \\ =\ & \lambda s \to (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k\ a_1\ a_2\ s\ q))\ ()\ p_1 \\ =\ & \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k\ ()\ a_2\ p_1\ q) && \{\text{ definition of } c_{put}\ \} \\ =\ & \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to c_1\ p_1\ (\lambda a_1 \to k\ a_1\ a_2)\ s\ q) \end{aligned}$$

Finally, every equation respected by the composite of two modular handlers is automatically a term congruence under the handler (Remark 5.2), eliminating the need of a separate proof. This example will be studied in detail in Section 6.1, and the point is that this is much less of a burden than checking all the conditions separately.

The key technique allowing us to derive such conditions is *handler fusion* [Wu and Schrijvers 2015]: given any two modular handlers h_1 and h_2, we show that there exists a modular handler $h_2 \diamond h_1$ such that

$$handle\ h_2 \cdot handle\ h_1 = handle\ (h_2 \diamond h_1)$$

Consequently, the composite handler $handle\ h_2 \cdot handle\ h_1$ respects an effect theory if and only if $handle\ (h_2 \diamond h_1)$ does, and the latter is easier to work with since it is a single *catamorphism* on computation trees. By the universal property of catamorphisms, $handle\ (h_2 \diamond h_1)$ respects an effect theory if $h_2 \diamond h_1$ respects it, from which we calculate various conditions for $handle\ h_2 \cdot handle\ h_1$ to respect various combinations of effect theories.

1.2 Contributions

After fixing notation for preliminary concepts (Section 2), we introduce and motivate modular handlers (Section 3), and then this paper makes the following contributions:

- a characterisation of correct syntax tree transformations and correct modular handlers with a soundness theorem relating them (Section 4);
- a fusion combinator ($\diamond$) of modular handlers (Section 5) that enables us to reason about the interaction of two handlers when composing them (Corollary 5.4). Particularly, we show that equations separately respected by modular handlers are preserved after composition (Theorem 5.5);
- conditions on handlers for their composite to correctly handle the tensor of the theories (Section 6). As applications, we show that (i) handling mutable state *before* any handler gives rise to a semantics in which stateful operations are commutative with any operation from the latter handler (Theorem 6.5), and that (ii) handling the writer effect and mutable state in either order gives rise to a correct handler of the commutative interaction of the two theories (Theorem 6.6);
- conditions on handlers for their composite to correctly handle the *distributive tensor* of the theories (Section 7), and an application to the handlers of nondeterministic and probabilistic choice (Section 7.1), which exhibits a limitation of the fusion approach.

Finally, we discuss related work (Section 8) and conclude (Section 9).

2 PRELIMINARIES

Throughout this paper, we use Haskell as a vehicle to present all the constructions and results to make them more accessible to functional programmers. We restrict ourselves to a subset of Haskell that is *total*: all recursion is structural; recursive datatypes are inductive; and polymorphism is predicative, etc. Readers familiar with category theory can understand our notation as a meta-language denoting constructions around the category of sets: types denote sets; inductive datatypes denote initial algebras; and polymorphic functions denote ends, etc. In this way, the results developed in this paper apply to any language implementing effect handlers that has a denotational semantics based on the constructions studied in this paper (As an illustration, Appendix F shows such a call-by-value calculus with handlers and a translation to our Haskell constructions). We hope that our notation can be a good compromise between concreteness and generality.

Functors. In Haskell, a functor $f :: * \to *$ is a type constructor instantiating the *Functor* type class (Figure 2). It is also expected to satisfy the functor laws:

$$fmap\ id = id \qquad fmap\ g \cdot fmap\ h = fmap\ (g \cdot h)$$

For any functor f, we call a function of type $f\ c \to c$ an f*-algebra* and the type c the *carrier* of this f-algebra. For example, given types P and A, then **data** $\Sigma\ x = O\ P\ (A \to x)$ with the following

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\\ \quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{array} \qquad \begin{array}{l}\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ \quad \mathit{return} :: a \to m\ a\\ \quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{array}$$

Fig. 2. Type classes for functors and monads in Haskell.

fmap is a functor:

$$\textbf{instance}\ \mathit{Functor}\ \Sigma\ \textbf{where}\ \mathit{fmap}\ f\ (O\ p\ k) = O\ p\ (f \cdot k) \tag{4}$$

Given any two functors f and g, their coproduct $f + g$ is given by the following datatype, and it can also be equipped with a functor instance.

$$\textbf{data}\ (f + g)\ a = \mathit{Inl}\ (f\ a) \mid \mathit{Inr}\ (g\ a) \tag{5}$$

$$\mathit{fmap}\ h\ (\mathit{Inl}\ x) = \mathit{Inl}\ (\mathit{fmap}\ h\ x) \qquad \mathit{fmap}\ h\ (\mathit{Inr}\ y) = \mathit{Inr}\ (\mathit{fmap}\ h\ x)$$

Monads. A functor m is a monad if it instantiates the *Monad* type class (Figure 2) and adheres to the monad laws:

$$\mathit{join} \cdot \mathit{return} = \mathit{id} \qquad \mathit{join} \cdot \mathit{fmap}\ \mathit{return} = \mathit{id} \qquad \mathit{join} \cdot \mathit{join} = \mathit{join} \cdot \mathit{fmap}\ \mathit{join} \tag{6}$$

where $\mathit{join} :: m\ (m\ a) \to m\ a$ is defined by $\mathit{join}\ m = m \gg\!= \mathit{id}$. Pioneered by Moggi [1991], monads are used to model computational effects. Intuitively, *return* turns a pure value into a trivial computation causing no effects, and $m \gg\!= f$ executes computation m first, letting its result be x, then executes $f\ x$. Thus Haskell supports the do-notation $\textbf{do}\ x \leftarrow m; b$, which is a syntactic sugar for $m \gg\!= (\lambda x \to b)$.

Free Monads. For any functor f, the inductive datatype $\mathit{Free}\ f$ is called the free monad:

$$\textbf{data}\ \mathit{Free}\ f\ v = \mathit{Var}\ v \mid \mathit{Op}\ (f\ (\mathit{Free}\ f\ v)) \tag{7}$$

Intuitively, an element of $\mathit{Free}\ f\ v$ is a tree with leaf nodes constructed by *Var* and internal nodes constructed by *Op*, where the functor f determines the branching structure of internal nodes. Given an f-algebra $\mathit{alg} :: f\ c \to c$ and function $\mathit{gen} :: v \to c$, there is function *fold* (also known as *catamorphism*) that recursively reduces $\mathit{Free}\ f\ v$ to the carrier c of *alg*:

$$\begin{array}{l}\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (v \to c) \to (f\ c \to c) \to \mathit{Free}\ f\ v \to c\\ \mathit{fold}\ \mathit{gen}\ \mathit{alg}\ (\mathit{Var}\ x)\ = \mathit{gen}\ x\\ \mathit{fold}\ \mathit{gen}\ \mathit{alg}\ (\mathit{Op}\ \mathit{op}) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{gen}\ \mathit{alg})\ \mathit{op})\end{array} \tag{8}$$

The monad instance of $\mathit{Free}\ f$ is implemented with *fold*:

$$\begin{array}{l}\mathit{return} :: v \to \mathit{Free}\ f\ v\\ \mathit{return} = \mathit{Var}\end{array} \qquad \begin{array}{l}(\gg\!=) :: \mathit{Free}\ f\ v \to (v \to \mathit{Free}\ f\ u) \to \mathit{Free}\ f\ u\\ m \gg\!= f = \mathit{fold}\ f\ \mathit{Op}\ m\end{array} \tag{9}$$

Intuitively, $\mathit{return}\ x$ is a variable x, and $m \gg\!= f$ performs substitution of variables in m using f.

2.1 Algebraic Theories

Plotkin and Power [2002] propose to represent a computational effect by an *algebraic theory*, which is a set of primitive effectful operations and a set of equations on those operations characterising the behaviour of the operations. In this section, we provide an account of algebraic theories as the basis for our development, and introduce our notation.

Signature Functors. A signature is a finite set of operation symbols $\{O_i\}$, each paired with a *parameter* type P_i and an *arity* type A_i (or *result type* by some authors). A signature with n operations can be described by a *signature functor* Σ of the following form:

$$\mathbf{data}\ \Sigma\ x = O_1\ P_1\ (A_1 \to x) \mid O_2\ P_2\ (A_2 \to x) \mid \cdots \mid O_n\ P_n\ (A_n \to x)$$

(with the evident *Functor* instance similar to (4)). In this paper, we use notation $O :: P \rightsquigarrow_\Sigma A$ to mean that O is an operation in Σ with parameter type P and arity type A, i.e. there is a constructor $O :: P \to (A \to x) \to \Sigma\ x$ for the signature functor Σ. We sometimes omit the subscript Σ in $\rightsquigarrow_\Sigma$ if it is clear from context. An intuitive interpretation of a $P \rightsquigarrow A$ operation is an effectful computation taking a P-value and returning an A-value, or equivalently, an operation parameterised by a P-value and combining $|A|$-many possible ways of continuing the computation into one computation [Bauer 2018; Plotkin and Power 2004].

Example 2.1 (Nondeterministic Choice). The signature *NDet* of nondeterministic choice has one operation $Coin :: () \rightsquigarrow Bool$ with *Bool* as its arity type. For aesthetic reasons, we prefer the infix $(\sqcap) :: x \to x \to NDet\ x$ instead of *Coin*, where

$$x \sqcap y = Coin\ ()\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)$$

The operation *Coin* is intended to return a *Bool* value nondeterministically, or equivalently, $p \sqcap q$ behaves like p or q nondeterministically.

Example 2.2 (Mutable State). The signature $State_s$ of mutable state of type s has two operations: $Get :: () \rightsquigarrow s$ and $Put :: s \rightsquigarrow ()$. The operation *Get* is intended to read and return the state, and *Put* is intended to overwrite the state with its parameter of type s (and return nothing).

Example 2.3 (Empty Theory). An algebraic theory useful for our later development is the trivial theory *Empty* with no operations and equations. Thus its signature functor *Empty* has no constructors.

Equations. An equation for a signature Σ is a pair of terms built from operations in Σ and some free variables. For example, the following is an equation for signature $State_s$:

$$Put\ u\ (\lambda() \to Put\ u'\ (\lambda() \to k)) \quad = \quad Put\ u'\ (\lambda() \to k) \tag{10}$$

where u, u' and k are free variables. Note that we have two kinds of free variables: k stands for a *computation*, whereas u and u' stands for *values* of type s. One way to formalise equations is to use the free monad (7): an equation is formalised as a pair of elements of $\Gamma \to Free\ \Sigma\ v$ for some types Γ and v, where Γ is the type representing all free *value variables* and v is the type *indexing* all free *computation variables*:

$$\mathbf{data}\ Equation\ \Sigma\ \Gamma\ v = (\doteq)\ (\Gamma \to Free\ \Sigma\ v)\ (\Gamma \to Free\ \Sigma\ v) \tag{11}$$

where binary operator $\doteq$ is the constructor. Free value variables and computations variables are treated differently to leave the type of computations abstract in equations. For the example (10) above, Γ is (s, s) since there are two free value variables of type s in the equation, and v is the unit type $()$ indicating that there is one free computation variable in the equation:

```
putPutEq :: Equation State_s (s, s) ()
putPutEq = (lhs ≐ rhs) where
  lhs, rhs :: (s, s) → Free State_s ()
  lhs (u, u′) = Op (Put u (λ() → Op (Put u′ (λ() → Var ()))))
  rhs (u, u′) = Op (Put u′ (λ() → Var ()))
```

In the main text of this paper, we will stick to the informal form of equations as in Equation 10 for brevity, and the formal form will only be used in proofs. It is straightforward to convert between the two forms, i.e. elements of *Equation* $\Sigma\ \Gamma\ v$, by collecting free variables of computations into a type v and free variables of values into a type Γ and inserting *Var* and *Op* appropriately.

Example 2.4. Continuing Example 2.2, the theory of mutable state traditionally comes with four equations [Plotkin and Power 2002]. Letting $put\ s\ c = Put\ s\ (\lambda() \to c)$ and $get\ k = Get\ ()\ k$, the four equations of mutable state are

$$put\ s\ (get\ k) = put\ s\ (k\ s) \qquad put\ s\ (put\ s'\ k) = put\ s'\ k$$

$$get\ (\lambda s \to get\ (\lambda s' \to k\ s\ s')) = get\ (\lambda s \to k\ s\ s) \qquad get\ (\lambda s \to put\ s\ k) = k$$

where k, s and s' are all free variables. The type of k may be different for each equation.

Example 2.5. Continuing Example 2.1, the theory *NDet* of nondeterminism has as equations idempotence, symmetry and associativity of the operation $\sqcap$, which are the axioms of semi-lattices:

$$x \sqcap x = x \qquad x \sqcap y = y \sqcap x \qquad x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$$

where x, y and z are all free variables of computations.

Definition 2.1 (Equation Respecting)**.** Given $(lhs \doteq rhs) :: Equation\ \Sigma\ \Gamma\ v$ and $alg :: \Sigma\ c \to c$, a Σ-algebra with carrier c, we say that alg respects this equation if for any $t :: \Gamma$ and $k :: v \to c$,

$$fold\ k\ alg\ (lhs\ t) = fold\ k\ alg\ (rhs\ t)$$

In other words, substituting *alg* for operations in the equation and any values for the free variables in the equation gives equal results.

Example 2.6. Consider the $State_s$-algebra $alg_{ST} :: State_s\ (s \to a) \to s \to a$

$$alg_{ST}\ (Put\ s'\ k) = \lambda s \to k\ ()\ s' \qquad alg_{ST}\ (Get\ s\ k) = \lambda s \to k\ s\ s$$

It can be checked to respect all the equations in Example 2.4. For example, the first equation $put\ s\ (get\ k) = put\ s\ (k\ s)$ is respected because for any $k :: s \to s \to a$ and $t :: s$,

$$\begin{aligned} fold\ k\ alg\ (lhs\ t) &= fold\ k\ alg_{ST}\ (Op\ (Put\ t\ (\lambda() \to Op\ (Get\ ()\ (\lambda s \to Var\ s))))) \\ &= \lambda s \to k\ t\ t \\ &= fold\ k\ alg_{ST}\ (Op\ (Put\ t\ (\lambda() \to Var\ t))) = fold\ k\ alg\ (rhs\ t) \end{aligned}$$

Example 2.7. Given any semi-lattice $(L, \cup)$, the equations in Example 2.5 are respected by the *NDet*-algebra $alg\ (Coin\ ()\ k) = k\ True \cup k\ False$.

Definition 2.2 (Algebraic Theories)**.** An algebraic theory T is a signature functor Σ equipped with a set of equations of type $Equation\ \Sigma\ \Gamma\ v$ for some types Γ and v (different equations may have different Γ and v's). We use the notation $T :: Theory\ \Sigma$ to mean a theory T of signature Σ.

Algebraic theories are also known as *equational theories*, and are equivalent to *Lawvere theories* [Hyland et al. 2006] that present theories as categories. When the associated equations are clear, we sometimes abuse the name of a signature functor to mean a theory of this signature. For example, when we say the theory $State_s$ in the rest of the paper, we mean the theory of signature $State_s$ and the four equations in Example 2.4.

2.2 Combinations of Theories

Hyland et al. [2006] show how algebraic theories can be combined by various ways to specify the operations and equations of the combined theory based on the sub-theories. In this section, we reformulate the *sum*, *tensor* and *distributive tensor* [Plotkin and Power 2004] in our simplified setting for convenience.

For all the ways of combining effect theories in this paper, the operations of the combined theory are the disjoint union of the operations of the sub-theories, i.e. the signature functor of the combined theory is the coproduct of the signature functors of the sub-theories (5). Equations of the combined theory have greater freedom of choice. A straightforward combination of two theories is just taking the union of the equations of the sub-theories and no more.

Definition 2.3 (Sum of Theories [Hyland et al. 2006])**.** The *sum* of $T_1 :: \mathit{Theory}\ \Sigma_1$ and $T_2 :: \mathit{Theory}\ \Sigma_2$, denoted $T_1 + T_2$, is the theory of signature $\Sigma_1 + \Sigma_2$ with exactly the equations of T_1 and T_2 (regarded as equations on signature $\Sigma_1 + \Sigma_2$).

One can also include equations in the combined theory to specify interactions between operations from the sub-theories, such as commutativity between operations from sub-theories.

Definition 2.4 (Tensor of Theories [Hyland et al. 2006])**.** The *commutative combination* or *tensor* of $T_1 :: \mathit{Theory}\ \Sigma_1$ and $T_2 :: \mathit{Theory}\ \Sigma_2$, denoted $T_1 \otimes T_2$, is the theory of signature $\Sigma_1 + \Sigma_2$ with all equations of T_1 and T_2, and for each $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ and $O_2 :: P_2 \rightsquigarrow_{\Sigma_2} A_2$, a commutativity law:

$$\overline{O_1}\ p_1\ (\lambda a_1 \to \overline{O_2}\ p_2\ (\lambda a_2 \to k\ a_1\ a_2)) = \overline{O_2}\ p_2\ (\lambda a_2 \to \overline{O_1}\ p_1\ (\lambda a_1 \to k\ a_1\ a_2))$$

where $\overline{O_1}\ p\ k = \mathit{Inl}\ (O_1\ p\ k)$ and $\overline{O_2}\ p\ k = \mathit{Inr}\ (O_2\ p\ k)$ lift O_1 and O_2 as operations in signature $\Sigma_1 + \Sigma_2$, and $p_1 :: P_1$, $p_2 :: P_2$ and k are free variables.

Example 2.8. When a program involves two mutable states that are independent of each other, we can model the situation by the tensor $\mathit{State}_{s_1} \otimes \mathit{State}_{s_2}$ of two mutable states, since the order of two consecutive operations on independent mutable states can be swapped without changing the semantics, as long as the parameter of the second operation does not depend on the result of the first operation.

Another combination that we are going to discuss in Section 7 is adding distributivity laws in the combination. Distributivity is commonly stated for binary operations, such as for + and ×,

$$x_1 \times (y_1 + y_2) = (x_1 \times y_1 + x_1 \times y_2) \qquad (y_1 + y_2) \times x_2 = (y_1 \times x_2 + y_2 \times x_2)$$

It can be generalised to operations of finite arity: let O_1 be an n-ary operation and O_2 be an m-ary operation for natural numbers n and m, the distributive laws of O_1 over O_2 are those equations that for each $0 \leqslant k \leqslant n-1$,

$$O_1\ x_1 \cdots x_k\ (O_2\ y_1\ \ldots\ y_m)\ x_{k+2} \cdots x_n \ = \ \begin{array}{l} O_2\ (O_1\ x_1 \cdots x_k\ y_1\ x_{k+2} \cdots x_n) \\ \quad (O_1\ x_1 \cdots x_k\ y_2\ x_{k+2} \cdots x_n) \\ \quad \vdots \\ \quad (O_1\ x_1 \cdots x_k\ y_m\ x_{k+2} \cdots x_n) \end{array}$$

By passing all operands by a function as in we do in signature functors, the distributive laws generalise to operations $O_1 :: P_1 \rightsquigarrow A_1$ and $O_2 :: P_2 \rightsquigarrow A_2$ with possibly infinite arity:

$$\begin{array}{l} O_1\ p_1\ (\lambda i \to \mathbf{if}\ i \equiv a\ \mathbf{then} \\ \quad O_2\ p_2\ (\lambda j \to y\ j)\ \mathbf{else}\ x\ i) \end{array} = \begin{array}{l} O_2\ p_2\ (\lambda j \to O_1\ p_1\ (\lambda i \to \\ \quad \mathbf{if}\ i \equiv a\ \mathbf{then}\ y\ j\ \mathbf{else}\ x\ i)) \end{array} \tag{12}$$

where x and y are free variables of computations and $p_1 :: P_1$, $p_2 :: P_2$ and $a :: A_1$ are free variables of values.

Definition 2.5 (Distributive Tensor [Plotkin and Power 2004])**.** The *distributive combination* or *distributive tensor* of $T_1 :: \mathit{Theory}\ \Sigma_1$ and $T_2 :: \mathit{Theory}\ \Sigma_2$, denoted $T_1 \triangleright T_2$, is the theory of signature $\Sigma_1 + \Sigma_2$ with all equations of T_1 and T_2 and additionally for each $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ and $O_2 :: P_2 \rightsquigarrow_{\Sigma_2} A_2$, the distributive law (12) of O_1 over O_2 (lifted to be operations in $\Sigma_1 + \Sigma_2$ as in Definition 2.4).

Example 2.9 (Combined Choice)*.* Some nondeterministic systems involve probabilistic behaviour too. The theory *Prob* of probabilistic choice has a binary operation $\mathit{PChoose} :: \mathit{Float} \rightsquigarrow \mathit{Bool}$ with a *Float* parameter in the range $[0, 1]$. Operation $\mathit{PChoose}\ p\ k$ is preferably written in infix notation $x \triangleleft p \triangleright y = \mathit{PChoose}\ p\ (\lambda b \rightarrow \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y)$ following Hoare [Hoare 1985]. Letting $\bar{p}$ denote $1 - p$, theory *Prob* has the following equations:

$$x \triangleleft 1 \triangleright y = x \qquad x \triangleleft p \triangleright x = x \qquad x \triangleleft p \triangleright y = y \triangleleft \bar{p} \triangleright x$$

$$x \triangleleft p \triangleright (y \triangleleft q \triangleright z) = (x \triangleleft r \triangleright y) \triangleleft s \triangleright z \quad (p = rs, \bar{s} = \bar{p}\bar{q})$$

For a system involving nondeterministic choice and probabilistic choice, one desirable interaction of the two effects is the distributive tensor of *Prob* over *NDet* [Mislove et al. 2004], i.e. operations and equations from both theories with additional equations:

$$x \triangleleft p \triangleright (y \sqcap z) = (x \triangleleft p \triangleright y) \sqcap (x \triangleleft p \triangleright z) \qquad (x \sqcap y) \triangleleft p \triangleright z = (x \triangleleft p \triangleright z) \sqcap (y \triangleleft p \triangleright z)$$

3 SYNTAX AND SEMANTICS OF COMPUTATIONS

Now that we have formalised theories of effects, we continue to set the stage by showing how one can formalise the syntax and semantics of computations involving an effect. Given an effect theory, the syntax of computations involving the effect is modelled by terms built from operations of the theory (Section 3.1), and semantics is provided by *handlers* that interpret operations in syntax trees by *fold* (Section 3.2). However, we show that the traditional formulation of handlers lacks *modularity* when the effect theory is composed from sub-effects. Particularly, equations respected by one handler may be invalidated by other handlers when composing handlers together. The problem motivates *modular handlers* [Schrijvers et al. 2019], which ensure handlers to work independently of each other by parametricity (Section 3.3) and play a crucial role in later sections.

