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 correct order of composition is a non-trivial 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 using 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.

CCS Concepts: • **Theory of computation → Program reasoning**; **Control primitives**.

Additional Key Words and Phrases: Haskell, fusion, modular handlers, CPS transformation

ACM Reference Format:
Zhixuan Yang and Nicolas Wu. 2021. Reasoning about Effect Interaction by Fusion. *Proc. ACM Program. Lang.* 5, ICFP, Article 73 (August 2021), 29 pages. https://doi.org/10.1145/3473578

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

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.

2475-1421/2021/8-ART73
https://doi.org/10.1145/3473578

```
h_ST = handler {
  val x    ↦ val (fun s ↦ val (x, s)),
  get () k ↦ val (fun s ↦ k s s),
  put s′ k ↦ val (fun s ↦ k () s′)}
```

```
h_ND = handler {
  val x    ↦ val [x],
  coin () k ↦ {let l₁ = k True in
               let l₂ = k False in
               val (l₁ ++ l₂)}}
```

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

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.

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 two *operations get* and *put* for reading and writing the state, and *equations* that characterise 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

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

or nondeterminism gets handled first

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

and it is well known that the two orders result in different handling behaviours. On the specification side, the theories *State* and *NDet* can be composed into one single theory too. One desirable combination is the *commutative tensor* [Hyland et al. 2006], 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*:

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

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

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 `HNdSt` and `HStNd` 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 $k\ b$, where k 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 $k\ b = \textbf{do}\ \{a \leftarrow op\ p; k'\ a\}$, 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, but the common structure in these equations are not exploited.

1.1 A Taste of the Results

The aim of this paper is to develop techniques for proving the correctness of composite handlers (by *correctness* of handlers, we always mean validating the expected equations) 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. We argue that verifying these conditions is much easier than proving the correctness directly based on the definitions.

To provide a taste of the techniques developed in the paper, consider the running example in Figure 1. Suppose the programmer has proved that the handlers h_{ST} and h_{ND} are correct handlers for effect theories *State* and *NDet* respectively, and that the programmer wants to show that the composite handler HStNd correctly handles the commutative tensor of *State* and *NDet*.

According to the definition, the commutative tensor of *State* and *NDet* inherits all the equations in the two sub-theories, so the programmer first needs to show that the composite handler HStNd respects these equations. Our results can prove this almost for free: Theorem 5.5 tells us that any equations respected by modular handlers separately are still respected by their composite, and indeed both h_{ST} and h_{ND} are modular handlers. Thus without any further work, we immediately know that HStNd respects the equations from *State* and *NDet* because these equations are respected by h_{ST} and h_{ND} separately.

Also, the programmer needs to show that HStNd validates the commutativity equations (1, 2) in the tensor. The conventional way to show this is to do an induction on the free computation k, apply HStNd to both sides of the equations, and then perform equational reasoning to establish the required equality. Although proving the correctness of composite handlers in this way is typically not difficult, the calculation can be tedious. This paper offers a more efficient way to do this: let c_{put}, c_{get} and c_{coin} be the *clauses* (in a sense made clear in Section 5.3) of the two handlers h_{ST} and h_{ND}:

$$c_{put}\ p_1\ k = \lambda s \rightarrow k\ ()\ p_1 \qquad\qquad c_{get}\ ()\ k = \lambda s \rightarrow k\ s\ s$$
$$c_{coin}\ ()\ k = \textbf{do}\ \{l_1 \leftarrow k\ \mathit{True}; l_2 \leftarrow k\ \mathit{False}; \mathit{return}\ (l_1 \mathbin{+\!\!+} l_2)\}$$

Then Theorem 6.1 says that the composite handler HStNd respects the required equations (1, 2) if each clause $c_{ST} \in \{c_{put}, c_{get}\}$ of h_{ST} and each clause $c_{ND} \in \{c_{coin}\}$ of h_{ND} satisfy the following

Table 1. Results for various combinations of effect theories and applications

Effect Combinations	**Results**	**Examples**
Sum	Theorem 5.5	All composites of modular handlers
Commutative tensor	Theorem 6.1	State and nondeterminism (Section 6.1) State and writer (Section 6.2)
Distributive tensor	Theorem 7.1	Probabilistic and nondeterministic choices (Section 7.1)
Application-specific interactions	Remark 5.3	Put-or law (Remark 6.2)

equation for all p_1, p_2, k (the types of the variables in the equation can be ignored for now):

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

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

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

Finally, every equation respected by the composite of two modular handlers is automatically a term congruence under the handler (Remark 5.2), which is an important property for reasoning about effectful programs with these equations. This example will be studied in detail in Section 6.1, and the point is that this is much less of a burden than proving directly from the definitions.

The key technique underlying the results described above 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 syntax trees of programs. By the properties of catamorphisms, $handle\ (h_2 \diamond h_1)$ respects an effect theory if $h_2 \diamond h_1$ respects it, from which we calculate conditions for $handle\ h_2 \cdot handle\ h_1$ to respect various combinations of effect theories, such as Theorem 5.5 and Theorem 6.1 used in the above example, and more results are listed in Table 1.