3.1 Terms of Computations

Given a signature Σ, computations that involve operations in Σ and produce values of type a are modelled by the free monad $\mathit{Free}\ \Sigma\ a$ in (7). An element of $\mathit{Free}\ \Sigma\ a$ is either $\mathit{Var}\ x$, which represents a pure computation returning x, or $\mathit{Op}\ (O\ p\ k)$ for some $O :: P \rightsquigarrow_{\Sigma} A$, $p :: P$ and $k :: A \rightarrow \mathit{Free}\ \Sigma\ a$, which represents a computation making an operation call O with parameter p and continuing as $k\ x$ when the result of the operation is $x :: A$.

Recall that $\mathit{Free}\ \Sigma$ is a monad (9), and its $\gg\!=$ precisely means sequential composition of operations when understanding $\mathit{Free}\ \Sigma$ as computations. For any operation $O :: P \rightsquigarrow_{\Sigma} A$, we have a function $O_g :: P \rightarrow \mathit{Free}\ \Sigma\ A$, called a *generic operation* [Plotkin and Power 2003], such that $O_g\ p = \mathit{Op}\ (O\ p\ \mathit{Var})$. Generic operations and the monadic instance of $\mathit{Free}\ \Sigma$ usually allow one to build computation terms more easily than directly using the underlying constructors.

Example 3.1. Computation $\mathit{incr} :: \mathit{Free}\ \mathit{State}_{\mathit{Int}}\ \mathit{Int}$ gets the state, increments it and returns the original value.

$$\mathit{incr} = \mathit{Op}\ (\mathit{Get}\ ()\ (\lambda i \rightarrow \mathit{Op}\ (\mathit{Put}\ (i+1)\ (\lambda () \rightarrow \mathit{Var}\ i))))$$

Using generic operations, incr can be conveniently written as $\mathbf{do}\ i \leftarrow \mathit{Get}_g\ (); \mathit{Put}_g\ (i+1); \mathit{return}\ i$.

The equations of an effect theory indicate that some terms of computations should be deemed as equivalent, which is captured by the following relation on computations.

Definition 3.1 (Equivalent Computations)**.** Given a theory $T :: \mathit{Theory}\ \Sigma$ and a type a, we define a binary relation $\sim_T$ on elements of $\mathit{Free}\ \Sigma\ a$ inductively by the following rules:

$$\frac{c :: \mathit{Free}\ \Sigma\ a}{c \sim_T c}\ \textsc{Refl} \qquad \frac{c \sim_T d}{d \sim_T c}\ \textsc{Sym} \qquad \frac{c \sim_T d \qquad d \sim_T e}{c \sim_T e}\ \textsc{Trans}$$

$$\frac{O_i :: P \rightsquigarrow A \in \Sigma \qquad k, k' :: A \to \mathit{Free}\ \Sigma\ a \qquad \forall x :: A.\ k\ x \sim_T k'\ x}{\mathit{Op}\ (O_i\ p\ k) \sim_T \mathit{Op}\ (O_i\ p\ k')}\ \textsc{Cong}$$

$$\frac{((\mathit{lhs} \doteq \mathit{rhs}) :: \mathit{Equation}\ \Sigma\ \Gamma\ V) \in T \qquad g :: \Gamma \qquad k :: V \to \mathit{Free}\ \Sigma\ a}{\mathit{fold}\ k\ \mathit{Op}\ (\mathit{lhs}\ g) \sim_T \mathit{fold}\ k\ \mathit{Op}\ (\mathit{rhs}\ g)}\ \textsc{Eq}$$

Relation $x \sim_T y$ captures the idea of two computations being equivalent under theory T. The first three rules make it an equivalence relation; rule Cong makes it compatible with the structure of free monad, i.e. a *term congruence*—whenever k and k' are equivalent subterms, enclosing them in the context $\mathit{Op}\ (O_i\ p\ \_)$ is still equivalent; the rule Eq asserts that instantiating equations $\mathit{lhs} = \mathit{rhs}$ from the theory T with any value g and subterms k gives rise to equivalent computations.

Example 3.2. Consider the theory State_s from Example 2.4 and computation

$$\mathit{incr}' = \mathbf{do}\ i \leftarrow \mathit{Get}_g\ (); \mathit{Put}_g\ (i+1); \mathit{Put}_g\ (i+1); \mathit{return}\ i$$

With the theory $\mathit{State}_{\mathit{Int}}$ from Example 2.4, it is derivable that

$$\mathbf{do}\ \{\mathit{Put}_g\ (i+1); \mathit{Put}_g\ (i+1); \mathit{return}\ i\} \sim_{\mathit{State}_{\mathit{Int}}} \mathbf{do}\ \{\mathit{Put}_g\ (i+1); \mathit{return}\ i\}$$

using the Eq rule and the second equation in Example 2.4. Then using the Cong rule, it is derivable that $\mathit{incr}' \sim_{\mathit{State}_{\mathit{Int}}} \mathit{incr}$ for the incr from Example 3.1.

The relation $\sim_T$ plays an important role in the separation of specification and implementation of algebraic effects. The 'user' of effects uses relation $\sim_T$ to reason about and optimise programs without knowing how effect operations are implemented, and the 'implementer' of effects is responsible for the correctness of the implementation with respect to the relation $\sim_T$.

3.2 Traditional Handlers and Non-Modularity

Assuming an effect signature Σ, the simplest form of a handler is a pair of two functions $\mathit{gen} :: a \to \mathit{Free}\ \Sigma\ b$ and $\mathit{alg} :: \Sigma\ (\mathit{Free}\ \Sigma\ b) \to \mathit{Free}\ \Sigma\ b$ for some types a and b. We call $(\mathit{gen}, \mathit{alg})$ *a handler from a to b*. It induces a function $\mathit{handle}_T\ (\mathit{gen}, \mathit{alg}) :: \mathit{Free}\ \Sigma\ a \to \mathit{Free}\ \Sigma\ b$ that applies the handler to a computation $\mathit{Free}\ \Sigma\ a$ by

$$\begin{aligned} &\mathit{handle}_T\ (\mathit{gen}, \mathit{alg})\ (\mathit{Var}\ x) &&= \mathit{gen}\ x \\ &\mathit{handle}_T\ (\mathit{gen}, \mathit{alg})\ (\mathit{Op}\ (O\ p\ k)) &&= \mathit{alg}\ (O\ p\ (\mathit{handle}_T\ (\mathit{gen}, \mathit{alg}) \cdot k)) \end{aligned}$$

for any operation O in Σ. The gen function is the 'return clause' of the handler transforming a pure a-value $\mathit{Var}\ x$ to a computation of a b-value. The alg function is the 'operation clauses' transforming an operation call $O\ p\ k$ with its continuations k for any possible result of this operation to a computation of a b-value.

Example 3.3. Assuming $\Sigma = \mathit{State}_s + \mathit{NDet}$ and a datatype $\mathit{Set}\ a$ whose elements are subsets of the set denoted by a, then $(\mathit{gen}_{\mathit{ND}}, \mathit{alg}_{\mathit{ND}})$ below is a handler from a to $\mathit{Set}\ a$:

$$\mathit{gen}_{\mathit{ND}}\ x \qquad = \mathit{return}\ \{\, x \,\}$$

$$\begin{aligned} alg_{ND}\ (Inl\ op) &= Op\ (Inl\ op) \\ alg_{ND}\ (Inr\ (Coin\ ()\ k)) &= \mathbf{do}\ \{x \leftarrow k\ True; y \leftarrow k\ False; return\ (x \cup y)\} \end{aligned}$$

Note how *alg* forwards any operation not in *Coin* using *Op*.

Non-Modularity. This formulation of handlers is suitable for giving denotational semantics to calculi of effect handlers that assume a global signature of effects and do not come with a type-and-effect system, such as the original one in [Plotkin and Pretnar 2009]. However, this simple formulation suffers from the problem that a handler of a signature can potentially alter operations not intended to be handled by it arbitrarily, breaking the modular principle followed by the approach of algebraic effects and causing difficulties in reasoning. We demonstrate the problem by the following example.

Example 3.4. Assuming $\Sigma = State_s + NDet$, consider the following handler $(gen_{ND}{}', alg_{ND}{}')$

$$\begin{aligned} gen_{ND}{}'\ x &= return\ \{x\} \\ alg_{ND}{}'\ (Inl\ op) &= Op\ (Inl\ op) \\ alg_{ND}{}'\ (Inr\ (Coin\ ()\ k)) &= \mathbf{do}\ \{x \leftarrow fold\ Var\ alg'\ (k\ True); y \leftarrow k\ False; return\ (x \cup y)\} \\ &\quad \mathbf{where}\ alg'\ x = \mathbf{case}\ x\ \mathbf{of}\ \{(Inl\ (Put\ s\ k)) \rightarrow k\ (); \_ \rightarrow Op\ x\} \end{aligned}$$

which handles *NDet* but additionally erases every call to *Put* in the first branch of nondeterministic choice using a *fold*. Compared to (gen_{ND}, alg_{ND}) from Example 3.3, $(gen_{ND}{}', alg_{ND}{}')$ is less modular because it not only handles *NDet* but also alters operations not in *NDet*. Consequently, $(gen_{ND}{}', alg_{ND}{}')$ interacts less nicely with other handlers. To see this, consider the following handler (gen_{ST}, alg_{ST}) from a to $s \rightarrow Free\ (State_s + NDet)\ a$:

$$\begin{aligned} gen_{ST}\ x &= return\ (\lambda s \rightarrow return\ x) \\ alg_{ST}\ (Inl\ (Get\ ()\ k)) &= return\ (\lambda s \rightarrow \mathbf{do}\ f \leftarrow k\ s; f\ s) \\ alg_{ST}\ (Inl\ (Put\ s'\ k)) &= return\ (\lambda s \rightarrow \mathbf{do}\ f \leftarrow k\ (); f\ s') \\ alg_{ST}\ op &= Op\ op \end{aligned}$$

which respects all the equations of $State_s$ in Example 2.4. However, the composite handler

$$handle_T\ (gen_{ST}, alg_{ST}) \cdot handle_T\ (gen_{ND}{}', alg_{ND}{}')$$

no longer respects the first equation $put\ s\ (get\ k) = put\ s\ (k\ s)$ because the left-hand side is transformed to $Var\ (\lambda s_0 \rightarrow k\ s_0)$, while the right-hand side is transformed to $Var\ (\lambda s_0 \rightarrow k\ s)$, which are not equal in general.

In general, even when (gen_1, alg_1) and (gen_2, alg_2) respect effect theories T_1 and T_2 respectively, it is *not* guaranteed that their composite handler respects all the equations in T_1 and T_2, which hinders modular reasoning about effect handlers.

3.3 Modular Carriers and Handlers

The problem in the last subsection can be rectified by restricting handlers to *modular handlers* introduced by Schrijvers et al. [2019]. The key idea is to require handlers to be explicit about what operations got handled and be *polymorphic* (or *natural* in categorical terminology) in unhandled operations so that a handler cannot alter unhandled operations arbitrarily, precluding handlers such as $(gen_{ND}{}', alg_{ND}{}')$.

One seemingly reasonable way to achieve this is to require the *alg* function of a handler of signature *sig* to type *b* to have type

$$alg :: \forall sig'.\ sig\ (Free\ sig'\ b) \rightarrow Free\ sig'\ b$$

so that *alg* is polymorphic in the signature sig' of unhandled operations. Although this restriction precludes (gen_{ND}', alg_{ND}'), this type of *alg* still exposes the fact that the result is a free monad $Free\ sig'\ b$, and therefore *alg* can still alter the tree structure of $Free\ sig'$, such as duplicating and removing nodes in a $Free\ sig'\ b$ while being polymorphic in sig'. One way to fix this is to require the *alg* function to have type

$$alg :: \forall m.\ Monad\ m \Rightarrow sig\ (m\ b) \to m\ b \tag{13}$$

so that *alg* is polymorphic in a monad m representing the remaining computational effects in the computation. This idea is further generalised by Schrijvers et al. [2019] to *modular carriers*, which is a type $c\ m$ parameterised by a monad m that represents the remaining computational effects in the computation, and moreover, $c\ m$ should provide a way to *forward* operations in m.

Definition 3.2 (Modular Carriers [Schrijvers et al. 2019])**.** Type constructor $c :: (* \to *) \to *$ is a *modular carrier* if it instantiates the following type class

$$\textbf{class}\ MCarrier\ c\ \textbf{where}\ fwd :: Monad\ m \Rightarrow m\ (c\ m) \to c\ m$$

subject to the laws of Eilenberg-Moore algebras [Mac Lane 1998], i.e. for any monad m,

$$fwd \cdot return_m = id \qquad fwd \cdot fmap\ fwd = fwd \cdot join_m \tag{14}$$

The first equation is on type $c\ m \to c\ m$, and it states that forwarding a trivial computation created by *return* does nothing. The second one is on type $m\ (m\ (c\ m)) \to c\ m$, and it states that forwarding two layers of computational effects one-by-one is equivalent to forwarding the sequential composition of them.

Example 3.5. A straightforward but useful modular carrier is

$$\textbf{newtype}\ FreeEM\ a\ m = FreeEM\ \{unFreeEM :: m\ a\}$$

in the record syntax of Haskell, which defines a constructor and destructor of the following types:

$$FreeEM :: m\ a \to FreeEM\ a\ m \qquad unFreeEM :: FreeEM\ a\ m \to m\ a$$

It is a modular carrier with the following *fwd*:

$$\textbf{instance}\ MCarrier\ (FreeEM\ a)\ \textbf{where}\ fwd = FreeEM \cdot join \cdot fmap\ unFreeEM$$

The laws of monads in (6) imply that the laws in (14) are satisfied. The name *FreeEM* comes from the fact that $FreeEM\ a\ m \cong m\ a$ with *join* is the *free Eilenberg-Moore algebra* for a. The scheme in (13) is then equivalent to

$$alg :: \forall m.\ Monad\ m \Rightarrow sig\ (FreeEM\ b\ m) \to FreeEM\ b\ m$$

Example 3.6. Another modular carrier is a family of computations indexed by some type s:

$$\begin{array}{l}\textbf{newtype}\ StateC\ s\ a\ m = StateC\ \{unStateC :: s \to m\ a\} \\ \textbf{instance}\ MCarrier\ (StateC\ s\ a)\ \textbf{where}\ fwd\ mc = StateC\ (\lambda s \to (\textbf{do}\ \{f \leftarrow mc; unStateC\ f\ s\}))\end{array}$$

This carrier is useful for interpreting handlers with parameters [Brady 2013; Kammar et al. 2013; Kiselyov et al. 2013]. We will use this carrier for the handler of mutable state later.

The *fwd* function of a modular carrier is polymorphic in any monad m. In particular, when m is $Free\ sig'$, the following function is able to forward one operation call:

$$\begin{array}{l}forward :: (MCarrier\ c, Functor\ sig') \Rightarrow sig'\ (c\ (Free\ sig')) \to c\ (Free\ sig') \\ forward\ op = fwd\ (Op\ (fmap\ return\ op))\end{array} \tag{15}$$

Definition 3.3 (Modular Handlers [Schrijvers et al. 2019]). Given a signature *sig*, a *modular handler* h for *sig* from type a to b carried by modular carrier c consists of three functions (gen, alg, run) packed into the following record:

$$\begin{aligned}\textbf{data}\ \mathit{MHandler}\ sig\ c\ a\ b = \mathit{MHandler}\ \{&gen :: \forall m.\ \mathit{Monad}\ m \Rightarrow a \rightarrow c\ m\\ ,\ &alg :: \forall m.\ \mathit{Monad}\ m \Rightarrow sig\ (c\ m) \rightarrow c\ m\\ ,\ &run :: \forall m.\ \mathit{Monad}\ m \Rightarrow c\ m \rightarrow m\ b\}\end{aligned}$$

which induces a function $(handle\ h) :: \forall sig'.\ \mathit{Free}\ (sig + sig')\ a \rightarrow \mathit{Free}\ sig'\ b$ such that

$$handle\ h = run \cdot fold\ gen\ alg'$$

where $alg'\ (\mathit{Inl}\ op') = alg\ op'$ and $alg'\ (\mathit{Inr}\ op') = forward\ op'$.

The *gen* and *alg* functions of a modular handler play similar roles as in traditional handlers. The *run* function additionally allows a modular handler to do some post-processing after the fold, such as providing a initial state to a parameterised handler.

Example 3.7. The handler of *NDet* in Example 3.3 can be turned into a modular hander with modular carrier *FreeEM* from Example 3.5:

$$\begin{aligned}&ndetH :: \mathit{MHandler}\ \mathit{NDet}\ (\mathit{FreeEM}\ (\mathit{Set}\ a))\ a\ (\mathit{Set}\ a)\\ &ndetH = \mathit{MHandler}\ \{gen = gen_{ND}, alg = alg_{ND}, run = unFreeEM\}\ \textbf{where}\\ &\quad gen_{ND}\ a = \mathit{FreeEM}\ (return\ \{\,a\,\})\\ &\quad alg_{ND}\ (\mathit{Coin}\ ()\ k) = \mathit{FreeEM}\ (\textbf{do}\ x \leftarrow unFreeEM\ (k\ \mathit{True}); y \leftarrow unFreeEM\ (k\ \mathit{False});\\ &\qquad\qquad\qquad\qquad\qquad\qquad return\ (x \cup y))\end{aligned}$$

Compared to its non-modular counterpart in Example 3.3, alg_{ND} does not deal with forwarding unhandled operations, since they are forwarded by *handle*.

Example 3.8. The handler of $State_s$ in Example 3.4 can be translated into a modular handler with modular carrier $m\ (s \rightarrow m\ a)$, but the outer layer of m is unnecessary, and we can define the following modular handler of $State_s$ with carrier $\mathit{StateC}\ s\ a\ m \cong s \rightarrow m\ a$:

$$\begin{aligned}&stH :: s \rightarrow \mathit{MHandler}\ State_s\ (\mathit{StateC}\ s\ a)\ a\ a\\ &stH\ s = \mathit{MHandler}\ \{gen = gen_{ST}, alg = alg_{ST}, run = (\lambda c \rightarrow unStateC\ c\ s)\}\ \textbf{where}\\ &\quad gen_{ST}\ a = \mathit{StateC}\ (\lambda s \rightarrow return\ a)\\ &\quad alg_{ST}\ (\mathit{Put}\ s'\ k)\ = \mathit{StateC}\ (\lambda s \rightarrow unStateC\ (k\ ())\ s')\\ &\quad alg_{ST}\ (\mathit{Get}\ ()\ k) = \mathit{StateC}\ (\lambda s \rightarrow unStateC\ (k\ s)\ s)\end{aligned}$$

The handler takes an additional parameter of s that is used as the initial state by the *run* function.

4 CORRECTNESS OF TRANSFORMATIONS AND HANDLERS

A notable missing part in the formulation of modular handlers in the previous section (and [Schrijvers et al. 2019]) is how modular handlers interact with the equations of effect theories. In this section, we recover the missing link between modular handlers and equations by defining notions of *correctness* of syntax-tree transformations and handlers with respect to effect theories.

Definition 4.1 (Correct Open Transformations). Given a theory T of signature Σ and a function f of type $\forall sig'.\ \mathit{Free}\ (\Sigma + sig')\ a \rightarrow \mathit{Free}\ sig'\ b$ for some types a and b, we say that f is a *correct open transformation* for T if for any signature sig', $T' :: \mathit{Theory}\ sig'$ and any two computations $t_1, t_2 :: \mathit{Free}\ (\Sigma + sig')\ a$,

$$t_1 \sim_{T+T'} t_2 \quad \Longrightarrow \quad f\ t_1 \sim_{T'} f\ t_2$$

where $T + T'$ is the sum of T and T' (Definition 2.3).

Under a correct open transformation for T, the programmer can freely use the equations from T to rewrite the operations from T in syntax trees in the presence of operations from other theories. A weaker notion of correctness is desired when a function on syntax trees is expected only to be used in the absence of any other effects.

Definition 4.2 (Correct Closed Transformations)**.** Assuming T and f as in Definition 4.1, we call f a *correct closed transformation* for T if for any two computations $t_1, t_2 :: \mathit{Free}\ (\Sigma + \mathit{Empty})\ A$,

$$t_1 \sim_{T+\mathit{Empty}} t_2 \quad \Longrightarrow \quad \mathit{extract}\ (f\ \ t_1) =_B \mathit{extract}\ (f\ \ t_2)$$

where $\mathit{extract} :: \mathit{Free}\ \mathit{Empty}\ a \rightarrow a$ is defined by $\mathit{extract}\ (\mathit{Var}\ a) = a$.

Remark 4.1. The definition of open correctness implies closed correctness by instantiating T' with the empty theory *Empty*.

The correctness of function *handle h* for some modular handler h is implied by the correctness of handler h defined as follows.

Definition 4.3 (Correct Open and Closed Handlers)**.** Letting T be a theory of signature Σ and $h :: \mathit{MHandler}\ \Sigma\ c\ a\ b$ be a modular handler, (a) we call h a correct *open* handler of T if $\mathit{alg}\ \ h :: \mathit{Monad}\ m \Rightarrow \Sigma\ (c\ m) \rightarrow c\ m$ respects (in the sense of Definition 2.1) all equations of T for any monad m, and (b) we call h a correct *closed* handler of T if $\mathit{alg}\ h$ respects equations of T when m is *Free Empty*.

Theorem 4.1 (Soundness of Correct Handlers)**.** *Letting T be a theory of signature Σ and h be a modular handler of Σ, if h is a correct open (or closed) handler of T, then handle h is a correct open (or closed) transformation of T.*

Proof sketch. We generalise *handle* to work with a polymorphic *term monad* [Wu and Schrijvers 2015] of the remaining effects, which allows us to use parametricity [Voigtländer 2009] to relate the free monad $\mathit{Free}\ \mathit{sig}'$ and the and the monad mapping X to the free model of T' generated by X, i.e. $\mathit{Free}\ \mathit{sig}'\ X$ modulo relation $\sim_{T'}$. A detailed proof can be found in Appendix C. □

Example 4.1. It can be checked that handler *stH* from Example 3.8 is a correct open handler of the theory State_s (Example 2.4). Consequently, *handle stH* is a correct open transformation for State_s.

Example 4.2. It can be checked that *ndetH* from Example 3.7 is a correct open handler of the associativity of nondeterministic choice but not the symmetric law or idempotence law from Example 2.5. This is rather expected because alg_{ND} in Example 3.7 executes both branches of nondeterministic choice *sequentially*. In the open setting, each branch may invoke arbitrary computational effects, so the symmetric law and idempotence cannot hold because they imply that the two branches can be swapped or absorbed into one if they invoke the same operations. However, it is a correct *closed* handler for all of the laws of *NDet* since in the closed setting both branches must be pure.

5 FUSING MODULAR HANDLERS

Throughout the section we assume two modular handlers $h_1 :: \mathit{MHandler}\ \Sigma_1\ c\ x\ y$ and $h_2 :: \mathit{MHandler}\ \Sigma_2\ d\ y\ z$ for some modular carriers c and d and types x, y, z. Their composition

$$\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1 :: \forall \mathit{sig}'.\ \mathit{Free}\ (\Sigma_1 + (\Sigma_2 + \mathit{sig}'))\ x \rightarrow \mathit{Free}\ \mathit{sig}'\ z$$

can interpret operations from $\Sigma_1 + \Sigma_2$ in syntax trees, but which theories does this transformation respect? This is the question that we answer in the rest of the paper.

The function $\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1$ can be more easily understood if we can find some handler $h_3 :: \mathit{MHandler}\ (\Sigma_1 + \Sigma_2)\ x\ z$ satisfying $\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1 = \mathit{handle}\ h_3 \cdot \mathit{assoc}$ where *assoc* and its inverse $assoc^{\circ}$ is the evident isomorphism between $Free\ (\Sigma_1 + (\Sigma_2 + \Sigma_3))$ and $Free\ ((\Sigma_1 + \Sigma_2) + \Sigma_3)$ for any Σ_1, Σ_2 and Σ_3. In this section, we show how this can be accomplished by *fold/build fusion* [Gill et al. 1993; Hinze et al. 2011] and continuation-passing style (CPS) transformation.

5.1 Carrier Fusion by CPS Transformation

The idea of fold/build fusion is that when we see an operation O_2 in the computation when running h_1, the modularity of h_1 guarantees that this operation will be handled later by h_2. Thus instead of leaving O_2 in the computation, we would like to handle it directly using $alg\ h_2$ in the fold of h_1, thus *fusing* the handling of h_1 and h_2 into *one* traversal over the computation tree. However, this idea does not directly work because the modular carrier for h_2 is only computed from the final result of h_1, and is not available in the stage of running h_1. Fortunately, this can be solved with CPS transformation as shown by Wu and Schrijvers [2015].

Given any type r, the *continuation monad with result type r* is

$$\textbf{newtype}\ Cont_r\ a = Cont\ \{runCont :: (a \to r) \to r\} \tag{16}$$

Intuitively, a computation of some a value in the continuation monad $Cont_r\ a \cong (a \to r) \to r$ does not directly return the value but feeds it to a given function $a \to r$ and *continues* the computation thereafter. The monad instance of $Cont_r$ is witnessed by:

$$\begin{array}{ll} return :: a \to Cont_r\ a & (\gg\!=) :: Cont_r\ a \to (a \to Cont_r\ b) \to Cont_r\ b \\ return\ x = Cont\ (\lambda k \to k\ x) & m \gg\!= f = Cont\ (\lambda k \to runCont\ m\ (\lambda x \to runCont\ (f\ x)\ k)) \end{array}$$

The pure computation $return\ x$ simply supplies x to the continuation. Monadic bind $m \gg\!= f$ runs m with a continuation that feeds the result x of m to f and runs f with the given continuation k, so bind is sequential composition. The continuation monad makes the final result type r explicit, and one can operate on the final result when it is not actually computed yet, which is demonstrated in the following minimal example.

Example 5.1. The following function $incrCont :: Cont_{Int}\ a \to Cont_{Int}\ a$ takes a computation in the continuation monad $Cont_{Int}$ and increments the integer that will be eventually computed.

$$incrCont\ m = Cont\ (\lambda k \to (runCont\ m\ k) + 1)$$

By definition, it satisfies that for any k, $runCont\ (incrCont\ m)\ k = (runCont\ m\ k) + 1$.

Back to the problem of fusing handlers, when running the first handler h_1, we can take the final result type r to be the carrier $d\ m$ of the second handler h_2, since $d\ m$ is what will be eventually computed from the result of handling h_1. Furthermore, when we see an operation handled by h_2, now we can let h_2 act on the result type $d\ m$ of the continuation monad in the same way as in Example 5.1. This is made precise by the following lemma.

Lemma 5.1. *Given any Σ_2-algebra, i.e. a function $alg :: \Sigma_2\ r \to r$, there is a Σ_2-algebra with carrier $(Cont_r\ a)$ for any type a by*

$$\begin{array}{l} liftAlgCont :: Functor\ \Sigma_2 \Rightarrow (\Sigma_2\ r \to r) \to \Sigma_2\ (Cont_r\ a) \to Cont_r\ a \\ liftAlgCont\ alg\ s = Cont\ (\lambda k \to alg\ (fmap\ (\lambda m \to runCont\ m\ k)\ s)) \end{array} \tag{17}$$

In particular, if $r = d\ m$, since $alg\ h_2 :: \Sigma_2\ (d\ m) \to d\ m$, then

$$liftAlgCont\ (alg\ h_2) :: \Sigma_2\ (Cont_{d\ m}\ a) \to Cont_{d\ m}\ a$$

provides a way to handle operations from Σ_2 using $Cont_{d\ m}\ a$.

Theorem 5.2 (Modular Carrier Fusion). *For any modular carriers c and d, the data type $c\ (Cont_{d\ m})$ for any m is also a modular carrier.*

Proof. First we note that there is a natural transformation from m to $Cont_{d\ m}$:

$$cps_{EM} :: (MCarrier\ d, Monad\ m) \Rightarrow m\ a \rightarrow Cont_{d\ m}\ a$$
$$cps_{EM}\ m = Cont\ (\lambda k \rightarrow fwd_d\ (fmap\ k\ m))$$

In fact cps_{EM} is a monad morphism because it preserves *return* and *join* following the laws of *fwd* (14). Then we can define the following *MCarrier* instance:

$$\textbf{newtype}\ Fused\ c\ d\ m = Fused\ \{unFused :: c\ (Cont_{d\ m})\}$$
$$\textbf{instance}\ (MCarrier\ c, MCarrier\ d) \Rightarrow MCarrier\ (Fused\ c\ d)\ \textbf{where}$$
$$\quad fwd = Fused \cdot fwd_c \cdot fmap\ unFused \cdot cps_{EM}$$

The required laws of *fwd* follow from the corresponding laws of c and d (Appendix E.1). □

5.2 Fused Modular Handlers

We intend to use $Fused\ c\ d$ as the modular carrier of the fused handler of h_1 and h_2, so it should carry both a Σ_1- and a Σ_2-algebra. Since $Fused\ c\ d \cong c\ (Cont_{d\ m})$ and $Cont_{d\ m}$ is a monad, $alg\ h_1$ can be used as the Σ_1-algebra for $Fused\ c\ d$. Also, the Σ_2-algebra $alg\ h_2 :: \Sigma_2\ (d\ m) \rightarrow d\ m$ can be lifted to $Fused\ c\ d$ in the following way:

$$liftAlgF :: (\Sigma_2\ (d\ m) \rightarrow d\ m) \rightarrow \Sigma_2\ (Fused\ c\ d\ m) \rightarrow Fused\ c\ d\ m$$
$$liftAlgF\ alg = Fused \cdot fwd_c \cdot liftAlgCont\ alg \cdot fmap\ (return \cdot unFused)$$

Theorem 5.3 (Handler Fusion). *For any modular handlers h_1 and h_2, it is the case that $handle\ h_2 \cdot handle\ h_1 = handle\ (h_2 \diamond h_1) \cdot assoc$ where assoc is the isomorphism between Free $(\Sigma_1 + (\Sigma_2 + sig'))$ and Free $((\Sigma_1 + \Sigma_2) + sig')$ and $h_2 \diamond h_1$ is defined as follows:*

$$(\diamond) :: (MCarrier\ c, MCarrier\ d, Functor\ \Sigma_1, Functor\ \Sigma_2)$$
$$\quad \Rightarrow MHandler\ \Sigma_2\ d\ y\ z \rightarrow MHandler\ \Sigma_1\ c\ x\ y \rightarrow MHandler\ (\Sigma_1 + \Sigma_2)\ (Fused\ c\ d)\ x\ z$$
$$h_2 \diamond h_1 = MHandler\ \{gen = Fused \cdot gen\ h_1, alg = alg_F, run = run_F\}\ \textbf{where}$$
$$\quad alg_F\ (Inl\ op) = Fused\ (alg\ h_1\ (fmap\ unFused\ op))$$
$$\quad alg_F\ (Inr\ op) = liftAlgF\ (alg\ h_2)\ op$$
$$\quad run_F\ x = run\ h_2\ (runCont\ (run\ h_1\ (unFused\ x))\ (gen\ h_2))$$

Proof sketch. We use the technique by Wu and Schrijvers [2015] to fuse $handle\ h_2 \cdot handle\ h_1$ into one function and show that the result is equivalent to $handle\ (h_2 \diamond h_1)$. A detailed proof can be found in Appendix B. □

It is revealing to compare *liftAlgF* with the *forward* function (15) of modular handlers. Ignoring the isomorphisms *Fused* and *unFused*, we can see that the *Op* in (15) that forwards an operation call is replaced by *liftAlgCont alg*, which is exactly the idea of fold/build fusion.

Corollary 5.4. *Let h_1 and h_2 be modular handlers of signatures Σ_1 and Σ_2 respectively and T be any theory of signature $\Sigma_1 + \Sigma_2$. The function $handle\ h_2 \cdot handle\ h_1 \cdot assoc^{\circ}$ is a correct open (or closed) transformation for T if $h_2 \diamond h_1$ is a correct open (or closed) handler of T.*

Proof. By Theorem 5.3, $handle\ h_2 \cdot handle\ h_1 \cdot assoc^{\circ} = handle\ (h_2 \diamond h_1)$. Then by Theorem 4.1, $handle\ (h_2 \diamond h_1)$ is correct for T if $h_2 \diamond h_1$ is correct for T. □

Corollary 5.4 is our main tool to reason about composed transformation $handle\ h_2 \cdot handle\ h_1$ because the correctness of $h_2 \diamond h_1$ is spelled by $alg\ (h_2 \diamond h_1)$ (Definition 4.3), which is much simpler for calculation than $handle\ h_2 \cdot handle\ h_1$, a composite of two *fold*'s. As the first application, we show that $handle\ h_2 \cdot handle\ h_1$ respects equations that are respected by h_1 and h_2 separately.

Theorem 5.5 (Preservation of Equations). *Suppose h_1 and h_2 are modular handlers of signatures Σ_1 and Σ_2 respectively. If h_1 and h_2 are correct open (resp. closed) handlers of T_1 :: Theory Σ_1 and T_2 :: Theory Σ_2 correspondingly, then $h_2 \diamond h_1$ is a correct open (resp. closed) handler of $T_1 + T_2$.*

Proof sketch. By Definition 2.3, an equation in $T_1 + T_2$ is either an equation from T_1 or an equation from T_2. In either case, it can be showed that *alg* $(h_2 \diamond h_1)$ respects the equation. Appendix D contains a detailed proof. □

Remark 5.1. This theorem justifies the modularity of modular handlers in [Schrijvers et al. 2019] to a greater extent: when two modular handlers are composed together, operations from both theories are handled and equations from both theories are preserved, which is a property not true for non-modular handlers (Example 3.4).

Remark 5.2. If $h_2 \diamond h_1$ is correct (open or closed) for some theory T, then equations in T are automatically term congruences under *handle* $(h_2 \diamond h_1)$ (and thus *handle* $h_2 \cdot$ *handle* h_1), since relation $\sim_T$ (Definition 3.1) contains the congruence rule Cong and Theorem 4.1 shows that *handle* $(h_2 \diamond h_1)$ respects relation $\sim_T$.

5.3 Clauses of Fused Handlers

Before we use $\diamond$ to reason about more interactions of handlers, we calculate some bookkeeping lemmas that characterise the handling action of $h_2 \diamond h_1$ on operations from the first and the second theories respectively.

Definition 5.1 (Clauses). Let h be any modular handler with modular carrier C. For any operation $O :: P \rightsquigarrow A$ in Σ, we call the following function the *clause* for O of h:

$$\begin{aligned}&c :: \mathit{Monad}\ m \Rightarrow P \to (A \to C\ m) \to C\ m\\&c\ p\ k = \mathit{alg}\ h\ (O\ p\ k)\end{aligned}$$

Lemma 5.6. *Let h_1 and h_2 be two modular handlers with modular carriers C and D respectively, and c_1 be the clause of h_1 for $O_1 :: P_1 \rightsquigarrow A_1$ and c_2 be the clause of h_2 for $O_2 :: P_2 \rightsquigarrow A_2$. Then the clause for O_1 of $h_2 \diamond h_1$ is*

$$\begin{aligned}&\overline{c_1} :: \mathit{Monad}\ m \Rightarrow P_1 \to (A_1 \to \mathit{Fused}\ C\ D\ m) \to \mathit{Fused}\ C\ D\ m\\&\overline{c_1}\ p_1\ k = \mathit{Fused}\ (c_1\ p_1\ (\mathit{unFused} \cdot k))\end{aligned}$$

and the clause for O_2 of $h_2 \diamond h_1$ is

$$\begin{aligned}&\overline{c_2} :: \mathit{Monad}\ m \Rightarrow P_2 \to (A_2 \to \mathit{Fused}\ C\ D\ m) \to \mathit{Fused}\ C\ D\ m\\&\overline{c_2}\ p_2\ k = \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p_2\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2))))))\end{aligned} \tag{18}$$

where binder t has type $C\ (\mathit{Cont}_{D\ m}) \to D\ m$ and fwd is the following instance:

$$\mathit{fwd} :: \mathit{Cont}_{D\ m}\ (C\ (\mathit{Cont}_{D\ m})) \to C\ (\mathit{Cont}_{D\ m})$$

This lemma can be calculated from the definition of *alg* $(h_2 \diamond h_1)$ (Appendix E.2). It is useful to simplify $\overline{c_2}$ from Lemma 5.6 further for specific modular carriers:

Lemma 5.7. *Assume the data in Lemma 5.6. When the modular carrier of h_1 is FreeEM Y for some type Y, (18) is equal to*

$$\overline{c_2}\ p_2\ k = \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_2\ q)))) \tag{19}$$

where $k' = \mathit{runCont} \cdot \mathit{unFreeEM} \cdot \mathit{unFused} \cdot k$. And when the modular carrier of h_1 is StateC S Y for some types S and Y, (18) is equal to

$$\overline{c_2}\ p_2\ k = \mathit{Fused}\ (\mathit{StateC}\ (\lambda s \to \mathit{Cont}\ (\lambda q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_2\ s\ q))))$$

where $k'\ a_2\ s = \mathit{runCont}\ (\mathit{unStateC}\ (\mathit{unFused}\ (k\ a_2))\ s)$.

The proof for this lemma is straightforward calculation based on the definitions of *fwd* for *FreeEM* and *StateC* (see Appendix E.2 for details).

Let h_1 and h_2 be correct (open or closed) handlers of theory T_1 and T_2 respectively. With Corollary 5.4 and Lemma 5.7, we can synthesise a sufficient condition for $\mathit{handle}\ h_1 \cdot \mathit{handle}\ h_2$ to be correct for any combination of T_1 and T_2: given any equation $L = R$ involving operations from T_1 and T_2, we substitute $\overline{c_1}$ for each operation O_1 in $L = R$ that comes from T_1 and substitute $\overline{c_2}$ for each operation O_2 that comes from T_2. Then we get an equation holds if and only if $h_2 \diamond h_1$ is correct for this equation by Definition 4.3, and this condition is solely characterised by the clauses of h_1 and h_2 for relevant operations appearing in this equation, rather than involving the whole handler. In the following sections, we apply this method to the commutative and distributive combinations of theories and study the correctness of the composite of some common handlers.

6 REASONING ABOUT COMMUTATIVE INTERACTION

In this section we apply the techniques developed in Section 5 to the tensor (Definition 2.4) of effect theories. We obtain a condition (20) on the clause of h_1 for O_1 and the clause of h_2 for O_2 such that operations O_1 and O_2 are commutative under the composite handler $\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1$. Then we use this result to study the interactions between some common handlers, specifically the handlers of mutable state, nondeterminism and the writer effect.

Theorem 6.1. *Given* $T_1 :: \mathit{Theory}\ \Sigma_1$ *and* $T_2 :: \mathit{Theory}\ \Sigma_2$ *and* $h_1 :: \mathit{MHandler}\ \Sigma_1\ C\ X\ W$ *and* $h_2 :: \mathit{MHandler}\ \Sigma_2\ D\ W\ Z$*, if* h_1 *and* h_2 *are correct open (or closed) handlers of* T_1 *and* T_2 *respectively, a sufficient condition for* $h_2 \diamond h_1$ *to be a correct open (or closed) handler of the tensor* $T_1 \otimes T_2$ *is: for any* $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ *and* $O_2 :: P_2 \rightsquigarrow_{\Sigma_2} A_2$*, letting* c_1 *be the clause for* O_1 *of* h_1 *and* c_2 *be the clause for* O_2 *of* h_2 *as in Definition 5.1, it holds that*

$$\begin{aligned} & c_1\ p_1\ (\lambda a_1 \to \mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p_2\ (\lambda a_2 \to t\ (k\ a_1\ a_2))))) \\ = {} & \mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p_2\ (\lambda a_2 \to t\ (c_1\ p_1\ (\lambda a_1 \to k\ a_1\ a_2))))) \end{aligned} \tag{20}$$

for any $p_1 :: P_1$*,* $p_2 :: P_2$ *and* $k :: A_1 \to A_2 \to C\ (\mathit{Cont}_{D\ m})$ *for any monad* m *(or* $m = \mathit{Free}\ \mathit{Empty}$ *for closed correctness). In (20), binder* t *has type* $C\ \ (\mathit{Cont}_{D\ m}) \to D\ \ m$ *and* fwd *is the instance* $\mathit{fwd} :: \mathit{Cont}_{D\ m}\ (C\ (\mathit{Cont}_{D\ m})) \to C\ (\mathit{Cont}_{D\ m})$.

Proof. It directly follows from Definition 2.4 of the tensor and the characterisation of clauses for O_1 and O_2 of $h_2 \diamond h_1$ (Lemma 5.6): substituting $\overline{c_1}$ and $\overline{c_2}$ in Lemma 5.6 for $\overline{O_1}$ and $\overline{O_2}$ in Definition 2.4 results in (20). ◻

Since *FreeEM* and *StateC* cover almost all examples of modular handlers in practice, we specialise the theorem above to these two cases and obtain conditions easier to use.

Corollary 6.2. *When the modular carrier* C *of* h_1 *is* $\mathit{FreeEM}\ Y\ m$ *for some type* Y*, we can simplify (20) with Lemma 5.7. Define* c_1' *and* k' *as follows to unwrap the constructors:*

$$\begin{aligned} & c_1' :: P_1 \to (A_1 \to (Y \to D\ m) \to D\ m) \to (Y \to D\ m) \to D\ m \\ & c_1'\ p\ a = \mathit{runCont}\ (\mathit{unFreeEM}\ (c_1\ p\ (\mathit{FreeEM} \cdot \mathit{Cont} \cdot a))) \\ & k' :: A_1 \to A_2 \to (Y \to D\ m) \to D\ m \\ & k'\ a_1\ a_2 = \mathit{runCont}\ (\mathit{unFreeEM}\ (k\ a_1\ a_2)) \end{aligned}$$

Then (20) is equivalent to

$$\begin{aligned} & c_1'\ p_1\ (\lambda a_1 \to (\lambda q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ q))) \\ = \quad & \lambda q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ q) \end{aligned} \tag{21}$$

where binder q has type $Y \to D\ m$.

Corollary 6.3. *When the modular carrier of h_1 is $StateC\ S\ Y\ m$, (20) can be simplified with the corresponding result of Lemma 5.7 too. Define c_1' and k' as follows to unwrap the constructors:*

$$\begin{aligned}
&c_1' :: P_1 \to (A_1 \to S \to (Y \to D\ m) \to D\ m) \to (S \to (Y \to D\ m) \to D\ m)\\
&c_1'\ p\ k = runCont \cdot unStateC\ (c_1\ p\ (\lambda a \to StateC\ (Cont \cdot k\ a)))\\
&k' :: A_1 \to A_2 \to S \to (Y \to D\ m) \to Y\\
&k'\ a_1\ a_2\ s = runCont\ (unStateC\ (k\ a_1\ a_2)\ s)
\end{aligned}$$

Then (20) can be simplified to

$$\begin{aligned}
&\quad c_1'\ p_1\ (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ s\ q))\\
=&\quad \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ s\ q)
\end{aligned} \tag{22}$$

where binder $q :: Y \to D\ m$.

6.1 Combining Nondeterminism and State

Theorem 6.4. *Handler $ndetH \diamond stH\ s$ is a correct closed handler of the tensor of $NDet$ and $State_s$.*

Proof. For each pair of $op_1 \in \{Get, Put\}$ and $op_2 \in \{Coin\}$ we verify that (22) holds. For $op_1 = Get$ and $op_2 = Coin$, we have

$$c_1'\ ()\ k = \lambda s \to k\ s\ s \tag{23}$$

Then we can establish (22) by plugging in c_1' and simplifying both sides:[1]

$$\begin{aligned}
&\ c_1'\ p_1\ (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ s\ q)) && \{\downarrow \text{ definition (23) of } c_1'\ \}\\
&= \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k'\ s\ a_2\ s\ q) && \{\uparrow \text{ definition (23) of } c_1'\ \}\\
&= \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ s\ q)
\end{aligned}$$

For $op_1 = Put$, $op_2 = Coin$ and any $p_1 :: s$, we have

$$c_1'\ p\ k = \lambda s \to k\ ()\ p \tag{24}$$

Accordingly, we calculate:

$$\begin{aligned}
&\ c_1'\ p_1\ (\lambda a_1 \to \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ s\ q)) && \{\downarrow \text{ definition (24) of } c_1'\ \}\\
&= \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to k'\ ()\ a_2\ p_1\ q) && \{\uparrow \text{ definition (24) of } c_1'\ \}\\
&= \lambda s\ q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ s\ q)
\end{aligned}$$

Since handlers *ndetH* and *stH* are correct closed handlers for *NDet* and $State_s$, we can conclude that $ndetH \diamond stH\ s$ is a *correct closed* handler of the tensor of nondeterminism and mutable state. □

Note that in the proof we did not rely on any property of c_2 or *ndetH*. In fact, we can strengthen the above proof to arbitrary handler h in place of *ndetH*.

Theorem 6.5. *Given a correct open (or closed) handler h of effect theory T, handler $h \diamond stH\ s$ is a correct open (or closed) handler of the tensor of T and the theory of mutable state.*

Remark 6.1. Pauwels et al. [2019] axiomatise the *local state semantics* of the combination of state and nondeterminism by the sum of $State_s$ and *NDet* with additionally two right-zero and right-distributive laws. Both of the additional laws can be derived from the equations of $State_s \otimes NDet$ and algebraicity (Appendix E.3). Thus $ndetH \diamond stH\ s$ is a correct (closed) handler of the local state semantics in [Pauwels et al. 2019].

[1] The arrows in the proof hints indicate the natural direction to read the calculation step.