1.2 Contributions

After fixing notation for preliminary concepts (Section 2), Schrijvers et al. [2019]'s modular handlers are motivated and recalled (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);

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

$$\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\quad \mathit{return} :: a \to m\ a \qquad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$$

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

- 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:

$$\mathit{fmap}\ \mathit{id} = \mathit{id} \qquad\qquad \mathit{fmap}\ g \cdot \mathit{fmap}\ h = \mathit{fmap}\ (g \cdot h)$$

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

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

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

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 $join :: m\ (m\ a) \to m\ a$ is defined by $join\ m = m \gg\!= id$. 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 $f\ x$. Additionally, Haskell supports the do-notation **do** $x \leftarrow m; b$ as a syntactic sugar for $m \gg\!= (\lambda x \to b)$.

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

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

Intuitively, an element of $Free\ f\ v$ 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 $alg :: f\ c \to c$ and function $gen :: v \to c$, there is a function *fold* (also known as *catamorphism*) that recursively reduces $Free\ f\ v$ to the carrier c of *alg*:

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

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

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

Intuitively, $return\ x$ 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 model 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 in our Haskell notation as the basis for our development.

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. A computational 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

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

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). Another theoretically useful algebraic theory 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*:

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

where binary operator $\doteq$ is the constructor. Free value variables and computation 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:

$$\begin{array}{l}
putPutEq :: Equation\ State_s\ (s, s)\ () \\
putPutEq = (lhs \doteq rhs)\ \textbf{where} \\
\quad lhs, rhs :: (s, s) \to Free\ State_s\ () \\
\quad lhs\ (u, u') = Op\ (Put\ u\ (\lambda() \to Op\ (Put\ u'\ (\lambda() \to Var\ ())))) \\
\quad rhs\ (u, u') = Op\ (Put\ u'\ (\lambda() \to Var\ ()))
\end{array}$$

In the main text of this paper, we will stick to the informal form of equations as in (10) for brevity, and the formal form will only be used in proofs. It is straightforward to convert an informal equation to the formal form $Equation\ \Sigma\ \Gamma\ v$ 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

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

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

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:

$$p \sqcap p = p \qquad\qquad p \sqcap q = q \sqcap p \qquad\qquad p \sqcap (q \sqcap r) = (p \sqcap q) \sqcap r$$

where p, q and r are all free variables of computations. The three equations axiomatise the so-called *internal choice* in the literature of process algebra, thus the symbol $\sqcap$ is used following Hoare [1985a] instead of the seemingly more natural $\sqcup$, which is conventionally used for external choice.

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

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

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 $put\ t\ (get\ k) = put\ t\ (k\ t)$ is respected because for all $k :: s \to s \to a$ and $t :: s$,

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

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*, which are equivalent to *Lawvere theories* that present theories as categories [Plotkin and Power 2004]. 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 in 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 (5) of the signature functors of the sub-theories. Equations of the combined theory have greater freedom of choice. A straightforward choice is just taking the union of the equations of the sub-theories and no more, which is called the *sum* of the sub-theories.

Definition 2.3 (Sum of Theories [Hyland et al. 2006])**.** The *sum* of $T_1 :: Theory\ \Sigma_1$ and $T_2 :: Theory\ \Sigma_2$, 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 of the computation, 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 ×,

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

By passing all operands by a function as 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:

$$\begin{array}{l} O_1\ p_1\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv b\ \mathbf{then} \\ \quad O_2\ p_2\ (\lambda a_2 \to k_2\ a_2)\ \mathbf{else}\ k_1\ a_1) \end{array} = \begin{array}{l} O_2\ p_2\ (\lambda a_2 \to O_1\ p_1\ (\lambda a_1 \to \\ \quad \mathbf{if}\ a_1 \equiv b\ \mathbf{then}\ k_2\ a_2\ \mathbf{else}\ k_1\ a_1)) \end{array} \tag{12}$$

where computations k_1 and k_2 and values $p_1 :: P_1$, $p_2 :: P_2$ and $b :: A_1$ are free variables. Intuitively, variable b marks the position of the inner computation O_2. Thus (12) implies distributivity laws for all positions of O_2 inside O_1.

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 $\mathit{PChoose} :: \mathit{Real} \rightsquigarrow \mathit{Bool}$ with a *Real* parameter in the range $[0, 1]$. Operation $\mathit{PChoose}\ \theta\ k$ is preferably written in infix notation $p \triangleleft\theta\triangleright q = \mathit{PChoose}\ \theta\ (\lambda b \to \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ q)$ following Hoare [Hoare 1985b]. Letting $\bar{\theta}$ denote $1 - \theta$, theory *Prob* has the following equations:

$$p \triangleleft 1 \triangleright q = p \qquad\qquad p \triangleleft\theta\triangleright p = p \qquad\qquad p \triangleleft\theta\triangleright q = q \triangleleft\bar{\theta}\triangleright p$$

$$p \triangleleft\theta_1\triangleright (q \triangleleft\theta_2\triangleright r) = (p \triangleleft\delta_1\triangleright q) \triangleleft\delta_2\triangleright r \quad (\theta_1 = \delta_1\delta_2, \bar{\delta}_2 = \bar{\theta}_1\bar{\theta}_2)$$

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:

$$p \triangleleft\theta\triangleright (q \sqcap r) = (p \triangleleft\theta\triangleright q) \sqcap (p \triangleleft\theta\triangleright r) \qquad\qquad (p \sqcap q) \triangleleft\theta\triangleright r = (p \triangleleft\theta\triangleright r) \sqcap (q \triangleleft\theta\triangleright r)$$

3 SYNTAX AND SEMANTICS OF COMPUTATIONS

Now that we have theories of effects, we continue to set the stage by showing how one can formalise the syntax and semantics of computations involving effects. 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 $\mathit{Free}\ \Sigma\ a$ (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 $O :: P \leadsto_\Sigma A$, $p :: P$ and $k :: A \to \mathit{Free}\ \Sigma\ a$, 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 $O :: P \leadsto_\Sigma A$, we have a function $O_g :: P \to \mathit{Free}\ \Sigma\ A$, 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. The following computation $\mathit{incr} :: \mathit{Free}\ \mathit{State}_{\mathit{Int}}\ \mathit{Int}$ gets the state, increments it and returns the original value:

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

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 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:

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

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

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

Relation $c \sim_T d$ 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 terms, enclosing them in all contexts $\mathit{Op}\ (O_i\ p\ \_)$ 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 $incr' \sim_{State_{Int}} incr$ 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 $gen :: a \to Free\ \Sigma\ b$ and $alg :: \Sigma\ (Free\ \Sigma\ b) \to Free\ \Sigma\ b$ for some types a and b. We call (gen, alg) *a handler from a to b*. It induces a function $handleTr\ (gen, alg) :: Free\ \Sigma\ a \to Free\ \Sigma\ b$ that applies the handler to a computation $Free\ \Sigma\ a$ by $handleTr\ (gen, alg) = fold\ alg\ gen$, or more explicitly,

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

for each operation O in Σ. The gen function corresponds to the 'return clause' of handlers in EFF [Bauer and Pretnar 2015] that transforms a pure a-value $Var\ x$ 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 all possible results of this operation to a computation of a b-value.

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

$$\begin{aligned} &gen_{ND}\ x &&= return\ \{ x \} \\ &alg_{ND}\ (Inl\ op) &&= Op\ (Inl\ op) \\ &alg_{ND}\ (Inr\ (Coin\ ()\ k)) &&= \mathbf{do}\ \{ l_1 \leftarrow k\ True; l_2 \leftarrow k\ False; return\ (l_1 \cup l_2) \} \end{aligned}$$

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 expected to be handled by it, breaking the modular principle followed by algebraic effects and causing difficulties in reasoning. We demonstrate the problem in the following example.

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

$$\begin{aligned} &gen_{ND}'\ x &&= return\ \{ x \} \\ &alg_{ND}'\ (Inl\ op) &&= Op\ (Inl\ op) \\ &alg_{ND}'\ (Inr\ (Coin\ ()\ k)) &&= \mathbf{do}\ \{ l_1 \leftarrow fold\ alg'\ Var\ (k\ True); l_2 \leftarrow k\ False; return\ (l_1 \cup l_2) \} \\ &\quad \mathbf{where}\ alg'\ x = \mathbf{case}\ x\ \mathbf{of}\ \{ (Inl\ (Put\ s\ k)) \to k\ (); \_ \to Op\ x \} \end{aligned}$$

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 $s \to Free\ (State_s + NDet)\ a$:

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

$$alg_{ST}\ (Inr\ op) \qquad\qquad = Op\ (Inr\ op)$$

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

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

no longer respects the first equation $put\ \ s\ \ (get\ \ k) = put\ \ s\ \ (k\ \ s)$ because the left-hand side is transformed to $Var\ (\lambda s_0 \to k\ s_0)$, while the right-hand side is transformed to $Var\ (\lambda s_0 \to k\ s)$, 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

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

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 increase the level of abstraction by replacing $Free\ sig'$ with an abstract monad *m*:

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

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 every monad *m*,

$$fwd \cdot return = id \qquad\qquad fwd \cdot fmap\ fwd = fwd \cdot join \tag{14}$$

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:

$$\mathit{FreeEM} :: m\ a \rightarrow \mathit{FreeEM}\ a\ m \qquad\qquad \mathit{unFreeEM} :: \mathit{FreeEM}\ a\ m \rightarrow m\ a$$

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

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

The laws of monads in (6) imply that the laws in (14) are satisfied. The name *FreeEM* comes from the fact that $\mathit{FreeEM}\ a\ m \cong m\ a$ with *join* is the *free Eilenberg-Moore algebra* for a. The scheme in (13) is then equivalent to $\mathit{alg} :: \forall m.\ \mathit{Monad}\ m \Rightarrow \mathit{sig}\ (\mathit{FreeEM}\ b\ m) \rightarrow \mathit{FreeEM}\ b\ m$.