By contrast, handling nondeterminism before state with $stH\ \ s \diamond ndetH$ will not validate the conditions of the corresponding Corollary 6.2. For example, if $op_1 = Coin$ and $op_2 = Put$, then

$$c_1'\ ()\ k = \lambda q \to k\ True\ (\lambda x \to k\ False\ (\lambda y \to q\ (x \cup y))) \tag{25}$$

$$c_2\ p_2\ k = StateC\ (\lambda s \to unStateC\ (k\ ())\ p_2) \tag{26}$$

The left-hand side of (21) becomes

$$\begin{aligned}
& c_1'\ p_1\ (\lambda a_1 \to \lambda q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ q)) && \{\Downarrow \text{ definition (26) of } c_2\ \} \\
= {} & c_1'\ p_1\ (\lambda a_1 \to \lambda q \to StateC\ (\lambda s \to unStateC\ (k'\ a_1\ ()\ q)\ p_2)) && \{\Downarrow \text{ definition (25) of } c_1'\ \} \\
= {} & \lambda q \to StateC\ (\lambda s \to unStateC\ (k'\ True\ ()\ (\lambda x \to \\
& \quad \boxed{StateC\ (\lambda s \to unStateC\ (k'\ False\ ()\ (\lambda y \to q\ (x \cup y)))\ p_2)}\ ))\ p_2)
\end{aligned}$$

and the right-hand side becomes:

$$\begin{aligned}
& \lambda q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ q) && \{\Downarrow \text{ definition (26) of } c_2\ \} \\
= {} & \lambda q \to StateC\ (\lambda s \to unStateC\ (c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ ())\ q)\ p_2) && \{\Downarrow \text{ definition (25) of } c_1'\ \} \\
= {} & \lambda q \to StateC\ (\lambda s \to unStateC\ (k'\ True\ ()\ (\lambda x \to \\
& \quad \boxed{k'\ False\ ()\ (\lambda y \to q\ (x \cup y))}\ ))\ p_2)
\end{aligned}$$

The boxed parts are the difference between both sides, making (21) not hold in general. The difference also matches our intuition: if nondeterminism is handled first, computation $\{\, b \leftarrow coin; put\ p_2; k\ b\}$ corresponding to the left-hand side is transformed to $\{\, put\ p_2; k\ True; put\ p_2; k\ False\}$ by $ndetH$, while computation $\{\, put\ p_2; b \leftarrow coin; k\ b\}$ corresponding to the right-hand side is transformed to $\{\, put\ p_2; k\ True; k\ False\}$. This explains why the boxed part of the left-hand side is $StateC\ (\lambda s \to RB\ p_2)$ where RB is the boxed part in the right-hand side.

Remark 6.2. Pauwels et al. [2019] axiomatise the *global state semantics* of the combination of state and nondeterminism by the sum of $State_s$ and $NDet$ in addition with the following *put-or law*:

$$(Put\ s\ (\lambda() \to m)) \sqcap n = Put\ s\ (\lambda() \to m \sqcap n)$$

It is not difficult to show that $stH\ \ s \diamond ndetH$ is a correct open handler for this law using Lemma 5.7 (Appendix E.4), and thus it is a correct closed handler of the global state semantics.

6.2 Combining State and Writer

For another example, we prove that handling the writer effect and mutable state in either order is a correct handler of their tensor. The writer effect $Writer\ w$ is parameterised by a monoid w with unit $mempty$ and operation $\diamond$, and it has one operation $Tell :: w \rightsquigarrow ()$ with an *accumulation law*:

$$Tell\ w_1\ (Tell\ w_2\ k) = Tell\ (w_1 \diamond w_2)\ k$$

Writer effect can be handled by the following handler:

$$\begin{aligned}
& wtH :: Monoid\ w \Rightarrow MHandler\ (Writer\ w)\ (FreeEM\ (a, w))\ a\ (a, w) \\
& wtH = MHandler\ gen\ alg\ unFreeEM\ \mathbf{where} \\
& \quad gen\ a = FreeEM\ (return\ (a, mempty)) \\
& \quad alg\ (Tell\ w\ k) = FreeEM\ (\mathbf{do}\ (a, u) \leftarrow unFreeEM\ (k\ ()); return\ (a, w \diamond u))
\end{aligned}$$

It is straightforward calculation to verify that wtH is a correct open handler of the accumulation law (see Appendix E.5 for details).

Theorem 6.6. *Both $stH\ \ s \diamond wtH$ and $wtH \diamond stH$ are correct open handlers of the tensor of mutable state and writer.*

Proof sketch. Following Theorem 6.5, $wtH \diamond stH\ s$ is a correct open handler of the tensor, and Corollary 6.2 can be used to show that $stH\ s \diamond wtH$ is correct. A detailed calculation can be found in Appendix E.5. □

7 REASONING ABOUT DISTRIBUTIVE INTERACTION

We apply the technique so far to distributive tensor of effects (Definition 2.5) in this section. We present a condition similar to Theorem 6.1 on two modular handlers for their composite to be correct with respect to the distributive tensor of the sub-theories, and a specialised version similar to Corollary 6.2 when the modular carrier is *FreeEM*. Then we use the results to reason about the correctness of composing the handlers of nondeterministic and probabilistic choice with respect to the theory of combined choice discussed in Example 2.9.

Theorem 7.1. *Given $T_1 :: Theory\ \Sigma_1$ and $T_2 :: Theory\ \Sigma_2$ and modular handlers $h_1 :: MHandler\ \Sigma_1\ C\ X\ W$ and $h_2 :: MHandler\ \Sigma_2\ D\ W\ Z$, if h_1 and h_2 are correct open (or closed) handlers of T_1 and T_2 respectively, a sufficient condition for $h_2 \diamond h_1$ to be a correct open (or closed) handler of the distributive tensor $T_1 \triangleright T_2$ of T_1 over T_2 is: for any $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ and $O_2 :: P_2 \rightsquigarrow_{\Sigma_2} A_2$ of Σ_2, letting c_1 be the clause for O_1 of h_1 and c_2 be the clause for O_2 of h_2 as in Definition 5.1, it holds that*

$$\begin{array}{l} c_1\ p_1\ (\lambda a_1 \rightarrow \mathbf{if}\ a_1 \equiv u \\ \quad \mathbf{then}\ fwd\ (Cont\ (\lambda t \rightarrow \\ \qquad c_2\ p_2\ (\lambda a_2 \rightarrow t\ (y\ a_2)))) \\ \quad \mathbf{else}\ x\ a_1) \end{array} = \begin{array}{l} fwd\ (Cont\ (\lambda t \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow \\ \quad t\ (c_1\ p_1\ (\lambda a_1 \rightarrow \\ \qquad \mathbf{if}\ a_1 \equiv u\ \mathbf{then}\ y\ a_2 \\ \qquad\quad \mathbf{else}\ x\ a_1))))) \end{array} \tag{27}$$

for any $u :: A_1$, $p_1 :: P_1$, $p_2 :: P_2$, any monad m (or $m = Free\ Empty$ for closed correctness) and

$$x :: A_1 \rightarrow C\ (Cont_{D\ m}) \qquad y :: A_2 \rightarrow C\ (Cont_{D\ m})$$

Proof. By Definition 2.5, Lemma 5.6 and Definition 4.3, substituting $\overline{c_1}$ and $\overline{c_2}$ in Lemma 5.6 for O_1 and O_2 in Equation 12 results in (27). □

Corollary 7.2. *If the modular carrier of h_1 is $FreeEM\ Y$ for some type Y, by Lemma 5.7 the condition above can be simplified to*

$$\begin{array}{l} c_1'\ p_1\ (\lambda a_1 \rightarrow \\ \quad \mathbf{if}\ a_1 \equiv u\ \mathbf{then} \\ \qquad \lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow y'\ a_2\ q) \\ \quad \mathbf{else}\ x'\ a_1) \end{array} = \begin{array}{l} \lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow \\ \quad c_1'\ p_1\ (\lambda a_1 \rightarrow \\ \qquad \mathbf{if}\ a_1 \equiv u\ \mathbf{then}\ y'\ a_2\ \mathbf{else}\ x'\ a_1) \\ \qquad q) \end{array} \tag{28}$$

where c_1', x' and y' are the corresponding unprimed function with various data constructors unwrapped:

$$x' :: A_1 \rightarrow (Y \rightarrow D\ m) \rightarrow D\ m \qquad y' :: A_2 \rightarrow (Y \rightarrow D\ m) \rightarrow D\ m$$

$$c_1' :: P_1 \rightarrow (A_1 \rightarrow (Y \rightarrow D\ m) \rightarrow D\ m) \rightarrow ((Y \rightarrow D\ m) \rightarrow D\ m)$$

We can also specialise a version if the modular carrier of h_1 is *StateC*, but we leave it out for the sake of space.

7.1 Handling Combined Choice

Cheung [2017] shows that models of combined choice $Prob \triangleright NDet$ in Example 2.9 are exactly algebras of the geometrically convex monad (roughly speaking, the monad mapping a to the set of convex sets of distributions over a-elements), but it is not obvious if composing the standard handlers of the two theories gives rise to such a model, i.e. handling the distributive tensor correctly. In this subsection, we explore this question using Theorem 7.1.

A computation using probabilistic choice can be handled to a probability distribution of outcomes which we represent as functions **type** $Distr\ a = a \to Float$ which range in interval $[0, 1]$ and sums to 1 for all elements of a. Two distributions can be convexly combined by $+_f :: Distr\ a \to Distr\ a \to Distr\ a$ for any $f \in [0, 1]$:

$$p +_f q = \lambda x \to f * p\ x + (1 - f) * q\ x$$

Theory *Prob* (Example 2.9) can be closed-correctly handled by running both branches in sequence and convexly combine the results:

$$\begin{array}{l}
probH :: Eq\ a \Rightarrow MHandler\ Prob\ (FreeEM\ (Distr\ a))\ a\ (Distr\ a) \\
probH = MHandler\ gen\ alg\ unFreeEM\ \mathbf{where} \\
\quad gen\ a = FreeEM\ (return\ (\lambda x \to \mathbf{if}\ x \equiv a\ \mathbf{then}\ 1\ \mathbf{else}\ 0)) \\
\quad alg\ (PChoose\ p\ k) = FreeEM\ (\\
\qquad \mathbf{do}\ x \leftarrow unFreeEM\ (k\ True); y \leftarrow unFreeEM\ (k\ False); return\ (x +_p y))
\end{array}$$

In this section, we focus on the correctness of the composite handler $ndetH \diamond probH$ with respect to $Prob \triangleright NDet$. Since $probH$ has modular carrier $FreeEM\ (Distr\ a)$, we can try Corollary 7.2. The corresponding clauses for $\triangleleft p \triangleright$ and $\sqcap$ are

$$\begin{array}{l}
c_1'\ p\ k\ \ = \lambda q \to k\ True\ (\lambda x \to k\ False\ (\lambda y \to q\ (x +_p y))) \\
c_2\ ()\ k = FreeEM\ (\mathbf{do}\ \{ l_1 \leftarrow unFreeEM\ (k\ True); l_2 \leftarrow unFreeEM\ (k\ Flase)\ return\ (l_1 \cup l_2) \})
\end{array}$$

The left-hand side of (28) is

$$\begin{array}{l}
\quad c_1'\ p_1\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv u\ \mathbf{then}\ \lambda q \to c_2\ p_2\ (\lambda a_2 \to y'\ a_2\ q)\ \mathbf{else}\ x'\ a_1) \quad \{\downarrow \text{ definition of } c_2\ p_2 \} \\
= c_1'\ p_1\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv u\ \mathbf{then}\ \lambda q \to FreeEM\ (\mathbf{do}\ \{ l_1 \leftarrow unFreeEM\ (y'\ True\ q); \\
\qquad l_2 \leftarrow unFreeEM\ (y'\ False\ q); return\ (l_1 \cup l_2) \})\ \mathbf{else}\ x'\ a_1)
\end{array}$$

Let us consider the case $u = False$ first, which corresponds to the left distributivity $x \triangleleft p \triangleright (y \sqcap z)$. Setting $u = False$ and expanding $c_1'\ p_1$, the last equation becomes

$$\begin{array}{l}
\lambda q \to x'\ True\ (\lambda x \to FreeEM\ (\mathbf{do}\ \{ l_1 \leftarrow unFreeEM\ (y'\ True\ (\lambda y \to q\ (x +_{p_1} y))); \\
\quad l_2 \leftarrow unFreeEM\ (y'\ False\ (\lambda y \to q\ (x +_{p_1} y))); return\ (l_1 \cup l_2) \}))
\end{array} \tag{29}$$

Now from the right-hand side of Equation 28, we calculate:

$$\begin{array}{l}
\quad \lambda q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv u\ \mathbf{then}\ y'\ a_2\ \mathbf{else}\ x'\ a_1)\ q) \\
\qquad \{\downarrow \text{ definition } c_1'\ p_1 \} \\
= \lambda q \to c_2\ p_2\ (\lambda a_2 \to x'\ True\ (\lambda x \to y'\ a_2\ (\lambda y \to q\ (x +_p y)))) \\
\qquad \{\downarrow \text{ definition of } c_2\ p_2 \} \\
= \lambda q \to FreeEM\ (\mathbf{do}\ l_1 \leftarrow unFreeEM\ (x'\ True\ (\lambda x \to y'\ True\ (\lambda y \to q\ (x +_{p_1} y)))) \\
\qquad\qquad\qquad\qquad l_2 \leftarrow unFreeEM\ (x'\ True\ (\lambda x \to y'\ False\ (\lambda y \to q\ (x +_{p_1} y)))) \\
\qquad\qquad\qquad\qquad return\ (l_1 \cup l_2))
\end{array} \tag{30}$$

It is not difficult to see (29) $\neq$ (30) for arbitrary monad m in general, which matches our intuition: under $ndetH \diamond probH$, computation $x \triangleleft p \triangleright (y \sqcap z)$ executes x once but $(x \triangleleft p \triangleright y) \sqcap (x \triangleleft p \triangleright z)$ executes x twice. Thus $ndetH \diamond probH$ is not a correct open handler of $Prob \triangleright NDet$, but is it a correct closed handler of $Prob \triangleright NDet$? When m is the identity monad $Free\ Empty$, the **do**-notations in (29) and

(30) degenerates to **let**-bindings, and (29) = (30) is equivalent to

$$\begin{array}{l}\lambda q \to \\ \quad x'\ \mathit{True}\ (\lambda x \to \\ \quad\quad \mathbf{let}\ l_1 = y'\ \mathit{True}\ (\lambda y \to q\ (x +_{p_1} y)) \\ \quad\quad\quad\ \ l_2 = y'\ \mathit{False}\ (\lambda y \to q\ (x +_{p_1} y)) \\ \quad\quad \mathbf{in}\ l_1 \cup l_2)\end{array} \quad = \quad \begin{array}{l}\lambda q \to \mathbf{let}\ l_1 = x'\ \mathit{True}\ (\lambda x \to \\ \quad\quad\quad\quad\quad y'\ \mathit{True}\ (\lambda y \to q\ (x +_{p_1} y))) \\ \quad\quad\quad\quad l_2 = x'\ \mathit{True}\ (\lambda x \to \\ \quad\quad\quad\quad\quad y'\ \mathit{False}\ (\lambda y \to q\ (x +_{p_1} y))) \\ \quad\quad\ \ \mathbf{in}\ l_1 \cup l_2\end{array} \tag{31}$$

where $x', y' :: \mathit{Bool} \to (\mathit{Distr}\ A \to \mathit{Set}\ (\mathit{Distr}\ A)) \to \mathit{Set}\ (\mathit{Distr}\ A)$. However, (31) still does not hold in general. Thus our attempt with Corollary 7.2 seems inconclusive. However, with a closer look we notice that the functions x' and y' bear some properties not manifested in their types: they correspond to handled subterms of the computation, and therefore they must be built from $\mathit{gen}\ (\mathit{ndetH} \diamond \mathit{probH})$ and $\mathit{alg}\ (\mathit{ndetH} \diamond \mathit{probH})$. Indeed, if $f :: (\mathit{Distr}\ A \to \mathit{Set}\ (\mathit{Distr}\ A)) \to \mathit{Set}\ (\mathit{Distr}\ A)$ is built from $\mathit{gen}\ (\mathit{ndetH} \diamond \mathit{probH})$ and $\mathit{alg}\ (\mathit{ndetH} \diamond \mathit{probH})$, then it satisfies

$$f\ (\lambda x \to g\ x \cup h\ x) = f\ g \cup f\ h \tag{32}$$

and (32) for $f = x'\ \mathit{True}$ implies (31).

7.2 Generalising the Continuation Monad

Note that $\mathit{Set}\ (\mathit{Distr}\ A)$ with join operation $\cup$ is a semi-lattice, and for any set X, functions $X \to \mathit{Set}\ (\mathit{Distr}\ A)$ can be equipped with a semi-lattice structure with the join operation defined pointwise: for any $g, h :: X \to \mathit{Set}\ (\mathit{Distr}\ A)$,

$$g \cup h = \lambda x \to g\ b \cup h\ x$$

Then (32) states that f is a join-preserving mapping, i.e. an arrow in the category SL of semi-lattice. It is a standard result that there is an adjunctive bijection for any semi-lattices A, B, and set X

$$\mathit{SL}^{op}(B^X, A) \cong \mathit{Set}(X, \mathit{SL}(A, B))$$

where $\mathit{SL}^{op}(B^X, A)$ is the set of join-preserving functions from semi-lattice A to B^X and $\mathit{SL}(A, B)$ is the set of of join-preserving functions from A to B, and $\mathit{Set}(X, Y)$ is the set of functions from X to Y for any X and Y. Consequently, this adjunction gives rise to a monad on Set mapping X to the set $\mathit{SL}(B^X, B)$ for any semi-lattice B. Then replacing Cont in the construction of Fused in Theorem 5.2 with this monad allows us to prove (31) and thus $\mathit{ndetH} \diamond \mathit{probH}$ is a correct closed handler of the theory $\mathit{Prob} \triangleright \mathit{NDet}$ of combined choice.

More generally, for any category C with *powers* [Mac Lane 1998, p.70], there is an adjunction

$$C^{op}(B^X, A) \cong \mathit{Set}(X, C(A, B))$$

and monad $X \mapsto C(B^X, B)$ for any object B in C [Hinze 2012, p.344]. When C is Set, it is exactly the continuation monad. Some other instances are studied in the context of categorical semantics of predicate transformers [Hino et al. 2016; Jacobs 2017]. Similar to the situation of combined choice where we need $C = \mathit{SL}$, in some applications we may need to choose appropriate C to reflect the *invariants* in the handled computations that are preserved by the clauses of the handler to prove the correctness of composite handlers. We leave a systematic study of this extension as future work.

8 RELATED WORK

Combinations of Effects. Hyland et al. [2006] study the sum and tensor of computational effects and show that the sum with the theory of exceptions and interactive IO, and the tensor with mutable state lead to the corresponding monad transformers, and later Cheung [2017] follows this line of research and studies the distributive tensor of effect theories, in particular, the connection with the distributive laws of monads and the example of combining nondeterminism and probabilistic choice. Their work gives a unified account of modularity for computational effects and our work aims to connect this modularity with the modularity of handlers.

Effect Handlers. In the original work on effect handlers [Plotkin and Pretnar 2009, 2013], a global effect theory is assumed throughout the language. To avoid the interdependence of typing handlers and proving them correct, [Plotkin and Pretnar 2009] provides two calculi (one for defining handlers and one for using them) and, accordingly, two equational logics extending the logic in [Plotkin and Pretnar 2008] (one for proving handlers correct and the other for reasoning about computations using handlers). The later work [Plotkin and Pretnar 2013] adopts a simpler approach by leaving semantics of incorrect (though well-typed) handlers undefined. In comparison, in this paper handlers interpret signatures instead of theories, so correctness respecting theories becomes an extrinsic property of handlers.

Because many practically useful handlers do not respect the standard theories of their effects and fundamentally the correctness of handlers is undecidable [Plotkin and Pretnar 2013], most later work (with the exceptions [Ahman 2017; Kiselyov et al. 2021; Lukšič and Pretnar 2020]) on effect handlers only considers effect theories with no equations, resulting in fewer reasoning principles for algebraic effects and consequently weaker guarantee of correctness.

Ahman [2017] presents a dependently typed language in which handlers (and proofs showing their satisfaction of the equations of the theory) are represented as user-defined *algebra types* and applying handlers is done using sequential composition. With the power of dependent types, [Ahman 2017] uses handlers to define predicates on effectful computations.

Lukšič and Pretnar [2020] presents a type system in which computation types are tagged with a set of equations expected to hold. In fact, we have informally followed their ideas in our treatment of effects and handlers: computations are interpreted by free monads ignoring the equations, and equations are separately interpreted as a relation, and their judgement of handlers respecting theories corresponds to our Definition 4.3 of correct handlers. Another difference is that [Lukšič and Pretnar 2020] only considers closed handlers, whereas we consider both open and closed ones.

Kiselyov et al. [2021] advocate a different philosophy about the relationship between equations and handlers—they advocate that equations should *be distilled from* handlers rather than *specify handlers a prior*. They also study the equations respected by the handlers of state, nondeterminism and their composites. However, from either viewpoint, the eventual proof obligation is the same—an equation is respected by a handler. Thus the results developed in this paper for proving a composite handler respecting some equation are applicable in their setting too. They also emphasise that equational laws should be term congruences under a handler, which is reflected by the Cong rule in our definition of equivalent computations $\sim_T$ (Definition 3.1). Our restriction of modularity is reminiscent of the restriction in [Kiselyov et al. 2021] that operations must be uniquely handled by the concerned handler in their formulation of *equivalence modulo handlers*.

Zhang and Myers [2019] present an operational semantics for effect polymorphism based on *tunneling* in which the parametricity theorem holds for effect polymorphic functions. In this paper, effect polymorphism is achieved by being polymorphic in the signature functor, utilising the polymorphic mechanisms of Haskell. Since our results crucially rely on the parametricity of polymorphic abstractions, we expect our results only to hold for languages with proper effect parametricity such as the one in [Brachthäuser et al. 2020; Zhang and Myers 2019].

Schrijvers et al. [2019] introduce *modular handlers* that play an essential role in this paper. They also compare modular handlers to monad transformers, showing that the expressibility of modular handlers and monad transformers implementing only algebraic operations are equivalent in Haskell. However, the equal expressibility crucially depends on the features present in the language, as demonstrated by Forster et al. [2017] that there is no type-preserving translation from effect handlers to layered monads [Filinski 1999] in a call-by-push-value calculus without polymorphism and inductive types. In [Schrijvers et al. 2019], equations of algebraic theories are not considered, which we recover in this paper. We also formalise notions of the correctness of modular handlers and study the correctness of composite modular handlers using handler fusion.

Xie et al. [2020] introduce the *scoped-resumption* restriction on handlers to simplify reasoning and aid optimisation, while we impose the *modular* restriction for a similar purpose. Indeed, their non-scoped example in [Xie et al. 2020, Section 2.2] can be rejected by the modular restriction too. However, they check scoped resumptions dynamically, whereas modular handlers are statically typed. It is interesting future work to establish the relationship between these two restrictions.