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

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

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

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

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

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 $(\mathit{gen}, \mathit{alg}, \mathit{run})$ packed into the following record:

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

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

$$\mathit{handle}\ h = \mathit{run} \cdot \mathit{fold}\ \mathit{alg}'\ \mathit{gen}$$

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

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 an 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:

$$\begin{array}{l}\mathit{ndetH} :: \mathit{MHandler}\ \mathit{NDet}\ (\mathit{FreeEM}\ (\mathit{Set}\ a))\ a\ (\mathit{Set}\ a) \\ \mathit{ndetH} = \mathit{MHandler}\ \{\mathit{gen} = \mathit{gen}_{\mathit{ND}}, \mathit{alg} = \mathit{alg}_{\mathit{ND}}, \mathit{run} = \mathit{unFreeEM}\}\ \textbf{where} \\ \quad \mathit{gen}_{\mathit{ND}}\ a = \mathit{FreeEM}\ (\mathit{return}\ \{\ a\ \}) \\ \quad \mathit{alg}_{\mathit{ND}}\ (\mathit{Coin}\ ()\ k) = \mathit{FreeEM}\ (\textbf{do}\ l_1 \leftarrow \mathit{unFreeEM}\ (k\ \mathit{True}); l_2 \leftarrow \mathit{unFreeEM}\ (k\ \mathit{False}); \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}\ (l_1 \cup l_2))\end{array}$$

Compared to its non-modular counterpart in Example 3.3, $\mathit{alg}_{\mathit{ND}}$ 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 $m\ (s \to m\ a)$, but the outer layer of m is unnecessary, and we can define the following modular handler of $State_s$ with carrier $StateC\ s\ a\ m \cong s \to m\ a$ from Example 3.6:

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

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 $\forall sig'.\ Free\ (\Sigma + sig')\ a \to Free\ sig'\ b$ for some types a and b, we say that f is a *correct open transformation* for T if for all signatures sig', theories $T' :: Theory\ sig'$ and computations $t_1, t_2 :: Free\ (\Sigma + sig')\ a$,

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

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 all computations $t_1, t_2 :: Free\ (\Sigma + Empty)\ A$,

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

where $extract :: Free\ Empty\ a \to a$ is defined by $extract\ (Var\ a) = a$.

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 $h :: MHandler\ \Sigma\ c\ a\ b$ be a modular handler, (a) we call h a correct *open* handler of T if $alg\ h :: Monad\ m \Rightarrow \Sigma\ (c\ m) \to c\ m$ respects (in the sense of Definition 2.1) all equations of T for every monad m, and (b) we call h a correct *closed* handler of T if $alg\ h$ 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 to relate the free monad *Free sig′* and the monad mapping X to the free model of T' generated by X, i.e. *Free 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 $h_1 :: MHandler\ \Sigma_1\ c_1\ x\ y$ and $h_2 :: MHandler\ \Sigma_2\ c_2\ y\ z$ for some modular carriers c_1 and c_2 and types x, y, z. Their composite

$$handle\ h_2 \cdot handle\ h_1 :: \forall sig'.\ Free\ (\Sigma_1 + (\Sigma_2 + sig'))\ x \to Free\ sig'\ z$$

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 $handle\ h_2 \cdot handle\ h_1$ can be more easily understood if we can find some handler $h_3 :: MHandler\ (\Sigma_1 + \Sigma_2)\ c\ x\ z$ for some modular carrier c satisfying $handle\ h_2 \cdot handle\ h_1 = handle\ h_3 \cdot assoc$ where *assoc* and its inverse $assoc^\circ$ is the evident isomorphism between $Free\ (\Sigma_1 + (\Sigma_2 + sig'))$ and $Free\ ((\Sigma_1 + \Sigma_2) + sig')$ for all Σ_1, Σ_2 and sig'. 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 syntax tree of the computation. 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

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

Intuitively, a computation of some a value in the continuation monad $Cont_r\ a \cong (a \to r) \to r$ does not necessarily compute an a-value, but instead it computes an r-value given a continuation $a \to r$. 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 $c_2\ m$ of the second handler h_2, since $c_2\ m$ 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 $c_2\ m$ 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*

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

In particular, if $r = c_2\ m$, since $alg\ h_2 :: \Sigma_2\ (c_2\ m) \to c_2\ m$, then

$$liftAlgCont\ (alg\ h_2) :: \Sigma_2\ (Cont_{c_2\ m}\ a) \to Cont_{c_2\ m}\ a$$

provides a way to handle operations from Σ_2 using $Cont_{c_2\ m}\ a$.

Theorem 5.2 (Modular Carrier Fusion). *For any modular carriers c_1 and c_2, the data type $c_1\ (Cont_{c_2\ m})$ for any m is also a modular carrier.*

Proof. First we note that there is a natural transformation from m to $Cont_{c_2\ m}$ (which is essentially Filinski [1999]'s CPS-based *monadic reflection*):

$$\begin{aligned} &reflect_{EM} :: (MCarrier\ c_2, Monad\ m) \Rightarrow m\ a \to Cont_{c_2\ m}\ a \\ &reflect_{EM}\ m = Cont\ (\lambda k \to fwd_{c_2}\ (fmap\ k\ m)) \end{aligned}$$

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

$$\begin{aligned} &\mathbf{newtype}\ Fuse\ c_1\ c_2\ m = Fuse\ \{unFuse :: c_1\ (Cont_{c_2\ m})\} \\ &\mathbf{instance}\ (MCarrier\ c_1, MCarrier\ c_2) \Rightarrow MCarrier\ (Fuse\ c_1\ c_2)\ \mathbf{where} \\ &\quad fwd = Fuse \cdot fwd_{c_1} \cdot fmap\ unFuse \cdot reflect_{EM} \end{aligned}$$

The required laws of *fwd* follow from the corresponding laws of c_1 and c_2 (Appendix E.1). □

5.2 Fused Modular Handlers

We intend to use $Fuse\ c_1\ c_2$ 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 $Fuse\ c_1\ c_2 \cong c_1\ (Cont_{c_2\ m})$ and $Cont_{c_2\ m}$ is a monad, $alg\ h_1$ can be used as the Σ_1-algebra for $Fuse\ c_1\ c_2$. Also, the Σ_2-algebra $alg\ h_2 :: \Sigma_2\ (c_2\ m) \to c_2\ m$ can be lifted to $Fuse\ c_1\ c_2$ in the following way:

$$\begin{aligned} &liftAlgF :: (\Sigma_2\ (c_2\ m) \to c_2\ m) \to (\Sigma_2\ (Fuse\ c_1\ c_2\ m) \to Fuse\ c_1\ c_2\ m) \\ &liftAlgF\ alg = Fuse \cdot fwd_{c_1} \cdot liftAlgCont\ alg \cdot fmap\ (return \cdot unFuse) \end{aligned}$$

Theorem 5.3 (Handler Fusion)**.** *For any modular handlers h_1 and h_2, it is the case that handle h_2 · handle h_1 = handle $(h_2 \diamond h_1)$ · 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:*

$$
\begin{aligned}
&(\diamond) :: (\mathit{MCarrier}\ c_1, \mathit{MCarrier}\ c_2, \mathit{Functor}\ \Sigma_1, \mathit{Functor}\ \Sigma_2) \\
&\quad\Rightarrow \mathit{MHandler}\ \Sigma_2\ c_2\ y\ z \rightarrow \mathit{MHandler}\ \Sigma_1\ c_1\ x\ y \rightarrow \mathit{MHandler}\ (\Sigma_1 + \Sigma_2)\ (\mathit{Fuse}\ c_1\ c_2)\ x\ z \\
&h_2 \diamond h_1 = \mathit{MHandler}\ \{\mathit{gen} = \mathit{Fuse} \cdot \mathit{gen}\ h_1, \mathit{alg} = \mathit{alg}_F, \mathit{run} = \mathit{run}_F\}\ \textbf{where} \\
&\quad \mathit{alg}_F\ (\mathit{Inl}\ op) = \mathit{Fuse}\ (\mathit{alg}\ h_1\ (\mathit{fmap}\ \mathit{unFuse}\ op)) \\
&\quad \mathit{alg}_F\ (\mathit{Inr}\ op) = \mathit{liftAlgF}\ (\mathit{alg}\ h_2)\ op \\
&\quad \mathit{run}_F\ x = \mathit{run}\ h_2\ (\mathit{runCont}\ (\mathit{run}\ h_1\ (\mathit{unFuse}\ x))\ (\mathit{gen}\ h_2))
\end{aligned}
$$

Proof sketch. We use the technique by Wu and Schrijvers [2015] to fuse *handle* h_2 · *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 *Fuse* and *unFuse*, 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 · handle h_1 · 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 · *handle* h_1 · *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 · *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 · *handle* h_1, a composite of two *fold*'s. As the first application, we show that *handle* h_2 · *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. The name *modular handlers* is used by Schrijvers et al. [2019] because they allow operations to be modularly handled. The theorem above justifies the name to a greater extent: when two modular handlers are composed together, the equations from both theories are also preserved, which is 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 · *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:

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

Lemma 5.6. *Let h_1 and h_2 be two modular handlers with modular carriers C_1 and C_2 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*

$$\begin{aligned} &\overline{c_1} :: \mathit{Monad}\ m \Rightarrow P_1 \rightarrow (A_1 \rightarrow \mathit{Fuse}\ C_1\ C_2\ m) \rightarrow \mathit{Fuse}\ C_1\ C_2\ m \\ &\overline{c_1}\ p_1\ k = \mathit{Fuse}\ (c_1\ p_1\ (\mathit{unFuse} \cdot k)) \end{aligned}$$