The techniques developed in this paper only apply to modular handlers. However, not all handlers in the various languages discussed above are modular. A rough criterion is that a handler is modular as long as it does not use its resumption in any way other than invoking it. In particular, it cannot apply the handling construct on its resumption. For languages implementing effect polymorphism such as Koka [Leijen 2017], this condition is a consequence of a handler being polymorphic in unhandled operations. For languages without effect polymorphism such as [Bauer and Pretnar 2014; Plotkin and Pretnar 2009], this is not automatically guaranteed. Appendix F shows a fine-grained call-by-value calculus of handlers in which all handlers must be modular. Although most handlers appearing in the literature are modular, there is an example of non-modular handlers of mutable state by handling the get operation in the clause of put operation in [Biernacki et al. 2017, page 4]. We leave extending our work to non-modular handlers as future work.

CPS Transformations. There is a lot of work on using CPS transformations to optimise effectful programs. Here we discuss some typical ones in the context of algebraic effects and handlers and compare them with the transformation that we use for fusing handlers.

Voigtländer [2008] shows that CPS transformation of free monads with the codensity monad [Hinze 2012] gives an asymptotic improvement on the time complexity of monadic binding operations. Kammar et al. [2013] use CPS transformations based on the codensity and continuation monads in their implementations of effect handlers, in which the continuation monad is iterated to allow the operations in a computation to be handled by different *open handlers*, a concept that we borrow and use in this paper. Schuster et al. [2020] translate effectful programs written in capability-passing style into iterated continuation passing style. They also statically specialise the abstract capabilities in a CPS translated program to corresponding concrete handlers by translating to a two-stage simply typed lambda calculus, and thus eliminate all handling constructs in the translation result.

Compared to these works that apply CPS transformations to computations for performance improvement, this paper uses CPS transformation on handlers instead of computations, and the purpose is mostly for reasoning about handlers. Despite different motivations, the techniques of CPS transformation are similar, and we believe that it is possible to devise a handling-eliminating translation similar to the one given in [Schuster et al. 2020] if we iteratively fuse all handlers using our fusion combinator and inline the resulting all-in-one handler into a computation.

Our handler fusion is directly inspired by the work by Wu and Schrijvers [2015] with the minor difference that we use the continuation monad instead of the codensity monad for CPS transformation, and they rely on the compiler to perform static fusion, whereas our fusion combinator explicitly gives the result of fusion when the handlers are defined in the form of modular handlers. Similar fusion technique is also used in [Seynaeve et al. 2020] to eliminate intermediate lists when implementing nondeterminism with mutable state.

9 CONCLUSION

This paper has studied a way to reason about the semantics of sequentially composed handlers by fusing them into one, which allows us to derive relatively simple conditions for the semantics of the composite handler to agree with any combination of the effect theories separately handled. With this connection between modular specifications (effect theories) of effects and modular implementations (handlers) of effects, programmers are furnished with a principled way to determine the right order of composing handlers for their need by equational reasoning, as demonstrated in several case studies. The following directions can be explored in the future:

- We wish to find a concise categorical formulation of modular carriers and handlers, so that the techniques in this paper can be generalised to categories other than the category of sets.
- Our equational proofs in this paper are done in a paper-and-pencil way. It will be useful to find a way to formalise them with reasonable effort and even automate them.
- As demonstrated in Section 7.2, the continuation monad used for fusion needs to be generalised in some cases. We wish to find more examples of this and make a systematic study.
- The fusion combinator of modular handlers can possibly be used to implement a compiler of effect handlers that fuses all handlers statically known and inline them into computations.

REFERENCES

Danel Ahman. 2017. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158095

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? arXiv:cs.LO/1807.05923 https://arxiv.org/abs/1807.05923

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (Dec 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

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158096

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* 30 (2020), e8. https://doi.org/10.1017/S0956796820000027

Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

Kwok-Ho Cheung. 2017. *Distributive interaction of algebraic effects.* Ph.D. Dissertation. University of Oxford.

Andrzej Filinski. 1999. Representing Layered Monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/292540.292557

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110257

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP '11* (2011), 2. https://doi.org/10.1145/2034773.2034777

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

W. Hino, H. Kobayashi, I. Hasuo, and B. Jacobs. 2016. Healthiness from Duality. *Proceedings - Symposium on Logic in Computer Science* 05-08-July (2016), 1–13. https://doi.org/10.1145/2933575.2935319 arXiv:arXiv:1605.00381v1

Ralf Hinze. 2012. Kan Extensions for Program Optimisation Or: Art and Dan Explain an Old Trick. In *Mathematics of Program Construction*, Jeremy Gibbons and Pablo Nogueira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–362. https://doi.org/978-3-642-31113-0_16

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

C. A. R. Hoare. 1985. A Couple of Novelties in the Propositional Calculus. *Mathematical Logic Quarterly* 31, 9-12 (1985), 173–178. https://doi.org/10.1002/malq.19850310905

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining Effects: Sum and Tensor. *Theor. Comput. Sci.* 357, 1 (July 2006), 70–99. https://doi.org/10.1016/j.tcs.2006.03.013

Bart Jacobs. 2017. A Recipe for State-and-Effect Triangles. *Logical Methods in Computer Science* 13 (03 2017). https://doi.org/10.23638/LMCS-13(2:6)2017

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/2500365.2500590

Oleg Kiselyov, Shin-cheng Mu, and Amr Sabry. 2021. Not by equations alone: Reasoning with extensible effects. *Journal of Functional Programming* 31 (2021), e2. https://doi.org/10.1017/S0956796820000271

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Paul Blain Levy. 2003. *Call-By-Push-Value*. Number 1. Springer Netherlands. https://doi.org/10.1007/978-94-007-0954-6

Žiga Lukšič and Matija Pretnar. 2020. Local algebraic effect theories. *Journal of Functional Programming* 30 (2020). https://doi.org/10.1017/s0956796819000212

Saunders Mac Lane. 1998. *Categories for the Working Mathematician, 2nd edn.* Springer, Berlin. https://doi.org/10.1007/978-1-4757-4721-8

Michael Mislove, Joël Ouaknine, and James Worrell. 2004. Axioms for Probability and Nondeterminism. *Electronic Notes in Theoretical Computer Science* 96 (2004), 7 – 28. https://doi.org/10.1016/j.entcs.2004.04.019 Proceedings of the 10th International Workshop on Expressiveness in Concurrency.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. 2019. Handling Local State with Global State. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Springer International Publishing, Cham, 18–44. https://doi.org/10.1007/978-3-030-33636-3_2

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures*, Mogens Nielsen and Uffe Engberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 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 Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *Electr. Notes Theor. Comput. Sci.* 73 (10 2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

G. Plotkin and M. Pretnar. 2008. A Logic for Algebraic Effects. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 118–129. https://doi.org/10.1109/LICS.2008.45

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec 2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 98–113. https://doi.org/10.1145/3331545.3342595

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408975

Willem Seynaeve, Koen Pauwels, and Tom Schrijvers. 2020. State Will do. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 204–225.

Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

Janis Voigtländer. 2009. Free theorems involving type constructor classes. *ACM SIGPLAN Notices* 44, 9 (2009), 173–184. https://doi.org/10.1145/1631687.1596577

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 302–322. https://doi.org/978-3-319-19797-5_15

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. *Proc. ACM Program. Lang.* 4, ICFP, Article 99 (Aug. 2020), 29 pages. 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, Article 5 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290318

A EQUATIONS IN CHURCH ENCODINGS

In the appendices, we switch to work with equations based on *Church encodings*, which are equivalent to the definition of equations Section 2.1 but simply the proofs.

Definition A.1 (Equations in Chuch encodings)**.** Given a signature Σ, types Γ and v, an *equation in Church encodings* for Σ with free value variables Γ and free computation variables v is a pair of *templates*:

$$\begin{aligned}
&\textbf{data}\ \mathit{Equation}_C\ \Sigma\ \Gamma\ v = \mathit{Eqn}_C\ (\mathit{Template}\ \Sigma\ \Gamma\ v)\ (\mathit{Template}\ \Sigma\ \Gamma\ v)\\
&\textbf{type}\ \mathit{Template}\ \Sigma\ \Gamma\ v\ \ = \forall c.\ (\Sigma\ c \to c) \to \Gamma \to (v \to c) \to c
\end{aligned}$$

Moreover, we say that an algebra $alg :: \Sigma\ c \to c$ respects an equation $(\mathit{Eqn}_C\ lhs\ rhs) :: \mathit{Equation}_C\ \Sigma\ \Gamma\ v$ if for any $t :: \Gamma$ and $k :: v \to c$,

$$lhs\ alg\ t\ k = rhs\ alg\ t\ k$$

Lemma A.1. *There is an isomorphism between equations based on free monads (11) and equations based on Church encodings.*

$$\phi :: \mathit{Functor}\ \Sigma \Rightarrow \mathit{Equation}\ \Sigma\ \Gamma\ v \to \mathit{Equation}_C\ \Sigma\ \Gamma\ v$$

Moreover, an algebra alg respects an equation in Church encodings if and only if it respects the isomorphic equation in free monads (Definition 2.1).

Proof. The isomorphism can defined as follows:

$$\phi\ (l \doteq r) = \mathit{Eqn}_C\ (\lambda alg\ t\ k \to \mathit{fold}\ k\ alg\ (l\ t))\ (\lambda alg\ t\ k \to \mathit{fold}\ k\ alg\ (r\ t))$$

and its inverse is

$$\begin{aligned}
&\phi^{\circ} :: \mathit{Functor}\ \Sigma \Rightarrow \mathit{Equation}_C\ \Sigma\ \Gamma\ v \to \mathit{Equation}\ \Sigma\ \Gamma\ v\\
&\phi^{\circ}\ (\mathit{Eqn}_C\ l\ r) = (\lambda t \to l\ \mathit{Op}\ t\ \mathit{Var}) \doteq (\lambda t \to r\ \mathit{Op}\ t\ \mathit{Var})
\end{aligned}$$

We refer the reader to [Hinze 2005] for the proof that ϕ and ϕ° form a pair of isomorphism. An algebra respects an equation in Church encodings iff it respects the isomorphic equation in free monads following the definitions of equation respecting and ϕ. □

B PROOF OF HANDLER FUSION

In this section we prove Theorem 5.3. The technique is essentially the same as the one in [Wu and Schrijvers 2015], although the setting in their paper is slightly different from the setting of modular handlers used in this paper.

Definition B.1. For convenience in our calculation, we divide *handle* in Definition 3.3 into the following smaller functions:

$$\begin{aligned}
&\mathit{split} :: (\Sigma_1\ c \to c) \to (\Sigma_2\ c \to c)\\
&\qquad \to (\Sigma_1 + \Sigma_2)\ c \to c\\
&\mathit{split}\ alg_1\ alg_2\ (\mathit{Inl}\ x) = alg_1\ x\\
&\mathit{split}\ alg_1\ alg_2\ (\mathit{Inr}\ x) = alg_2\ x
\end{aligned}$$

$$\begin{aligned}
&\mathit{openAlg} :: (\mathit{Functor}\ sig', \mathit{MCarrier}\ c)\\
&\qquad \Rightarrow \mathit{MHandler}\ \Sigma\ c\ a\ b\\
&\qquad \to (\Sigma + sig')\ (c\ (\mathit{Free}\ sig')) \to c\ (\mathit{Free}\ sig')\\
&\mathit{openAlg}\ h = \mathit{split}\ (alg\ h)\ \mathit{forward}
\end{aligned}$$

It it clear that

$$\mathit{handle}\ h = \mathit{run}\ h \cdot \mathit{fold}\ (\mathit{gen}\ h)\ (\mathit{openAlg}\ h) \tag{33}$$

B.1 Fold/Build fusion for free

The underlying technique for our proof is *fold/build fusion for free* introduced by [Hinze et al. 2011]. In this subsection, we state it in our context of free monads (Theorem B.1) and establish relevant prerequisites (mainly Lemma B.5) to apply the free theorem.

Definition B.2. We say that a monad M is a term monad of signature Σ if there is a parametric family of Σ-algebras $con :: \forall a.\ \Sigma\ (M\ a) \to M\ a$:

$$\begin{array}{l}\textbf{class}\ (Monad\ m, Functor\ \Sigma) \Rightarrow TermMonad\ m\ \Sigma\ \textbf{where}\\ \quad con :: \Sigma\ (m\ a) \to m\ a\end{array}$$

with the *algebraicity* law:

$$con\ op \gg\!\!= k = con\ (fmap\ ({\gg\!\!=} k)\ op)$$

Definition B.3. Given two term monads M_1 and M_2 of the same signature Σ, a term monad morphism from M_1 to M_2 is a monad morphism $f :: \forall a.\ M_1\ \ a \to M_2\ \ a$ from M_1 to M_2 that is simultaneously a Σ-algebra homomorphism from con_{M_1} to con_{M_2} for any a.

The parametricity of polymorphic functions entails the following property.

Theorem B.1 (Fusion for Free). *For any function $g :: TermMonad\ m\ \Sigma \Rightarrow X \to m\ Y$, term monads M_1 and M_2 of Σ, and term monad morphism $f :: M_1\ a \to M_2\ a$ from M_1 to M_2, the following diagram commutes:*

$$\begin{tikzcd} X \arrow[r, "g_{M_1}"] \arrow[dr, "g_{M_2}"'] & M_1\ Y \arrow[d, "f_Y"] \\ & M_2\ Y \end{tikzcd} \tag{34}$$

where subscripts are type applications of polymorphic functions.

Lemma B.2. *Monad Free Σ is a term monad of Σ for any functor Σ:*

$$\textbf{instance}\ Functor\ \Sigma \Rightarrow TermMonad\ (Free\ \Sigma)\ \Sigma\ \textbf{where}\ con = Op$$

Proof. As shown in Section 2, *Free* Σ is a monad and its $\gg\!\!=$ implementation directly entails algebraicity. □

Remark B.1. In fact, *Free* Σ is the *initial* term monad of Σ: for any term monad m of Σ, there is exactly one term monad morphism from *Free* Σ to m.

Lemma B.3. *For any type c carrying a Σ-algebra $alg :: \Sigma\ c \to c$, monad $Cont_c$ is a term monad of Σ with $con = liftAlgCont\ alg$ where liftAlgCont (17) is defined in Section 5.*

Proof. $Cont_C$ is clearly a monad. What remains is to show that con satisfies algebraicity:

$$\begin{array}{cl} & con\ op \gg\!\!= k \\ = & \{\Downarrow \text{Expanding } con\} \\ & liftAlgCont\ alg\ op \gg\!\!= k \\ = & \{\Downarrow \text{Expanding } liftAlgCont\ (17)\} \\ & Cont\ (\lambda q \to alg\ (fmap\ (\lambda m \to runCont\ m\ q)\ op)) \gg\!\!= k \\ = & \{\Downarrow \text{Expanding} \gg\!\!= \text{and letting } m = Cont\ (\lambda q \to alg\ (fmap\ (\lambda n \to runCont\ n\ q)\ op))\} \\ & Cont\ (\lambda t \to runCont\ m\ (\lambda x \to runCont\ (k\ x)\ t)) \\ = & \{\Downarrow \text{Expanding } runCont\ m\} \\ & Cont\ (\lambda t \to (\lambda q \to alg\ (fmap\ (\lambda n \to runCont\ n\ q)\ op))\ (\lambda x \to runCont\ (k\ x)\ t)) \end{array}$$

$$
\begin{aligned}
&= \quad \{\downarrow\ \beta\text{-reducing}\,\} \\
&\quad \mathit{Cont}\ (\lambda t \to \mathit{alg}\ (\mathit{fmap}\ (\lambda n \to \mathit{runCont}\ n\ (\lambda x \to \mathit{runCont}\ (k\ x)\ t))\ \mathit{op})) \\
&= \quad \{\uparrow\ \text{Definition of } n \ggg k\,\} \\
&\quad \mathit{Cont}\ (\lambda t \to \mathit{alg}\ (\mathit{fmap}\ (\lambda n \to \mathit{runCont}\ (n \ggg k)\ t)\ \mathit{op})) \\
&= \quad \{\uparrow\ \text{Functorial law: } \mathit{fmap} \text{ preserves function composition}\,\} \\
&\quad \mathit{Cont}\ (\lambda t \to \mathit{alg}\ (\mathit{fmap}\ (\lambda n \to \mathit{runCont}\ n\ t)\ (\mathit{fmap}\ ({\ggg}k)\ \mathit{op}))) \\
&= \quad \{\uparrow\ \text{Expanding } \mathit{liftAlgCont}\ (17)\,\} \\
&\quad \mathit{liftAlgCont}\ \mathit{alg}\ (\mathit{fmap}\ ({\ggg}k)\ \mathit{op} \\
&= \quad \{\uparrow\ \text{Expanding } \mathit{con}\,\} \\
&\quad \mathit{con}\ (\mathit{fmap}\ ({\ggg}k)\ \mathit{op})
\end{aligned}
$$

□

Lemma B.4. *Given a modular handler $h_2 :: \mathit{MHandler}\ \Sigma_2\ D\ Y\ Z$, then for any signature sig', $\mathit{Cont}_{D\ (\mathit{Free}\ sig')}$ is a term monad of $\Sigma_2 + sig'$ with*

$$\mathit{con}_{CPS} = \mathit{liftAlgCont}\ (\mathit{openAlg}\ h_2)$$

Proof. Because $\mathit{openAlg}\ h_2$ have type $(\Sigma_2 + sig')\ (D\ (\mathit{Free}\ sig')) \to D\ (\mathit{Free}\ sig')$, by Lemma B.3, $\mathit{Cont}_{D\ (\mathit{Free}\ sig')}$ is a term monad of $\Sigma_2 + sig'$. □

Lemma B.5. *Given the data as in the last lemma, $\mathit{fold}\ \mathit{return}_{Cont}\ \mathit{con}_{CPS}$ is a term monad morphism from $\mathit{Free}\ (\Sigma_2 + sig')$ to $\mathit{Cont}_{D\ (\mathit{Free}\ sig')}$.*

Proof. By the definition of *fold*, it is clearly a $(\Sigma_2 + sig')$-homomorphism. With some calculation, it can be shown that it is also a monad morphism. □

B.2 Handle with Term Monads

To apply Theorem B.1 to fuse $\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1$, we need (i) $\mathit{handle}\ h_1$ to operate on a parametric term monad of $\Sigma_2 + Sig'$ instead of just $\mathit{Free}\ (\Sigma_2 + Sig')$, and (ii) the fold in $\mathit{handle}\ h_2$ to be a term monad morphism.

For the first requirement, we define the following generalised version of *handle*:

$$
\begin{aligned}
&\mathit{ghandle} :: (\mathit{MCarrier}\ c, \mathit{Functor}\ \Sigma, \mathit{Functor}\ sig', \mathit{TermMonad}\ m\ sig') \\
&\qquad\qquad \Rightarrow \mathit{MHandler}\ \Sigma\ c\ a\ b \to \mathit{Free}\ (\Sigma + sig')\ a \to m\ b \\
&\mathit{ghandle}\ h = \mathit{run}\ h \cdot \mathit{fold}\ (\mathit{gen}\ h)\ (\mathit{gopenAlg}\ h) \\
&\mathit{gopenAlg} :: (\mathit{Functor}\ sig', \mathit{MCarrier}\ c, \mathit{TermMonad}\ m\ sig') \\
&\qquad\qquad \Rightarrow \mathit{MHandler}\ \Sigma\ c\ a\ b \to (\Sigma + sig')\ (c\ m) \to c\ m \\
&\mathit{gopenAlg}\ h = \mathit{split}\ (\mathit{alg}\ h)\ (\mathit{fwd} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return})
\end{aligned}
$$

It is clear that

$$(\mathit{ghandle}\ h_1)_{\mathit{Free}\ sig'} = \mathit{handle}\ h_1 \tag{35}$$

Then for the second requirement, we have the following lemma.

Lemma B.6. *Given $h_2 :: \mathit{MHandler}\ \Sigma_2\ D\ Y\ Z$, define*

$$\mathit{fold}\ (\mathit{gen}\ h_2)\ (\mathit{openAlg}\ h_2) = (\lambda x \to \mathit{runCont}\ x\ (\mathit{gen}\ h_2)) \cdot \mathit{fold}\ \mathit{return}_{Cont}\ \mathit{con}_{CPS}$$

Proof. It is an ordinary fold fusion [Bird and de Moor 1997; Hinze 2013], and it can be verified that $(\lambda x \to \mathit{runCont}\ x\ (\mathit{gen}\ h_2))$ is an algebra homomorphism from $(\mathit{return}, \mathit{con}_{CPS})$ to $(\mathit{gen}\ h_2, \mathit{openAlg}\ h_2)$. □

Now we can use Theorem B.1 to fuse the folds of the two handlers:

$$
\begin{aligned}
& \mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1 \\
= \;& \{\Downarrow \text{ Equation 33 } \} \\
& \mathit{run}\ h_2 \cdot \mathit{fold}\ (\mathit{gen}\ h_2)\ (\mathit{openAlg}\ h_2) \cdot \mathit{handle}\ h_1 \\
= \;& \{\Downarrow \text{ Equation 35 } \} \\
& \mathit{run}\ h_2 \cdot \mathit{fold}\ (\mathit{gen}\ h_2)\ (\mathit{openAlg}\ h_2) \cdot (\mathit{ghandle}\ h_1)_{\mathit{Free}\ (\Sigma_2 + \mathit{sig}')} \\
= \;& \{\Downarrow \text{ Lemma B.6 } \} \\
& \mathit{run}\ h_2 \cdot (\lambda x \rightarrow \mathit{runCont}\ x\ (\mathit{gen}\ h_2)) \cdot \mathit{fold}\ \mathit{return}_{\mathit{Cont}}\ \mathit{con}_{\mathit{CPS}} \cdot (\mathit{ghandle}\ h_1)_{\mathit{Free}\ (\Sigma_2 + \mathit{sig}')} \\
= \;& \{\Downarrow \text{ Theorem B.1 and Lemma B.5 } \} \\
& \mathit{run}\ h_2 \cdot (\lambda x \rightarrow \mathit{runCont}\ x\ (\mathit{gen}\ h_2)) \cdot (\mathit{ghandle}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')}
\end{aligned}
$$

Note that in the last line we only have one fold on the syntax tree now.