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

$$\begin{aligned} &\overline{c_2} :: \mathit{Monad}\ m \Rightarrow P_2 \rightarrow (A_2 \rightarrow \mathit{Fuse}\ C_1\ C_2\ m) \rightarrow \mathit{Fuse}\ C_1\ C_2\ m \\ &\overline{c_2}\ p_2\ k = \mathit{Fuse}\ (\mathit{fwd}\ (\mathit{Cont}\ (\lambda t \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow t\ (\mathit{unFuse}\ (k\ a_2)))))) \end{aligned} \tag{18}$$

where binder t has type $C_1\ (\mathit{Cont}_{C_2\ m}) \rightarrow C_2\ m$ and fwd is the following instance:

$$\mathit{fwd} :: \mathit{Cont}_{C_2\ m}\ (C_1\ (\mathit{Cont}_{C_2\ m})) \rightarrow C_1\ (\mathit{Cont}_{C_2\ m})$$

This lemma can be calculated from the definition of $\mathit{alg}\ (h_2 \diamond h_1)$ (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 W for some type W, (18) is equal to*

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

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

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

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

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

Remark 5.3. 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 for relevant operations in the 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 *handle* $h_2 \cdot$ *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 :: Theory Σ_1 and T_2 :: Theory Σ_2 and h_1 :: MHandler Σ_1 C_1 X Y and h_2 :: MHandler Σ_2 C_2 Y 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 each $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ 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*

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

for all $p_1 :: P_1$, $p_2 :: P_2$ and $k :: A_1 \to A_2 \to C_1\ (Cont_{C_2\ m})$ for every monad m (or m = Free Empty for closed correctness). In (20), binder t has type $C_1\ (Cont_{C_2\ m}) \to C_2\ m$ and fwd is the instance $fwd :: Cont_{C_2\ m}\ (C_1\ (Cont_{C_2\ m})) \to C_1\ (Cont_{C_2\ m})$.

Proof. It directly follows from 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 of the tensor 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_1 of h_1 is FreeEM W m for some type W, we can simplify (20) with Lemma 5.7. Define c_1' and k' as follows to unwrap the constructors:*

$$\begin{aligned} & c_1' :: P_1 \to (A_1 \to (W \to C_2\ m) \to C_2\ m) \to (W \to C_2\ m) \to C_2\ m \\ & c_1'\ p\ a = runCont\ (unFreeEM\ (c_1\ p\ (FreeEM \cdot Cont \cdot a))) \\ & k' :: A_1 \to A_2 \to (W \to C_2\ m) \to C_2\ m \\ & k'\ a_1\ a_2 = runCont\ (unFreeEM\ (k\ a_1\ a_2)) \end{aligned}$$

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 $W \to C_2\ m$.

Corollary 6.3. *When the modular carrier of h_1 is StateC S W 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:*

$$\begin{aligned} & c_1' :: P_1 \to (A_1 \to S \to (W \to C_2\ m) \to C_2\ m) \to (S \to (W \to C_2\ m) \to C_2\ m) \\ & c_1'\ p\ k = runCont \cdot unStateC\ (c_1\ p\ (\lambda a \to StateC\ (Cont \cdot k\ a))) \\ & k' :: A_1 \to A_2 \to S \to (W \to C_2\ m) \to W \\ & k'\ a_1\ a_2\ s = runCont\ (unStateC\ (k\ a_1\ a_2)\ s) \end{aligned}$$

Then (20) can be simplified to

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

where binder $q :: W \to C_2\ m$.

6.1 Combining Nondeterminism and State

Theorem 6.4. *Handler* $ndetH \diamond stH\ s_0$ *(Example 3.7 and Example 3.8) 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

$$c'_1\ ()\ k = \lambda s \rightarrow k\ s\ s \tag{23}$$

Then we can establish (22) by plugging in c'_1 and simplifying both sides:[1]

$$\begin{aligned}
& c'_1\ p_1\ (\lambda a_1 \rightarrow \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k'\ a_1\ a_2\ s\ q)) && \{\downarrow \text{ definition (23) of } c'_1\} \\
={} & \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k'\ s\ a_2\ s\ q) && \{\uparrow \text{ definition (23) of } c'_1\} \\
={} & \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow c'_1\ p_1\ (\lambda a_1 \rightarrow k'\ a_1\ a_2)\ s\ q)
\end{aligned}$$

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

$$c'_1\ p\ k = \lambda s \rightarrow k\ ()\ p \tag{24}$$

Accordingly, we calculate:

$$\begin{aligned}
& c'_1\ p_1\ (\lambda a_1 \rightarrow \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k'\ a_1\ a_2\ s\ q)) && \{\downarrow \text{ definition (24) of } c'_1\} \\
={} & \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k'\ ()\ a_2\ p_1\ q) && \{\uparrow \text{ definition (24) of } c'_1\} \\
={} & \lambda s\ q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow c'_1\ p_1\ (\lambda a_1 \rightarrow k'\ a_1\ a_2)\ s\ q)
\end{aligned}$$

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].

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

$$c'_1\ ()\ k = \lambda q \rightarrow k\ True\ (\lambda l_1 \rightarrow k\ False\ (\lambda l_2 \rightarrow q\ (l_1 \cup l_2))) \tag{25}$$

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

The left-hand side of (21) becomes

$$\begin{aligned}
& c'_1\ p_1\ (\lambda a_1 \rightarrow \lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k'\ a_1\ a_2\ q)) && \{\text{ definition (26) of } c_2\} \\
={} & c'_1\ p_1\ (\lambda a_1 \rightarrow \lambda q \rightarrow StateC\ (\lambda s \rightarrow unStateC\ (k'\ a_1\ ()\ q)\ p_2)) && \{\text{ definition (25) of } c'_1\} \\
={} & \lambda q \rightarrow StateC\ (\lambda s \rightarrow unStateC\ (k'\ True\ ()\ (\lambda l_1 \rightarrow \\
& \quad \boxed{StateC\ (\lambda s \rightarrow unStateC\ (k'\ False\ ()\ (\lambda l_2 \rightarrow q\ (l_1 \cup l_2)))\ p_2)}\ ))\ p_2)
\end{aligned}$$

and the right-hand side becomes:

$$\lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow c'_1\ p_1\ (\lambda a_1 \rightarrow k'\ a_1\ a_2)\ q) \qquad \{\text{ definition (26) of } c_2\}$$

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

$$= \lambda q \rightarrow StateC\ (\lambda s \rightarrow unStateC\ (c'_1\ p_1\ (\lambda a_1 \rightarrow k'\ a_1\ ())\ q)\ p_2) \quad \{\ \text{definition (25) of } c'_1\ \}$$
$$= \lambda q \rightarrow StateC\ (\lambda s \rightarrow unStateC\ (k'\ True\ ()\ (\lambda l_1 \rightarrow \boxed{k'\ False\ ()\ (\lambda l_2 \rightarrow q\ (l_1 \cup l_2))}\ ))\ p_2)$$

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 $StateC\ (\lambda s \rightarrow RB\ p_2)$ 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*:

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

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*:

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

The writer effect can be handled by the following handler:

```
wtH :: Monoid w ⇒ MHandler (Writer w) (FreeEM (a, w)) a (a, w)
wtH = MHandler gen alg unFreeEM where
  gen a = FreeEM (return (a, mempty))
  alg (Tell w k) = FreeEM (do (a, u) ← unFreeEM (k ()); return (a, w ⋄ u))
```

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* $h_1 :: MHandler\ \Sigma_1\ C_1\ X\ Y$ *and* $h_2 :: MHandler\ \Sigma_2\ C_2\ Y\ Z$, *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 each $O_1 :: P_1 \rightsquigarrow_{\Sigma_1} A_1$ 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

$$\begin{array}{l} c_1\ p_1\ (\lambda a_1 \rightarrow \\ \quad \textbf{if } a_1 \equiv b \textbf{ then } fwd\ (Cont\ (\lambda t \rightarrow \\ \qquad c_2\ p_2\ (\lambda a_2 \rightarrow t\ (k_2\ a_2)))) \\ \qquad \textbf{else } k_1\ a_1) \end{array} = \begin{array}{l} fwd\ (Cont\ (\lambda t \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow \\ \quad t\ (c_1\ p_1\ (\lambda a_1 \rightarrow \\ \qquad \textbf{if } a_1 \equiv b \textbf{ then } k_2\ a_2 \\ \qquad\quad \textbf{else } k_1\ a_1))))) \end{array} \tag{27}$$

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

$$k_1 :: A_1 \rightarrow C_1\ (Cont_{C_2\ m}) \qquad k_2 :: A_2 \rightarrow C_1\ (Cont_{C_2\ m})$$

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 (12) results in (27). □

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

$$\begin{array}{l} c_1'\ p_1\ (\lambda a_1 \rightarrow \\ \quad \textbf{if } a_1 \equiv b \textbf{ then} \\ \qquad \lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow k_2'\ a_2\ q) \\ \qquad \textbf{else } k_1'\ a_1) \end{array} = \begin{array}{l} \lambda q \rightarrow c_2\ p_2\ (\lambda a_2 \rightarrow \\ \quad c_1'\ p_1\ (\lambda a_1 \rightarrow \\ \qquad \textbf{if } a_1 \equiv b \textbf{ then } k_2'\ a_2 \\ \qquad\quad \textbf{else } k_1'\ a_1)\ q) \end{array} \tag{28}$$

where c_1', k_1' and k_2' are the corresponding unprimed function with various data constructors unwrapped:

$$k_1' :: A_1 \rightarrow (W \rightarrow C_2\ m) \rightarrow C_2\ m \qquad k_2' :: A_2 \rightarrow (W \rightarrow C_2\ m) \rightarrow C_2\ m$$

$$c_1' :: P_1 \rightarrow (A_1 \rightarrow (W \rightarrow C_2\ m) \rightarrow C_2\ m) \rightarrow ((W \rightarrow C_2\ m) \rightarrow C_2\ m)$$

We leave out the specialised version for $C_1 = StateC$ for the sake of space.