Now we calculate from the side of $\mathit{handle}\ (h_2 \diamond h_1)$:

$$
\begin{aligned}
& \mathit{handle}\ (h_2 \diamond h_1) \\
={}& \mathit{run}\ (h_2 \diamond h_1) \cdot \mathit{fold}\ (\mathit{gen}\ (h_2 \diamond h_1))\ (\mathit{openAlg}\ (h_2 \diamond h_1)) \\
={}& \mathit{run}\ h_2 \cdot (\lambda x \rightarrow \mathit{runCont}\ x\ (\mathit{gen}\ h_2)) \cdot \mathit{run}\ h_1 \cdot \mathit{fold}\ (\mathit{gen}\ (h_2 \diamond h_1))\ (\mathit{openAlg}\ (h_2 \diamond h_1)) \\
={}& \mathit{run}\ h_2 \cdot (\lambda x \rightarrow \mathit{runCont}\ x\ (\mathit{gen}\ h_2)) \cdot \mathit{run}\ h_1 \cdot \mathit{fold}\ (\mathit{gen}\ h_1)\ (\mathit{openAlg}\ (h_2 \diamond h_1))
\end{aligned}
$$

(we omit constructors and destructors *Fused* and *unFused* for clarity). To complete the proof of $\mathit{handle}\ h_2 \cdot \mathit{handle}\ h_1 = \mathit{handle}\ (h_2 \diamond h_1) \cdot \mathit{assoc}^{\circ}$, it is sufficient to show

$$(\mathit{ghandle}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')} = \mathit{run}\ h_1 \cdot \mathit{fold}\ (\mathit{gen}\ h_1)\ (\mathit{openAlg}\ (h_2 \diamond h_1)) \cdot \mathit{assoc}^{\circ} \tag{36}$$

By definition of *ghandle*,

$$(\mathit{ghandle}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')} = \mathit{run}\ h \cdot \mathit{fold}\ (\mathit{gen}\ h)\ (\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')}$$

Thus it is sufficient to show

$$\mathit{fold}\ (\mathit{gen}\ h)\ (\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')} = \mathit{fold}\ (\mathit{gen}\ h_1)\ (\mathit{openAlg}\ (h_2 \diamond h_1)) \cdot \mathit{assoc}^{\circ} \tag{37}$$

Lemma B.7. *Let* $\phi :: (\Sigma_1 + (\Sigma_2 + \mathit{sig}'))\ a \rightarrow ((\Sigma_1 + \Sigma_2) + \mathit{sig}')\ a$ *be the evident isomorphism between these two signatures. We have*

$$(\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')} = (\mathit{openAlg}\ (h_2 \diamond h_1)) \cdot \phi \tag{38}$$

Proof. By case analysis on input x,

Case B.7.1. If $x = \mathit{Inl}\ c$,

$$(\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')}\ (\mathit{Inl}\ c) = \mathit{alg}\ h_1\ c = \mathit{openAlg}\ (h_2 \diamond h_1)\ (\phi\ (\mathit{Inl}\ c))$$

Case B.7.2. If $x = \mathit{Inr}\ (\mathit{Inl}\ c)$,

$$
\begin{aligned}
& (\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')}\ (\mathit{Inr}\ (\mathit{Inl}\ c)) \\
= \;& \{\Downarrow \text{ definition of } \mathit{gopenAlg} \} \\
& (\mathit{fwd} \cdot \mathit{con}_{\mathit{CPS}} \cdot \mathit{fmap}\ \mathit{return})\ (\mathit{Inl}\ c) \\
= \;& \{\Downarrow \text{ definition of } \mathit{con}_{\mathit{CPS}} \} \\
& (\mathit{fwd} \cdot \mathit{liftAlgCont}\ (\mathit{openAlg}\ h_2) \cdot \mathit{fmap}\ \mathit{return})\ (\mathit{Inl}\ c) \\
= \;& \{\Downarrow\ \mathit{fmap} \text{ on } \mathit{Inl} \} \\
& (\mathit{fwd} \cdot \mathit{liftAlgCont}\ (\mathit{openAlg}\ h_2))\ (\mathit{Inl}\ (\mathit{fmap}\ \mathit{return}\ c)) \\
= \;& \{\Downarrow \text{ definition of } \mathit{liftAlgCont} \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{fwd}\ (\mathit{cont}\ (\lambda k \to \mathit{openAlg}\ h_2\ (\mathit{fmap}\ (\lambda m \to \mathit{runCont}\ m\ k)\ (\mathit{Inl}\ (\mathit{fmap}\ \mathit{return}\ c))))) \\
= \quad & \{\downarrow\ \mathit{fmap} \text{ on } \mathit{Inl}\ \} \\
& \mathit{fwd}\ (\mathit{cont}\ (\lambda k \to \mathit{openAlg}\ h_2\ (\mathit{Inl}\ (\mathit{fmap}\ ((\lambda m \to \mathit{runCont}\ m\ k) \cdot \mathit{return})\ c)))) \\
= \quad & \{\downarrow \text{ definition } \mathit{openAlg}\ h_2 \text{ on } \mathit{Inl}\ \} \\
& \mathit{fwd}\ (\mathit{cont}\ (\lambda k \to \mathit{alg}\ h_2\ (\mathit{fmap}\ ((\lambda m \to \mathit{runCont}\ m\ k) \cdot \mathit{return})\ c))) \\
= \quad & \{\downarrow \text{ cancelling } \mathit{return} \text{ and } \mathit{runCont}\ \} \\
& \mathit{fwd}\ (\mathit{cont}\ (\lambda k \to \mathit{alg}\ h_2\ (\mathit{fmap}\ ((\lambda m \to \mathit{runCont}\ m\ k) \cdot \mathit{return})\ c))) \\
= \quad & \{\uparrow \text{ definition of } \mathit{liftAlgCont}\ \} \\
& (\mathit{fwd} \cdot \mathit{liftAlgCont}\ (\mathit{alg}\ h_2) \cdot \mathit{fmap}\ \mathit{return})\ c \\
= \quad & \{\uparrow \text{ definition of } \diamond\ \} \\
& \mathit{alg}\ (h_2 \diamond h_1)\ (\mathit{Inr}\ c) \\
= \quad & \{\uparrow \text{ definition of } \mathit{openAlg}\ \} \\
& \mathit{openAlg}\ (h_2 \diamond h_1)\ (\mathit{Inl}\ (\mathit{Inr}\ c)) \\
= \quad & \{\uparrow \text{ definition of } \phi\ \} \\
& \mathit{openAlg}\ (h_2 \diamond h_1)\ (\phi\ (\mathit{Inr}\ (\mathit{Inl}\ c)))
\end{aligned}
$$

Case B.7.3. If $x = \mathit{Inr}\ (\mathit{Inr}\ c)$,

$$
\begin{aligned}
& (\mathit{gopenAlg}\ h_1)_{\mathit{Cont}_D\ (\mathit{Free}\ \mathit{sig}')}\ (\mathit{Inr}\ (\mathit{Inr}\ c)) \\
= \quad & \{\downarrow \text{ definition of } \mathit{gopenAlg}\ \} \\
& (\mathit{fwd} \cdot \mathit{con}_{\mathit{CPS}} \cdot \mathit{fmap}\ \mathit{return})\ (\mathit{Inr}\ c) \\
= \quad & \{\downarrow \text{ definition of } \mathit{con}_{\mathit{CPS}}\ \} \\
& (\mathit{fwd} \cdot (\mathit{liftAlgCont}\ (\mathit{openAlg}\ h_2)) \cdot \mathit{fmap}\ \mathit{return})\ (\mathit{Inr}\ c) \\
= \quad & \{\downarrow \text{ definition of } \mathit{liftAlgCont}\ \} \\
& (\mathit{fwd} \cdot (\mathit{liftAlgCont}\ (\mathit{fwd}_D \cdot \mathit{Op} \cdot \mathit{fmap}\ \mathit{Var})) \cdot \mathit{fmap}\ \mathit{return})\ c \\
= \quad & \{\downarrow \text{ definition of } \mathit{liftAlgCont} \text{ and simplification}\ \} \\
& \mathit{fwd}\ (\mathit{cont}\ (\lambda k \to (\mathit{fwd}_D \cdot \mathit{Op})\ (\mathit{fmap}\ (\lambda x \to \mathit{Var}\ (k\ x))\ c))) \\
= \quad & \{\uparrow\ (\mathit{fmap}\ k \cdot \mathit{Var}) = (\lambda x \to \mathit{Var}\ (k\ x))\ \} \\
& \mathit{fwd}_C\ (\mathit{cont}\ (\lambda k \to \mathit{fwd}_D\ (\mathit{Op}\ (\mathit{fmap}\ (\mathit{fmap}\ k \cdot \mathit{Var})\ c)))) \\
= \quad & \{\uparrow\ \mathit{fmap} \text{ preseves function composition}\ \} \\
& \mathit{fwd}_C\ (\mathit{cont}\ (\lambda k \to \mathit{fwd}_D\ (\mathit{Op}\ (\mathit{fmap}\ (\mathit{fmap}\ k)\ (\mathit{fmap}\ \mathit{Var}\ c))))) \\
= \quad & \{\uparrow\ \mathit{fmap} \text{ on } \mathit{Op}\ \} \\
& \mathit{fwd}_C\ (\mathit{cont}\ (\lambda k \to \mathit{fwd}_D\ (\mathit{fmap}\ k\ (\mathit{Op}\ (\mathit{fmap}\ \mathit{Var}\ c))))) \\
= \quad & \{\uparrow \text{ definition of } \mathit{cps}_{\mathit{EM}}\ \} \\
& (\mathit{fwd}_C \cdot \mathit{cps}_{\mathit{EM}} \cdot \mathit{Op} \cdot \mathit{fmap}\ \mathit{Var})\ c \\
= \quad & \{\uparrow \text{ definition of } \mathit{fwd} \text{ for } \mathit{Fused}\ \} \\
& (\mathit{fwd}_{\mathit{Fused}} \cdot \mathit{Op} \cdot \mathit{fmap}\ \mathit{Var})\ c \\
= \quad & \{\uparrow \text{ definition of } h_2 \diamond h_1\ \} \\
& \mathit{openAlg}\ (h_2 \diamond h_1)\ (\mathit{Inr}\ c) \\
= \quad & \{\uparrow \text{ definition of } \phi\ \} \\
& \mathit{openAlg}\ (h_2 \diamond h_1)\ (\phi\ (\mathit{Inr}\ (\mathit{Inr}\ c)))
\end{aligned}
$$

□

Now Equation 37 follows from this lemma by *base functor fusion* [Hinze 2013]. This complete our proof of Theorem 5.3.

C CORRECT HANDLERS INDUCE CORRECT TRANSFORMATIONS

This section proves Theorem 4.1. For brevity, we will only prove for open correctness of Theorem 4.1 since the case for closed correctness can be proved by replacing all occurrences of 'for any $T' :: \mathit{Theory}\ \mathit{sig}'$' in the proof with T' being the empty theory. We will use functions defined in Definition B.1, the function *ghandle*, and the concept of term monads (Definition B.2) from the previous section, but the other results from the last section are not used.

Given a theory $T :: \mathit{Theory}\ \Sigma$, a model of T is a set C with an algebra $\Sigma\ C \to C$ that respects the equations of T. It is standard [Bauer 2018; Plotkin and Power 2002] that given any set X, the quotient set $(\mathit{Free}\ \Sigma\ X)/{\sim_T}$ is the *free model* of T generated by X, and that the mapping from X to $(\mathit{Free}\ \Sigma\ X)/{\sim_T}$ is a monad, which we denote by monad $FM\ \Sigma$ with

$$\begin{aligned}
&\mathit{return}\ x = [\,\mathit{Var}\ x\,]\\
&[\,\mathit{return}\ x\,] \gg\!= k = k\ x\\
&[\,\mathit{Op}\ \mathit{op}\,] \quad \gg\!= k = [\,\mathit{Op}\ (\mathit{fmap}\ ({\gg\!=} k)\ \mathit{op})\,]
\end{aligned}$$

where $[-]$ means the equivalence class that an element belongs to, instead of a list. Because $\sim_T$ is defined to be a congruence relation on $\mathit{Free}\ \Sigma$ (see Definition 3.1), the above definition of $\gg\!=$ is well defined. FM is clearly a term monad (Definition B.2) of Σ with $\mathit{con}\ x = [\,\mathit{Op}\ x\,]$. The universal property of FM says that given any model $(C :: *, \mathit{alg} :: \Sigma\ C \to C)$ and a function $\mathit{gen} :: X \to C$, there is a unique T-model homomorphism $\overline{\mathit{fold}}\ \mathit{gen}\ \mathit{alg}$ from $FM\ X$ to C such that

$$\overline{\mathit{fold}}\ \mathit{gen}\ \mathit{alg}\ [\,\mathit{Var}\ x\,] = \mathit{gen}\ x$$

Additionally, for any $m :: \mathit{Free}\ \Sigma\ X$,

$$\overline{\mathit{fold}}\ \mathit{gen}\ \mathit{alg}\ [\,m\,] = \mathit{fold}\ \mathit{gen}\ \mathit{alg}\ m \tag{39}$$

Lemma C.1. *Given a term monad M of Σ, a modular carrier C, define*

$$\begin{aligned}
&\mathit{forward} :: (\mathit{TermMonad}\ M\ \Sigma, \mathit{MCarrier}\ C) \Rightarrow \Sigma\ (C\ M) \to C\ M\\
&\mathit{forward} = \mathit{fwd} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return}
\end{aligned}$$

then $\mathit{fwd} :: M\ (C\ M) \to C\ M$ is a Σ-algebra homomorphism from $\mathit{con} :: \Sigma\ (M\ (C\ M)) \to M\ (C\ M)$ to $\mathit{forward} :: \Sigma\ (C\ M) \to C\ M$:

$$\mathit{fwd} \cdot \mathit{con} = \mathit{forward} \cdot \mathit{fmap}\ \mathit{fwd}$$

Proof. First we define $\mathit{con}' :: \forall a.\ \Sigma\ a \to M\ a$ by $\mathit{con}' = \mathit{con} \cdot \mathit{fmap}\ \mathit{return}$. Conversely,

$$\begin{aligned}
&\mathit{con}\\
=\ &\quad \{\uparrow\ \mathit{join} \cdot \mathit{return} = \mathit{id}\,\}\\
&\mathit{con} \cdot \mathit{fmap}\ \mathit{join} \cdot \mathit{fmap}\ \mathit{return}\\
=\ &\quad \{\uparrow\ \text{algebraicity of } \mathit{con}\,\}\\
&\mathit{join} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return}\\
=\ &\mathit{join} \cdot \mathit{con}'
\end{aligned}$$

Then we calculate

$$\mathit{forward} \cdot \mathit{fmap}\ \mathit{fwd}$$

$$
\begin{aligned}
&= \mathit{fwd} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return} \cdot \mathit{fmap}\ \mathit{fwd} \\
&= \mathit{fwd} \cdot \mathit{con}' \cdot \mathit{fmap}\ \mathit{fwd} \\
&= \quad \{\downharpoonleft \text{ naturality of } \mathit{con}' \} \\
&\quad \mathit{fwd} \cdot \mathit{fmap}\ \mathit{fwd} \cdot \mathit{con}' \\
&= \quad \{\downharpoonleft \text{ Eilenberg-Moore property of } \mathit{fwd} \text{ (Equation 14)} \} \\
&\quad \mathit{fwd} \cdot \mathit{join} \cdot \mathit{con}' \\
&= \mathit{fwd} \cdot \mathit{con}
\end{aligned}
$$

□

Lemma C.2. *Given a modular carrier C, a term monad M of Σ, if $con_{M,a}$ respects an equation $lhs = rhs$ for any a in the sense of Definition A.1, then the forward function defined in the last lemma respects $lhs = rhs$ too.*

Proof. For any function f of type

$$\forall c.\ (\Sigma\ c \to c) \to G \to (V \to c) \to c$$

for some G and V, if we view c and the first argument $\Sigma\ c \to c$ as a Σ-algebra, by parametricity [Voigtländer 2009; Reynolds 1983; Wadler 1989] and Lemma C.1, given any $g :: G$ and $k :: V \to M\ (C\ M)$, we have

$$\mathit{fwd}\ (f\ \ \mathit{con}\ g\ k) = f\ \mathit{forward}\ g\ (\mathit{fwd} \cdot k) \tag{40}$$

Assuming lhs and rhs has type $\mathit{Template}\ G\ V$, to prove $lhs\ \mathit{forward}\ g\ k' = rhs\ \mathit{forward}\ g\ k'$ for any $g :: G$ and $k' :: V \to C\ M$, we define $k = \mathit{return} \cdot k'$. By the Eilenberg-Moore property of fwd, we have $k' = \mathit{fwd} \cdot k$ and we calculate

$$
\begin{aligned}
&\quad lhs\ \mathit{forward}\ g\ k' \\
&= lhs\ \mathit{forward}\ g\ (\mathit{fwd} \cdot k) \\
&= \quad \{ \text{ Equation 40 for } f := lhs \} \\
&\quad \mathit{fwd}\ (lhs\ \mathit{con}\ g\ k) \\
&= \quad \{ \text{ assumption that } \mathit{con} \text{ respects } lhs = rhs \} \\
&\quad \mathit{fwd}\ (rhs\ \mathit{con}\ g\ k) \\
&= \quad \{ \text{ reverse of the previous steps} \} \\
&\quad rhs\ \mathit{forward}\ g\ k'
\end{aligned}
$$

Then we conclude $\mathit{forward}$ respects the equation $lhs = rhs$. □

Lemma C.3. *Given $h :: \mathit{MHandler}\ \Sigma\ C\ A\ B$, if h is a correct open handler of theory $T :: \mathit{Theory}\ \Sigma$, then for any sig' and $T' :: \mathit{Theory}\ sig'$, $C\ (\mathit{FM}\ sig')$ with*

$$\mathit{gopenAlg}\ h :: (\Sigma + sig')\ (C\ (\mathit{FM}\ sig')) \to C\ (\mathit{FM}\ sig')$$

defined as in Section B.2 is a model of $T + T'$.

Proof. Because h is a correct open handler of T, $C\ (\mathit{FM}\ sig')$ is a model of T with algebra $\mathit{alg}\ h$. The monad $\mathit{FM}\ sig'$ is a term monad of sig', so Lemma C.2 implies that $C\ (\mathit{FM}\ sig')$ with algebra $\mathit{fwd} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return}$ is model of T'. By definition, equations of $T + T'$ are either an equation from T or T', and $\mathit{gopenAlg}\ h$ is exactly $\mathit{split}\ (\mathit{alg}\ h)\ (\mathit{fwd} \cdot \mathit{con} \cdot \mathit{fmap}\ \mathit{return})$. Thus $C\ (\mathit{FM}\ sig')$ is a model of $T + T'$. □

Lemma C.4. *Given a handler $h :: MHandler\ \Sigma\ C\ A\ B$, two theories $T :: \mathit{Theory}\ \Sigma$ and $T :: \mathit{Theory}\ sig'$, if $c_1, c_2 :: Free\ (\Sigma + sig')\ A$ for some type A such that $c_1 \sim_{T+T'} c_2$, then*

$$fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ c_1 = fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ c_2$$

Proof. By Lemma C.3, $C\ (FM\ sig')$ is a $(T + T')$-model with algebra $(gopenAlg\ h)_{FM\ sig'}$. Thus by Equation 39, for any $c :: Free\ (\Sigma + sig')\ A$

$$fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ c = \overline{fold}\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ [\, c\,]$$

Now that $c_1 \sim_{T+T'} c_2$, $[c_1] = [c_2]$. Therefore,

$$\begin{aligned} & fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ c_1 \\ = & \overline{fold}\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ [\, c_2\,] \\ = & fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'}\ c_2 \end{aligned}$$

□

Now we are ready to prove Theorem 4.1 using parametricity [Voigtländer 2009; Reynolds 1983; Wadler 1989]. Let $h :: MHandler\ \Sigma\ C\ A\ B$ be a correct open handler of $T :: \mathit{Theory}\ \Sigma$, $T' :: \mathit{Theory}\ sig'$ be any theory, $c_1, c_2 :: Free\ (\Sigma + sig')\ A$ be any two computations such that $c_1 \sim_{T+T'} c_2$. Because *ghandle* is polymorphic in its *TermMonad* argument m, and $[\,\_\,] :: \forall a.\ Free\ sig'\ a \to FM\ sig'\ a$ is evidently a *TermMonad* morphism, thus by parametricity, for any c,

$$[\, (ghandle\ h_1)_{Free\ sig'}\ c\,] = (ghandle\ h_1)_{FM\ sig'}\ c \tag{41}$$

Then

$$\begin{aligned} & [\, handle\ c_1\,] \\ = & [\, (ghandle\ h_1)_{Free\ sig'}\ c_1\,] \\ = & (ghandle\ h_1)_{FM\ sig'}\ c_1 \\ = & ((run\ h)_{FM\ sig'} \cdot fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'})\ c_1 \\ = & \quad \{\ \text{Lemma C.4}\ \} \\ & ((run\ h)_{FM\ sig'} \cdot fold\ (gen\ h)_{FM\ sig'}\ (gopenAlg\ h)_{FM\ sig'})\ c_2 \\ = & \quad \{\ \text{reverse of the above steps}\ \} \\ & [\, handle\ c_2\,] \end{aligned}$$

Then by definition $[\, handle\ c_1\,] = [\, handle\ c_2\,]$ iff. $handle\ c_1 \sim_{T'} handle\ c_2$, which is what we want to show.

D PROOFS OF PRESERVATION OF EQUATIONS

This section contains detailed calculations to prove Theorem 5.5.

Lemma D.1. *For any functor Σ, types Γ, V, function $f :: \forall c.\ (\Sigma\ c \to c) \to \Gamma \to (V \to c) \to c$, function $alg :: \Sigma\ R \to R$ for some type R, it holds that*

$$f\ (liftAlgCont\ alg)\ g\ k = Cont\ (\lambda q \to f\ alg\ g\ ((\lambda m \to runCont\ m\ q) \cdot k))$$

for any g and k.

Proof. It is sufficient to show that for any type a and any $q :: a \to R$,

$$runCont\ (f\ (liftAlgCont\ alg)\ g\ k)\ q = f\ alg\ g\ ((\lambda m \to runCont\ m\ q) \cdot k)$$

This is a consequence of the parametricity of f because $\lambda m \to runCont\ \ m\ \ q$ is a Σ-algebra homomorphism from

$$\begin{aligned}
&liftAlgCont\ alg :: \Sigma\ (Cont_R\ a) \to Cont_R\ a\\
&liftAlgCont\ alg\ s = cont\ (\lambda k \to alg\ (fmap\ (\lambda m \to runCont\ m\ k)\ s))
\end{aligned}$$

to $alg :: \Sigma\ R \to R$. □

Theorem D.2 (Preservation of Equations). *Suppose h_1 and h_2 are modular handlers of signatures Σ_1 and Σ_2 respectively. If h_1 and h_2 are correct open (resp. closed) handlers of $T_1 :: Theory\ \Sigma_1$ and $T_2 :: Theory\ \Sigma_2$ correspondingly, then $h_2 \diamond h_1$ is a correct open (resp. closed) handler of $T_1 + T_2$.*

Proof. By Definition 2.3, each equation $Eqn_C\ lhs\ rhs$ of $T_1 + T_2$ is either an equation from T_1 or an equation from T_2 lifted to signature $\Sigma_1 + \Sigma_2$. If it is from $Eqn_C\ lhs'\ rhs' :: Equation_C\ \Sigma_1\ \Gamma\ V$ from T_1, then

$$\begin{aligned}
&lhs, rhs :: \forall c.\ ((\Sigma_1 + \Sigma_2)\ c \to c) \to \Gamma \to (V \to c) \to c\\
&lhs\ alg = lhs'\ (alg \cdot Inl)\\
&rhs\ alg = rhs'\ (alg \cdot Inl)
\end{aligned}$$

Then

$$\begin{aligned}
&\quad lhs\ (alg\ (h_2 \diamond h_1)) = rhs\ (alg\ (h_2 \diamond h_1))\\
&\Leftrightarrow lhs'\ (alg\ (h_2 \diamond h_1) \cdot Inl) = rhs'\ (alg\ (h_2 \diamond h_1) \cdot Inl)\\
&\Leftrightarrow \quad \{\ \text{definition of } alg\ (h_2 \diamond h_1)\ \}\\
&\quad lhs'\ (alg\ h_1) = lhs'\ (alg\ h_1)
\end{aligned}$$

The last line holds because h_1 is a correct handler of $lhs' = rhs'$ by assumption.

If $Eqn_C\ lhs\ rhs$ is from T_2, then

$$\begin{aligned}
&lhs, rhs :: \forall c.\ ((\Sigma_1 + \Sigma_2)\ c \to c) \to \Gamma \to (V \to c) \to c\\
&lhs\ alg = lhs'\ (alg \cdot Inr)\\
&rhs\ alg = rhs'\ (alg \cdot Inr)
\end{aligned}$$

By definition,

$$alg\ (h_2 \diamond h_1)\ (Inr\ op) = (Fused \cdot fwd \cdot liftAlgCont\ (alg\ h_2) \cdot fmap\ (return \cdot unFused))\ op$$

By Lemma C.2, to show that $alg\ (h_2 \diamond h_1)$ respects $lhs = rhs$, it is sufficient to show that

$$liftAlgCont\ (alg\ h_2) :: Monad\ m \Rightarrow \Sigma_2\ (Cont_{D\ m}\ a) \to Cont_{D\ m}\ a$$

respects $lhs' = rhs'$ for any m and a. Then for any g and k,

$$\begin{aligned}
&\quad lhs'\ (liftAlgCont\ (alg\ h_2))\ g\ k\\
&= \quad \{\Downarrow\ \text{Lemma D.1}\ \}\\
&\quad Cont_{\lambda q \to lhs'\ (alg\ h_2)\ g\ ((\lambda m \to runCont\ m\ q)\cdot k)}\\
&= \quad \{\Downarrow\ \text{assumption that } alg\ h_2 \text{ respects } lhs' = rhs'\ \}\\
&\quad Cont_{\lambda q \to rhs'\ (alg\ h_2)\ g\ ((\lambda m \to runCont\ m\ q)\cdot k)}\\
&= lhs'\ (liftAlgCont\ (alg\ h_2))\ g\ k\\
&\quad \{\Downarrow\ \text{Lemma D.1}\ \}
\end{aligned}$$

□

E MISCELLANEOUS CALCULATIONS

This section contains the calculations omitted in the main text. Most of them are straightforward equational proofs.

E.1 Eilenberg-Moore Laws for the Fused Modular Carrier

In Theorem 5.2 we define the fused modular carrier to be

$$\textbf{newtype}\ \mathit{Fused}\ c\ d\ m = \mathit{Fused}\ \{\mathit{unFused} :: c\ (\mathit{Cont}_{d\ m})\}$$

with the following *fwd* function:

$$\begin{array}{l}
\textbf{instance}\ (\mathit{MCarrier}\ c, \mathit{MCarrier}\ d) \Rightarrow \mathit{MCarrier}\ (\mathit{Fused}\ c\ d)\ \textbf{where} \\
\quad \mathit{fwd} = \mathit{Fused} \cdot \mathit{fwd} \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{cps}_{EM} \\
\mathit{cps}_{EM} :: (\mathit{MCarrier}\ d, \mathit{Monad}\ m) \Rightarrow m\ a \to \mathit{Cont}_{d\ m}\ a \\
\mathit{cps}_{EM}\ m = \mathit{Cont}\ (\lambda k \to \mathit{fwd}\ (\mathit{fmap}\ k\ m))
\end{array}$$

Here we prove that this *fwd* instance indeed satisfies the Eilenberg-Moore laws (14). For the first equation $\mathit{fwd} \cdot \mathit{return} = \mathit{id}$ in (14):

$$\begin{array}{l}
\quad \mathit{fwd}\ (\mathit{return}\ x) \\
= (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused})\ (\mathit{cps}_{EM}\ (\mathit{return}\ x)) \\
= (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused})\ (\mathit{Cont}\ (\lambda k \to \mathit{fwd}_d\ (\mathit{fmap}\ k\ (\mathit{return}\ x)))) \\
= (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused})\ (\mathit{Cont}\ (\lambda k \to \mathit{fwd}_d\ (\mathit{return}\ (k\ x)))) \\
\qquad \{\downarrow\ \mathit{fwd}_d \cdot \mathit{return} = \mathit{id}\} \\
= (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused})\ (\mathit{Cont}\ (\lambda k \to k\ x)) \\
= (\mathit{Fused} \cdot \mathit{fwd}_c)\ (\mathit{Cont}\ (\lambda k \to k\ (\mathit{unFused}\ x))) \\
= (\mathit{Fused} \cdot \mathit{fwd}_c)\ (\mathit{return}_{\mathit{Cont}}\ (\mathit{unFused}\ x)) \\
\qquad \{\downarrow\ \mathit{fwd}_c \cdot \mathit{return} = \mathit{id}\} \\
= \mathit{Fused}\ (\mathit{unFused}\ x) \\
= x
\end{array}$$

For the second equation $\mathit{fwd} \cdot \mathit{fmap}\ \mathit{fwd} = \mathit{fwd} \cdot \mathit{join}$,

$$\begin{array}{l}
\quad \mathit{fwd} \cdot \mathit{fmap}\ \mathit{fwd} \\
= \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{cps}_{EM} \cdot \mathit{fmap}\ (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{cps}_{EM}) \\
= \quad \{\downarrow\ \text{Naturality}\} \\
\quad \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{fmap}\ (\mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused}) \cdot \mathit{cps}_{EM} \cdot \mathit{fmap}\ \mathit{cps}_{EM} \\
= \quad \{\downarrow\ \mathit{unFused} \cdot \mathit{Fused} = \mathit{id}\} \\
\quad \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ (\mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused}) \cdot \mathit{cps}_{EM} \cdot \mathit{fmap}\ \mathit{cps}_{EM} \\
= \quad \{\uparrow\ \mathit{fwd}_c \cdot \mathit{join} = \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{fwd}_c\} \\
\quad \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{join} \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{cps}_{EM} \cdot \mathit{fmap}\ \mathit{cps}_{EM} \\
= \quad \{\uparrow\ \text{Naturality}\} \\
\quad \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{join} \cdot \mathit{cps}_{EM} \cdot \mathit{fmap}\ \mathit{cps}_{EM} \\
= \quad \{\uparrow\ \mathit{cps}_{EM}\ \text{is a monad morphism preserving}\ \mathit{join}\} \\
\quad \mathit{Fused} \cdot \mathit{fwd}_c \cdot \mathit{fmap}\ \mathit{unFused} \cdot \mathit{cps}_{EM} \cdot \mathit{join} \\
= \mathit{fwd} \cdot \mathit{join}
\end{array}$$

E.2 Calculations for Clauses of Fused Handler

This subsection provides the detailed calculations for Lemma 5.6 and Lemma 5.7.

Proof of Lemma 5.6. The clause for O_1 of $h_2 \diamond h_1$ is easy to calculate:

$$\begin{array}{l}
\quad \overline{c_1}\ p\ k \\
= \mathit{alg}\ (h_2 \diamond h_1)\ (\mathit{Inl}\ (O_1\ p\ k))
\end{array}$$

$$
\begin{aligned}
&\qquad \{\Downarrow \text{ Definition of } h_2 \diamond h_1 \text{ (Theorem 5.3)}\,\} \\
&= \mathit{Fused}\ (\mathit{alg}\ h_1\ (\mathit{fmap}\ \mathit{unFused}\ (O_1\ p\ k))) \\
&\qquad \{\Downarrow\ \mathit{fmap} \text{ on signature functors}\,\} \\
&= \mathit{Fused}\ (\mathit{alg}\ h_1\ (O_1\ p\ (\mathit{unFused} \cdot k))) \\
&\qquad \{\Uparrow \text{ Definition of } c_1 \text{ in Lemma 5.6}\,\} \\
&= \mathit{Fused}\ (c_1\ p\ (\mathit{unFused} \cdot k))
\end{aligned}
$$

The clause for O_2 needs some calculation to expand *liftAlgCont*:

$$
\begin{aligned}
&\overline{c_2}\ p\ k \\
= {} & \mathit{alg}\ (h_2 \diamond h_1)\ (\mathit{Inr}\ (O_2\ p\ k)) \\
= {} & \quad \{\Downarrow \text{ Definition of } h_2 \diamond h_1 \text{ (Theorem 5.3)}\,\} \\
& \mathit{liftAlgF}\ (\mathit{alg}\ h_2)\ (O_2\ p\ k) \\
= {} & \quad \{\Downarrow \text{ Definition of } \mathit{liftAlgF} \text{ (5.2)}\,\} \\
& (\mathit{Fused} \cdot \mathit{fwd} \cdot \mathit{liftAlgCont}\ (\mathit{alg}\ h_2) \cdot \mathit{fmap}\ (\mathit{return} \cdot \mathit{unFused}))\ (O_2\ p\ k) \\
= {} & (\mathit{Fused} \cdot \mathit{fwd} \cdot \mathit{liftAlgCont}\ (\mathit{alg}\ h_2))\ (O_2\ p\ (\mathit{return} \cdot \mathit{unFused} \cdot k)) \\
= {} & \quad \{\Downarrow \text{ Definition of } \mathit{liftAlgCont} \text{ (17)}\,\} \\
& \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to \mathit{alg}\ h_2\ (\mathit{fmap}\ (\lambda m \to \mathit{runCont}\ m\ t)\ (O_2\ p\ (\mathit{return} \cdot \mathit{unFused} \cdot k)))))) \\
= {} & \quad \{\Downarrow\ \mathit{fmap} \text{ on signature functors}\,\} \\
& \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to \mathit{alg}\ h_2\ (O_2\ p\ ((\lambda m \to \mathit{runCont}\ m\ t) \cdot \mathit{return} \cdot \mathit{unFused} \cdot k))))) \\
= {} & \quad \{\Downarrow \text{ Expanding function composition}\,\} \\
& \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to \mathit{alg}\ h_2\ (O_2\ p\ (\lambda a_2 \to \mathit{runCont}\ (\mathit{return}\ (\mathit{unFused}\ (k\ a_2)))\ t))))) \\
= {} & \quad \{\Downarrow \text{ Expanding } \mathit{return} \text{ and } \mathit{runCont} \text{ (16)}\,\} \\
& \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2))))))
\end{aligned}
$$

Proof of Lemma 5.7. When the modular carrier c of the first handler is *FreeEM* Y for some type Y, we can simplify $\overline{c_2}$:

$$
\begin{aligned}
&\overline{c_2}\ p\ k \\
= {} & \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2)))))) \\
= {} & \quad \{\Downarrow \text{ Definition of } \mathit{fwd} \text{ for } \mathit{FreeEM} \text{ (Example 3.5)}\,\} \\
& \mathit{Fused}\ (\mathit{FreeEM} \cdot \mathit{join} \cdot \mathit{fmap}\ \mathit{unFreeEM}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2)))))) \\
= {} & \quad \{\Downarrow\ \mathit{fmap} \text{ of } \mathit{Cont}.\,\} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{join}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFreeEM}\ (\mathit{unFused}\ (k\ a_2)))))))) \\
= {} & \quad \{\Downarrow\ \mathit{join} \text{ of } \mathit{Cont}.\,\} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to \mathit{runCont}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to \\
& \quad t\ (\mathit{unFreeEM}\ (\mathit{unFused}\ (k\ a_2))))))\ (\lambda x \to \mathit{runCont}\ x\ q)))) \\
= {} & \quad \{\Downarrow \text{ Eliminating } \mathit{runCont}\,\} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to c_2\ p\ (\lambda a_2 \to (\lambda x \to \mathit{runCont}\ x\ q) \\
& \quad (\mathit{unFreeEM}\ (\mathit{unFused}\ (k\ a_2))))))) \\
= {} & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to c_2\ p\ (\lambda a_2 \to \mathit{runCont}\ (\mathit{unFreeEM}\ (\mathit{unFused}\ (k\ a_2)))\ q)))) \\
= {} & \quad \{\Downarrow \text{ Letting } k' = \mathit{runCont} \cdot \mathit{unFreeEM} \cdot \mathit{unFused} \cdot k\,\} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to c_2\ p\ (\lambda a_2 \to k'\ a_2\ q))))
\end{aligned}
$$

Similarly, when the modular carrier is *StateC* S Y for some S and Y, we can simplify $\overline{c_2}$ by

$$
\begin{aligned}
&\overline{c_2}\ p\ k \\
= {} & \mathit{Fused}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2))))))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\downarrow \text{ Definition of } \mathit{fwd} \text{ for } \mathit{StateC} \text{ (Example 3.6)} \}\\
&\quad \textbf{let } mc = \mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to t\ (\mathit{unFused}\ (k\ a_2))))\\
&\quad \textbf{in } \mathit{Fused}\ (\mathit{StateC}\ (\lambda s \to (\textbf{do}\ \{f \leftarrow mc; \mathit{unStateC}\ f\ s\})))\\
&= \quad \{\downarrow \text{ Expanding} \gg\!= \text{for } \mathit{Cont} \}\\
&\quad \mathit{Fused}\ (\mathit{StateC}\ (\lambda s \to \mathit{Cont}\ (\lambda q \to \mathit{runCont}\ (\mathit{Cont}\ (\lambda t \to c_2\ p\ (\lambda a_2 \to\\
&\qquad t\ (\mathit{unFused}\ (k\ a_2)))))(\lambda f \to \mathit{runCont}\ (\mathit{unStateC}\ f\ s)\ q)))\\
&= \quad \{\downarrow \text{ Applying functions} \}\\
&\qquad \mathit{Fused}\ (\mathit{StateC}\ (\lambda s \to \mathit{Cont}\ (\lambda q \to c_2\ p\ (\lambda a_2 \to\\
&\qquad\quad \mathit{runCont}\ (\mathit{unStateC}\ (\mathit{unFused}\ (k\ a_2))\ s)\ q)))\\
&= \quad \{\downarrow \text{ Letting } k'\ a_2\ s = \mathit{runCont}\ (\mathit{unStateC}\ (\mathit{unFused}\ (k\ a_2))\ s) \}\\
&\quad \mathit{Fused}\ (\mathit{StateC}\ (\lambda s \to \mathit{Cont}\ (\lambda q \to c_2\ p\ (\lambda a_2 \to k'\ a_2\ s\ q))))
\end{aligned}
$$

E.3 Local State Semantics from the Tensor

The following is the proof for our claim in Remark 6.1 that the laws of the local state semantics from [Pauwels et al. 2019] can be derived from the laws of the tensor of mutable state and nondeterminism. The laws in [Pauwels et al. 2019] are

$$m \gg\!= (\backslash\_ \to \mathit{fail}) = \mathit{fail} \tag{42}$$

$$m \gg\!= (\lambda x \to f_1\ x \sqcap f_2\ x) = (m \gg\!= f_1) \sqcap (m \gg\!= f_2) \tag{43}$$

where m ranges over computations in the combined theory. The first equation includes a nullary operation *fail* that fails a branch of nondeterminism subject to the equations

$$\mathit{fail} \sqcap m = \mathit{fail} \qquad m \sqcap \mathit{fail} = \mathit{fail} \tag{44}$$

which we did not include in the theory of nondeterminism in this paper. But this is easily fixable: *ndetH* can be extended to handle *fail* by returning an empty set, and it can be verified to respect the laws of *fail*. In the following, we show that for any $m :: \mathit{Free}\ (\mathit{State}_s + \mathit{NDet})\ a$, (42) and (43) hold when applying *handle h* to both sides of the equations, for any handler h that is correct for the tensor $\mathit{State}_s \otimes \mathit{NDet}$ and any $m :: \mathit{Free}\ (\mathit{State}_s + \mathit{NDet})\ a$.

For (42), we prove by induction on m: if m is $\mathit{Var}\ x$ for some x, $\mathit{Var}\ x \gg\!= \backslash\_ \to \mathit{fail} = \mathit{fail}$ holds directly. If m is $O\ p\ k$ for some O, p and k, and if O is an operation from State_s,

$$
\begin{aligned}
&\ \mathit{handle}\ h\ (O\ p\ k \gg\!= \backslash\_ \to \mathit{fail})\\
&= \mathit{handle}\ h\ (\textbf{do}\ \{O\ p\ k; \mathit{fail}\})\\
&= \quad \{\downarrow \text{ Handler } h \text{ respects the tensor of } \mathit{NDet} \text{ and } \mathit{State}_s \}\\
&\quad \mathit{handle}\ h\ (\textbf{do}\ \{x \leftarrow \mathit{fail}; O\ p\ k; \mathit{return}\ x\})\\
&= \quad \{\downarrow\ \mathit{fail} \text{ is a nullary operation} \}\\
&\quad \mathit{handle}\ h\ \mathit{fail}
\end{aligned}
$$

If O is an operation from *NDet*, the calculation above still holds because *fail* is commutative with any operations of *NDet* following the laws (44).

For (43), we also prove by induction on m: if m is $\mathit{Var}\ x$ for some x,

$$
\begin{aligned}
&\ m \gg\!= (\lambda x \to f_1\ x \sqcap f_2\ x)\\
&= \mathit{Var}\ x \gg\!= (\lambda x \to f_1\ x \sqcap f_2\ x)\\
&= f_1\ x \sqcap f_2\ x\\
&= (m \gg\!= f_1) \sqcap (m \gg\!= f_2)
\end{aligned}
$$

if m is some $O\ p\ k$ and if O is from $State_s$,

$$\begin{aligned}
& \mathit{handle}\ h\ (m \ggg (\lambda x \to f_1\ x \sqcap f_2\ x)) \\
=\ & \mathit{handle}\ h\ (O\ p\ k \ggg (\lambda x \to f_1\ x \sqcap f_2\ x)) \\
=\ & \mathit{handle}\ h\ (O\ p\ (\lambda a \to f_1\ (k\ a) \sqcap f_2\ (k\ a))) \\
=\ & \quad \{\Downarrow \text{ The commutativity of state operation } O \text{ and } \sqcap\ \} \\
& \mathit{handle}\ h\ (O\ p\ (f_1 \cdot k) \sqcap O\ p\ (f_2 \cdot k)) \\
=\ & \mathit{handle}\ h\ ((m \ggg f_1) \sqcap (m \ggg f_2))
\end{aligned}$$

Additionally, the calculation above still holds when O is some operation from *NDet* because $\sqcap$ is commutative with any operation of *NDet* following the laws of *NDet*.

E.4 Correctness for the Global State Semantics

The following is a proof for the claim in Remark 6.2 that the composite handler $st\ s \diamond ndetH$ is a correct closed handler of the global state semantics from [Pauwels et al. 2019]. In the *put-or* law,

$$(\mathit{Put}\ s\ (\lambda() \to m)) \sqcap n = \mathit{Put}\ s\ (\lambda() \to m \sqcap n) \tag{45}$$

Operation *Put* is from the second handler $st\ s$, and the modular carrier of first handler *ndetH* is *FreeEM*. Thus by Lemma 5.7, the clause for *Put* of $st\ s \diamond ndetH$ is

$$\begin{aligned}
& \overline{c_{put}}\ p_2\ k \\
=\ & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to c_{put}\ p_2\ (\lambda a_2 \to k'\ a_2\ q)))) \\
& \quad \{\Downarrow\ c_{put} \text{ is the clause for } \mathit{Put} \text{ of } \mathit{stH}\ \} \\
=\ & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda q \to \mathit{StateC}\ (\lambda s \to \mathit{unStateC}\ (k'\ ()\ q)\ p_2)
\end{aligned}$$

where $k' = \mathit{runCont} \cdot \mathit{unFreeEM} \cdot \mathit{unFused} \cdot k$. And the clause for $(\sqcap)$, i.e. *Coin* is

$$\begin{aligned}
\overline{c_{coin}}\ ()\ k = \mathit{Fused}\ (\mathit{FreeEM}\ (\mathbf{do}\ & l_1 \leftarrow \mathit{unFreeEM}\ (\mathit{unFused}\ (k\ \mathit{True})) \\
& l_2 \leftarrow \mathit{unFreeEM}\ (\mathit{unFused}\ (k\ \mathit{False})) \\
& \mathit{return}\ (l_1 \cup l_2)))
\end{aligned}$$

For any m, n, plugging in $\overline{c_{put}}$ and $\overline{c_{coin}}$ in (45) gives

$$\begin{aligned}
& (\mathit{Put}\ s\ (\lambda() \to n)) \sqcap m \\
=\ & \overline{c_{coin}}\ ()\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ \overline{c_{put}}\ s\ (\lambda() \to m)\ \mathbf{else}\ n) \\
=\ & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathbf{do}\ l_1 \leftarrow \mathit{unFusedEM}\ (\mathit{unFused}\ (\overline{c_{put}}\ s\ (\lambda() \to m))) \\
& \qquad\qquad\qquad\quad l_2 \leftarrow \mathit{unFreeEM}\ (\mathit{unFused}\ n) \\
& \qquad\qquad\qquad\quad \mathit{return}\ (l_1 \cup l_2))) \\
& \quad \{\Downarrow \text{ Letting } m' = \mathit{unFreeEM}\ (\mathit{unFused}\ m) \text{ and } n' = \mathit{unFreeEM}\ (\mathit{unFused}\ n)\ \} \\
=\ & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathbf{do}\ l_1 \leftarrow \mathit{Cont}\ (\lambda q \to \mathit{StateC}\ (\backslash_ \to \mathit{unStateC}\ (\mathit{runCont}\ m'\ q)\ s)) \\
& \qquad\qquad\qquad\quad l_2 \leftarrow n' \\
& \qquad\qquad\qquad\quad \mathit{return}\ (l_1 \cup l_2))) \\
=\ & \quad \{\Downarrow \text{ Expanding } \mathbf{do}\text{-notation for } \mathit{Cont}\ \} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda t \to \mathit{StateC}\ (\backslash_ \to \mathit{unStateC}\ (m'\ (\lambda l_1 \to n'\ (\lambda l_2 \to t\ (l_1 \cup l_2))))\ s)))) \\
& \quad \{\Uparrow \text{ Expanding } \mathbf{do}\text{-notation for } \mathit{Cont}\ \} \\
=\ & \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda t \to \mathit{StateC}\ (\backslash_ \to \mathit{unStateC}\ (\\
& \quad \mathit{runCont}\ (\mathit{unFreeEM}\ (\mathit{unFused}\ (\mathbf{do}\ \{ l_1 \leftarrow m'; l_2 \leftarrow n'; \mathit{return}\ (l_1 \cup l_2)\ \})))\ t)\ s)))) \\
=\ & \quad \{\Uparrow \text{ Expanding } \overline{c_{coin}}\ \} \\
& \mathit{Fused}\ (\mathit{FreeEM}\ (\mathit{Cont}\ (\lambda t \to \mathit{StateC}\ (\backslash_ \to \mathit{unStateC}\ (
\end{aligned}$$

$$
\begin{aligned}
&\quad \mathit{runCont}\ (\mathit{unFreeEM}\ (\mathit{unFused}\ (\overline{c_{coin}}\ ()\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ m\ \mathbf{else}\ n))))\ t)\ s))))\\
&= \quad \{\uparrow \text{ Exanpanding } \overline{c_{put}}\,\}\\
&\quad \overline{c_{put}}\ s\ (\lambda() \to \overline{c_{coin}}\ ()\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ m\ \mathbf{else}\ n))\\
&= \mathit{Put}\ s\ (\lambda() \to m \sqcap n)
\end{aligned}
$$

This establishes (45).

E.5 Correctness of the Writer Handler

Lemma E.1. *Handler wtH in Section 6.2 is a correct open handler of the theory of writer effect with equation wtAdd.*

Proof. The accumulation law (see Section 6.2) can be formalised by

$$
\begin{aligned}
&\mathit{wtAdd} :: \mathit{Monoid}\ w \Rightarrow \mathit{Equation}_C\ (\mathit{Writer}\ w)\ (w, w)\ ()\\
&\mathit{wtAdd} = \mathit{Eqn}_C\ \mathit{lhs}\ \mathit{rhs}\ \mathbf{where}\\
&\quad \mathit{lhs}\ \mathit{alg}\ (w_1, w_2)\ k = \mathit{alg}\ (\mathit{Tell}\ w_1\ (\lambda() \to \mathit{alg}\ (\mathit{Tell}\ w_2\ k)))\\
&\quad \mathit{rhs}\ \mathit{alg}\ (w_1, w_2)\ k = \mathit{alg}\ (\mathit{Tell}\ (w_1 \diamond w_2)\ k)
\end{aligned}
$$

Let *lhs* and *rhs* be the two sides of the equation, $w_1, w_2 :: w$ and $k :: () \to \mathit{FreeEM}\ (a, w)\ m$, then

$$
\begin{aligned}
&\quad \mathit{lhs}\ (\mathit{alg}\ \mathit{wtH})\ (w_1, w_2)\ k\\
&= \quad \{\downarrow \text{ definition of } \mathit{lhs}\,\}\\
&\quad \mathit{alg}\ \mathit{wtH}\ (\mathit{Tell}\ w_1\ (\mathit{alg}\ \mathit{wtH}\ (\mathit{Tell}\ w_2\ (k\ ()))))\\
&= \quad \{\downarrow \text{ definition of } \mathit{alg}\ \mathit{wtH} \text{ on } \mathit{Tell}\,\}\\
&\quad \mathit{alg}\ \mathit{wtH}\ (\mathit{Tell}\ w_1\ (\mathit{FreeEM}\ (\mathbf{do}\ \{(a, w) \leftarrow \mathit{unFreeEM}\ (k\ ())\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (a, w_2 \diamond w)\,\})))\\
&= \quad \{\downarrow \text{ definition of } \mathit{alg}\ \mathit{wtH} \text{ on } \mathit{Tell}\,\}\\
&\quad \mathit{FreeEM}\ (\mathbf{do}\ \{(b, u) \leftarrow (\mathbf{do}\ \{(a, w) \leftarrow \mathit{unFreeEM}\ (k\ ())\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (a, w_2 \diamond w)\,\})\\
&\qquad\qquad\qquad \mathit{return}\ (b, w_1 \diamond u)\,\})\\
&= \quad \{\downarrow \text{ monadic properties}\,\}\\
&\quad \mathit{FreeEM}\ (\mathbf{do}\ \{(a, w) \leftarrow \mathit{unFreeEM}\ (k\ ())\\
&\qquad\qquad\quad \mathit{return}\ (a, w_1 \diamond (w_2 \diamond w))\,\})\\
&= \quad \{\downarrow \text{ monoid law: } w_1 \diamond (w_2 \diamond w) = (w_1 \diamond w_2) \diamond w\,\}\\
&\quad \mathit{alg}\ \mathit{wtH}\ (\mathit{Tell}\ (w_1 \diamond w_2)\ (k\ ()))\\
&= \quad \{\uparrow \text{ definition of } \mathit{rhs}\,\}\\
&\quad \mathit{rhs}\ (\mathit{alg}\ \mathit{wtH})\ (w_1, w_2)\ k
\end{aligned}
$$

□

Theorem E.2 (Theorem 6.6). *Both stH s ⋄ wtH and wtH ⋄ stH are correct open handlers of the tensor of mutable state and writer.*

Proof. Following Theorem 6.5, $\mathit{wtH} \diamond \mathit{stH}\ s$ is a correct open handler of the tensor. To show that $\mathit{stH}\ s \diamond \mathit{wtH}$ is correct, we apply Corollary 6.2. For $op_1 = \mathit{Tell}$ and $op_2 = \mathit{Put}$, we have

$$
\begin{aligned}
c_1'\ p\ k &= \mathit{runCont}\ (\mathit{unFreeEM}\ (c_1\ p\ (\mathit{FreeEM} \cdot \mathit{Cont} \cdot k))) \quad \{\downarrow \text{ definition of } c_1\,\}\\
&= \mathit{runCont}\ (\mathbf{do}\ \{(a, u) \leftarrow \mathit{Cont}\ (k\ ()); \mathit{return}\ (a, p \diamond u)\,\})\\
&\quad \{\downarrow \text{ definition of } \ggg \text{ for the continuation monad}\,\}\\
&= \lambda q \to k\ ()\ (\lambda(a, u) \to q\ (a, p \diamond u))
\end{aligned}
$$

$$c_2\ p\ k = \mathit{StateC}\ (\lambda s \to \mathit{unStateC}\ (k\ ())\ p)$$

and we establish (21) by the following calculation:

$$\begin{aligned}
& c_1'\ p_1\ (\lambda a_1 \to (\lambda q \to c_2\ p_2\ (\lambda a_2 \to k'\ a_1\ a_2\ q))) && \{\Downarrow \text{ expanding } c_2\ \} \\
={} & c_1'\ p_1\ (\lambda a_1 \to (\lambda q \to \mathit{StateC}\ (\lambda s \to \mathit{unStateC}\ (k'\ a_1\ ()\ q)\ p_2))) && \{\Downarrow \text{ expanding } c_1\ \} \\
={} & \lambda q \to \mathit{StateC}\ (\lambda s \to \mathit{unStateC}\ (k'\ ()\ ()\ (\lambda(a,u) \to q\ (a, p_1 \diamond u)))\ p_2) && \{\Uparrow \text{ expanding } c_1\ \} \\
={} & \lambda q \to \mathit{StateC}\ (\lambda s \to \mathit{unStateC}\ (c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ ())\ q)\ p_2) && \{\Uparrow \text{ expanding } c_2\ \} \\
={} & \lambda q \to c_2\ p_2\ (\lambda a_2 \to c_1'\ p_1\ (\lambda a_1 \to k'\ a_1\ a_2)\ q)
\end{aligned}$$

For $op_1 = \mathit{Tell}$ and $op_2 = \mathit{Get}$, we can similarly show that both sides of Equation 21 are equal to

$$\lambda q \to \mathit{StateC}\ (\lambda s \to k'\ ()\ s\ (\lambda(a,u) \to q\ (a, p_1 \diamond u))\ s)$$

By Corollary 6.2 we conclude that $\mathit{stH}\ s \diamond \mathit{wtH}$ correctly handles the tensor. □

F A SIMPLE LANGUAGE FOR MODULAR HANDLERS

This section shows a fine-grained call-by-value [Levy 2003] language λ_M with effect handlers, its type system, and a denotational semantics based on the constructions discussed in this paper. The language is similar to the language core Eff in [Bauer and Pretnar 2015] except that the type system of λ_M requires handlers to work polymorphically in unhandled operations, so handlers in λ_M are always modular handlers.

F.1 Abstract Syntax

Let m, n, p, k, x, and y range over a set of variables, and op range over a set of operation symbols. The types of λ_M are split into *computation types* and *value types*, and a subset of value types are *ground types*, which does not contain functions and handlers.

$$\begin{aligned}
\textit{Ground types} \quad & G, P ::= \amalg_{i\in I}\, G_i \mid \Pi_{i\in I}\, G_i \\
\textit{Value types} \quad & A, B ::= G \mid A \to \underline{C} \mid A \Rightarrow_\Sigma B \mid \amalg_{i\in I}\ A_i \mid \Pi_{i\in I}\, A_i \\
\textit{Computation types} \quad & \underline{C}, \underline{D} ::= M\ A \\
\textit{Effects} \quad & M ::= \mathsf{F}_\Sigma \mid m \\
\textit{Signatures} \quad & \Sigma ::= \{\mathsf{op}_i : P \to G\}_{i\in I}
\end{aligned}$$

where the index set I is always finite. Note that when I is the empty set, the product type $\Pi\{\}$ can be used as the unit type and the coproduct type $\amalg\{\}$ can be used as the empty type. Therefore we do not need these base types in the language.

The terms of λ_M are split into two syntactic categories: pure *values* and potentially effectful *computations*:

$$\begin{aligned}
\textit{Values} \quad & v ::= x \mid \mathsf{inj}_{i\in I}\, v \mid \langle v_i\rangle_{i\in I} \mid \lambda x{:}A.\ c \\
& \quad\ \mid \mathsf{Hdl}_\Sigma\{\mathsf{val}\ \ x \mapsto c \mid (\mathsf{op}\ p\ k \mapsto c)_{\mathsf{op}\in\Sigma}\} \\
\textit{Computations} \quad & c ::= \mathsf{val}\ v \mid \mathsf{op}\ v\ (y.\ c) \mid \mathsf{with}\ v\ \mathsf{handle}\ c \mid v\ v \\
& \quad\ \mid \mathsf{let}\ x = c\ \mathsf{in}\ c \mid \mathsf{match}\ e\ \mathsf{as}\ \{\langle x_i\rangle_{i\in I} \mapsto c\} \\
& \quad\ \mid \mathsf{match}\ e\ \mathsf{as}\ \{\mathsf{inj}_i\ x_i \mapsto c_i\}_{i\in I}
\end{aligned}$$

F.2 Type System

Let Γ range over finite maps from variables to *value types* and Δ be finite set of variables. We say a type is *well-formed* under Δ if all effect variables m in the type are contained in Δ, and well-formedness is signified by judgements

$$\Delta \vdash A \qquad \text{and} \qquad \Delta \vdash \underline{C}$$

for both value types and computation types. We also have two typing judgements:

$$\Delta \mid \Gamma \vdash v : A \qquad\qquad \Delta \mid \Gamma \vdash c : \underline{C}$$

where A, $\underline{C}$ and $\Gamma(x)$ for each $x \in \mathrm{dom}(\Gamma)$ are well-formed under Δ. The typing rules for values types are the following:

$$\frac{(x : A) \in \Gamma}{\Delta \mid \Gamma \vdash x : A} \qquad \frac{\Delta \mid \Gamma \vdash x : A_j}{\Delta \mid \Gamma \vdash \mathtt{inj}_j\ v : \coprod_{i\in I} A_i} \qquad \frac{\Delta \mid \Gamma \vdash x_i : A_i \text{ for each } i \in I}{\Delta \mid \Gamma \vdash \langle x_i \rangle_{i\in I} : \Pi_{i\in I} A_i}$$

$$\frac{\Delta \mid \Gamma,\ x : A \vdash c : \underline{C} \qquad x \notin \mathrm{dom}(\Gamma)}{\Delta \mid \Gamma \vdash \lambda x{:}A.\ c : A \to \underline{C}}$$

$$\frac{\begin{array}{c} m \notin \Delta \qquad x \notin \mathrm{dom}(\Gamma) \qquad \{p_i,\ k_i\}_{\mathtt{op}_i \in \Sigma} \cap \mathrm{dom}(\Gamma) = \emptyset \\ \Delta,\ m \mid \Gamma,\ x : A \vdash c_0 : m\ B \qquad \Big(\Delta,\ m \mid \Gamma,\ p_i : P_i,\ k_i : (A_i \to m\ B) \vdash c_i : m\ B\Big)_{(\mathtt{op}_i : P_i \to A_i) \in \Sigma} \end{array}}{\Delta \mid \Gamma \vdash \mathtt{Hdl}_\Sigma\{\mathtt{val}\ x \mapsto c_0 \mid (\mathtt{op}_i\ p_i\ k_i \mapsto c_i)_{\mathtt{op}_i \in \Sigma}\} : A \Rightarrow_\Sigma B}\ \text{T-HDL}$$

and the typing rules for computations are

$$\frac{\Delta \mid \Gamma \vdash v : A}{\Delta \mid \Gamma \vdash \mathtt{val}\ \ v : M\,A}\ \text{T-RET} \qquad \frac{(\mathtt{op} : P \to A) \in \Sigma \qquad \Delta \mid \Gamma \vdash v : P \qquad \Delta \mid \Gamma,\ y : A \vdash c : \mathsf{F}_\Sigma\, B}{\Delta \mid \Gamma \vdash \mathtt{op}\ v\ (y.\ c) : \mathsf{F}_\Sigma\, B}\ \text{T-OP}$$

$$\frac{\Delta \mid \Gamma \vdash v : A \Rightarrow_\Sigma B \qquad \Delta \mid \Gamma \vdash c : \mathsf{F}_{\Sigma'}\, A}{\Delta \mid \Gamma \vdash \mathtt{with}\ v\ \mathtt{handle}\ c : \mathsf{F}_{\Sigma' \setminus \Sigma}\ B}\ \text{T-WITH} \qquad \frac{\Delta \mid \Gamma \vdash v_1 : A \to \underline{C} \qquad \Delta \mid \Gamma \vdash v_2 : A}{\Delta \mid \Gamma \vdash v_1\ v_2 : \underline{C}}\ \text{T-APP}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Delta \mid \Gamma \vdash c_1 : M\,A \qquad \Delta \mid \Gamma,\ x : A \vdash c_2 : M\,B}{\Delta \mid \Gamma \vdash \mathtt{let}\ x = c_1\ \mathtt{in}\ c_2 : M\,B}\ \text{T-BIND}$$

$$\frac{\Delta \mid \Gamma \vdash v : \Pi_{i\in I} A_i \qquad \{x_i\}_{i\in I} \cap \mathrm{dom}(\Gamma) = \emptyset \qquad \Delta \mid \Gamma, (x_i : A_i)_{i\in I} \vdash c : \underline{C}}{\Delta \mid \Gamma \vdash \mathtt{match}\ v\ \mathtt{as}\ \{\langle x_i \rangle_{i\in I} \mapsto c\} : \underline{C}}$$

$$\frac{\Delta \mid \Gamma \vdash v : \coprod_{i\in I} A_i \qquad \{x_i\}_{i\in I} \cap \mathrm{dom}(\Gamma) = \emptyset \qquad \big(\Delta \mid \Gamma, x_i : A_i \vdash c_i : \underline{C}\big)_{i\in I}}{\Delta \mid \Gamma \vdash \mathtt{match}\ v\ \mathtt{as}\ \{\mathtt{inj}_i\ x_i \mapsto c_i\}_{i\in I} : \underline{C}}$$

F.3 Denotational Semantics

In the following we show a denotational semantics of λ_M by translating typing derivations to Haskell functions. As we mentioned in Section 2, we meant to use Haskell as a total language denoting constructions around the category of sets. Thus the translation can be understood as a set-theoretic semantics of λ_M.

Denotation of Types. Assuming there is an injective map ρ from variables in λ_M to Haskell type variables of kind $* \to *$. For any well-formed type A or $\underline{C}$ under Δ, its semantics is a Haskell type $[\![A]\!]_\rho$ or $[\![\underline{C}]\!]_\rho$, in which variables in $\{ \rho(m) \mid m \in \Delta \}$ may occur freely. (Categorically, the meaning of a type is a profunctor $(\mathit{Mnd}^{|\Delta|})^{op} \times \mathit{Mnd}^{|\Delta|} \to \mathit{Set}$ where $\mathit{Mnd}^{|\Delta|}$ is the $|\Delta|$-fold product category of the category of set monads).

Unsurprisingly, the product type Π denotes tuples in Haskell: $[\![\Pi_{i\in I} A_i]\!]_\rho = ([\![A_i]\!]_\rho)_{i\in I}$. Coproduct types $\amalg$ denotes finite coproducts of Haskell types too, but Haskell does not have a syntax for nameless finite coproducts, so we translate $[\![\amalg_{i\in I} A_i]\!]_\rho$ into a datatype declaration:

$$\mathbf{data}\ T = (\mathit{Inj}_i\ [\![A_i]\!]_\rho)_{i\in I}$$

where T is a fresh name in the translation. Function types $A \to \underline{C}$ denotes functions $[\![A]\!]_\rho \to [\![\underline{C}]\!]_\rho$. Signatures Σ denote signature functors as in Section 2.1:

$$\mathbf{data}\ S\ x = \Big(\mathit{Op}_i\ [\![P_i]\!]_\rho\ ([\![A_i]\!]_\rho \to x)\Big)_{(\mathsf{op}_i : P_i \to A_i)\in\Sigma}$$

where S and Op_i are fresh names for the translation. Handler types $A \Rightarrow_\Sigma B$ denotes the datatype

$$\mathit{MHandler}\ [\![\Sigma]\!]_\rho\ (\mathit{FreeEM}\ [\![B]\!]_\rho)\ [\![A]\!]_\rho\ [\![B]\!]_\rho$$

of modular handlers described in Section 3.3. For computations, $[\![\mathsf{F}_\Sigma\ A]\!]_\rho$ is precisely $\mathit{Free}\ [\![\Sigma]\!]_\rho\ [\![A]\!]_\rho$, and $[\![m\ A]\!]_\rho$ is $\rho(m)\ [\![A]\!]_\rho$.

Denotation of Terms. Given a context Γ mapping to well-formed types under Δ, its meaning $[\![\Gamma]\!]_\rho$ is the product type of the meaning of the types that Γ mapped to. Then the meaning of a typing derivation $\Delta \mid \Gamma \vdash v : A$ or $\Delta \mid \Gamma \vdash c : \underline{C}$ is some Haskell function of type

$$\forall(\rho(m))_{m\in\Delta}.\ (\mathit{Monad}\ \rho(m))_{m\in\Delta}\ \Rightarrow\ [\![\Gamma]\!]_\rho \to [\![A]\!]_\rho$$

or

$$\forall(\rho(m))_{m\in\Delta}.\ (\mathit{Monad}\ \rho(m))_{m\in\Delta}\ \Rightarrow\ [\![\Gamma]\!]_\rho \to [\![\underline{C}]\!]_\rho$$

For most cases in the type system, their meanings are standard (see for example [Levy 2003] and [Bauer and Pretnar 2015]), thus we only describe the non-standard cases here:

- For rule T-Hdl, the meaning of a handler value $[\![\mathsf{Hdl}_\Sigma\{\mathsf{val}\ x \mapsto c_0 \mid (\mathsf{op}_i\ p_i\ k_i \mapsto c_i)_{\mathsf{op}_i\in\Sigma}\}]\!]_\rho$ is a function f of type

$$\begin{aligned}&\forall(\rho(m))_{m\in\Delta}.\ (\mathit{Monad}\ \rho(m))_{m\in\Delta}\\ \Rightarrow\ &[\![\Gamma]\!]_\rho \to \mathit{MHandler}\ [\![\Sigma]\!]_\rho\ (\mathit{FreeEM}\ [\![B]\!]_\rho)\ [\![A]\!]_\rho\ [\![B]\!]_\rho\end{aligned}$$

  defined by

$$\begin{aligned}f\ g = \mathit{MHandler}\ \{&\mathit{gen} = (\lambda a \to [\![c_0]\!]_\rho\ (g, a))\\ ,\ &\mathit{alg} = (\lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{(\mathit{Op}_i\ p\ k) \to [\![c_i]\!]_\rho\ (g, p, k)\\ &\qquad\qquad\qquad\qquad ;\ldots\})\\ ,\ &\mathit{run} = \mathit{unFusedEM}\}\end{aligned}$$

- For rule T-Ret, $[\![M\ A]\!]_\rho$ is either $\mathit{Free}\ [\![\Sigma]\!]_\rho\ [\![A]\!]_\rho$ or $\rho(m)\ [\![A]\!]_\rho$, and $\rho(m)$ is given a monad constraint when defining the meaning of $\mathsf{val}\ v$. Thus we can interpret $\mathsf{val}\ v$ by the *return* of $[\![M]\!]_\rho$:

$$[\![\mathsf{val}\ v]\!]_\rho\ g = \mathit{return}\ ([\![v]\!]_\rho\ g)$$

- For rule T-Bind, it is interpreted by $\ggg$ of $[\![M]\!]_\rho$:

$$[\![\mathsf{let}\ x = c_1\ \mathsf{in}\ c_2]\!]_\rho\ g = ([\![c_1]\!]_\rho\ g) \ggg (\mathit{curry}\ ([\![c_2]\!]_\rho\ g))$$

- For rule T-Op, it is interpreted by the *Op* constructor of the free monad *Free*:

$$[\![\mathsf{op}\ v\ (y.\ c)]\!]_\rho\ g = \mathit{Op}\ ([\![v]\!]_\rho\ g)\ (\lambda y \to [\![c]\!]_\rho\ (g, y))$$

- For rule T-With, it is interpreted by the *handle* function in Section 3:

$$[\![\mathsf{with}\ v\ \mathsf{handle}\ c]\!]_\rho\ g = \mathit{handle}\ ([\![v]\!]_\rho\ g)\ ([\![c]\!]_\rho\ g)$$

REFERENCES

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? arXiv:cs.LO/1807.05923 https://arxiv.org/abs/1807.05923

Richard Bird and Oege de Moor. 1997. *Algebra of Programming*. London.

Ralf Hinze. 2005. THEORETICAL PEARL Church numerals, twice! *Journal of Functional Programming* 15, 1 (2005), 1–13. https://doi.org/10.1017/S0956796804005313

Ralf Hinze. 2013. Adjoint folds and unfolds—An extended study. *Science of Computer Programming* 78, 11 (2013), 2108 – 2159. https://doi.org/10.1016/j.scico.2012.07.011

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

Koen Pauwels, Tom Schrijvers, and Shin-Cheng Mu. 2019. Handling Local State with Global State. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Springer International Publishing, Cham, 18–44. https://doi.org/10.1007/978-3-030-33636-3_2

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures*, Mogens Nielsen and Uffe Engberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–356. https://doi.org/10.1007/3-540-45931-6_24

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

Janis Voigtländer. 2009. Free theorems involving type constructor classes. *ACM SIGPLAN Notices* 44, 9 (2009), 173–184. https://doi.org/10.1145/1631687.1596577

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, Vol. 19. ACM Press, New York, New York, USA, 347–359. https://doi.org/10.1145/99370.99404

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 302–322. https://doi.org/978-3-319-19797-5_15