7.1 Handling Combined Choice

Cheung [2017] shows that models of combined choice *Prob* ▹ *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** $Distr\ a = a \rightarrow Real$ which range in interval $[0, 1]$ and sums to 1 for all elements of a. Two distributions can be convexly combined by $+_\theta :: Distr\ a \rightarrow Distr\ a \rightarrow Distr\ a$ for any $\theta \in [0, 1]$:

$$d_1 +_\theta d_2 = \lambda x \rightarrow \theta * d_1\ x + (1 - \theta) * d_2\ x$$

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

```
probH :: Eq a ⇒ MHandler Prob (FreeEM (Distr a)) a (Distr a)
probH = MHandler gen alg unFreeEM where
  gen x = FreeEM (return (λy → if y ≡ x then 1 else 0))
  alg (PChoose θ k) = FreeEM (
    do d₁ ← unFreeEM (k True); d₂ ← unFreeEM (k False); return (d₁ +_θ d₂))
```

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 $\triangleleft\theta\triangleright$ and $\sqcap$ are

$$\begin{aligned}
&c_1'\ \theta\ k = \lambda q \to k\ True\ (\lambda d_1 \to k\ False\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)))\\
&c_2\ ()\ k = FreeEM\ (\mathbf{do}\ \{l_1 \leftarrow unFreeEM\ (k\ True); l_2 \leftarrow unFreeEM\ (k\ Flase); return\ (l_1 \cup l_2)\})
\end{aligned}$$

The left-hand side of the proof obligation (28) is

$$\begin{aligned}
&\ c_1'\ \theta\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv b\ \mathbf{then}\ \lambda q \to c_2\ ()\ (\lambda a_2 \to k_2'\ a_2\ q)\ \mathbf{else}\ k_1'\ a_1) \quad \{\text{ definition of } c_2\ ()\ \}\\
=&\ c_1'\ \theta\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv b\ \mathbf{then}\ \lambda q \to FreeEM\ (\mathbf{do}\ \{l_1 \leftarrow unFreeEM\ (k_2'\ True\ q);\\
&\quad l_2 \leftarrow unFreeEM\ (k_2'\ False\ q); return\ (l_1 \cup l_2)\})\ \mathbf{else}\ k_1'\ a_1)
\end{aligned}$$

Let us consider the case $b = False$ first, which corresponds to the left distributivity $p \triangleleft\theta\triangleright (q \sqcap r)$. Setting $b = False$ and expanding $c_1'\ \theta$, the last equation becomes

$$\begin{aligned}
&\lambda q \to k_1'\ True\ (\lambda d_1 \to FreeEM\ (\mathbf{do}\ \{l_1 \leftarrow unFreeEM\ (k_2'\ True\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)));\\
&\quad l_2 \leftarrow unFreeEM\ (k_2'\ False\ (\lambda d_2 \to q\ (d_1 +_\theta d_2))); return\ (l_1 \cup l_2)\}))
\end{aligned} \tag{29}$$

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

$$\begin{aligned}
&\ \lambda q \to c_2\ ()\ (\lambda a_2 \to c_1'\ \theta\ (\lambda a_1 \to \mathbf{if}\ a_1 \equiv b\ \mathbf{then}\ k_2'\ a_2\ \mathbf{else}\ k_1'\ a_1)\ q)\\
&\quad \{\text{ definition } c_1'\ \theta\ \}\\
=&\ \lambda q \to c_2\ ()\ (\lambda a_2 \to k_1'\ True\ (\lambda d_1 \to k_2'\ a_2\ (\lambda d_2 \to q\ (d_1 +_\theta d_2))))\\
&\quad \{\text{ definition of } c_2\ ()\ \}\\
=&\ \lambda q \to FreeEM\ (\mathbf{do}\ l_1 \leftarrow unFreeEM\ (k_1'\ True\ (\lambda d_1 \to k_2'\ True\ (\lambda d_2 \to q\ (d_1 +_\theta d_2))))\\
&\qquad\qquad\qquad\quad l_2 \leftarrow unFreeEM\ (k_1'\ True\ (\lambda d_1 \to k_2'\ False\ (\lambda d_2 \to q\ (d_1 +_\theta d_2))))\\
&\qquad\qquad\qquad\quad return\ (l_1 \cup l_2))
\end{aligned} \tag{30}$$

It is not difficult to see (29) ≠ (30) for arbitrary monad m in general, which matches our intuition: under $ndetH \diamond probH$, computation $p \triangleleft\theta\triangleright (q \sqcap r)$ executes p once but $(p \triangleleft\theta\triangleright q) \sqcap (p \triangleleft\theta\triangleright r)$ executes p 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) degenerate to **let**-bindings, and (29) = (30) is equivalent to

$$\begin{aligned}
&\lambda q \to && \lambda q \to \mathbf{let}\ l_1 = k_1'\ True\ (\lambda d_1 \to\\
&\quad k_1'\ True\ (\lambda d_1 \to && \qquad\qquad k_2'\ True\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)))\\
&\quad\quad \mathbf{let}\ l_1 = k_2'\ True\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)) \ = && \qquad\ \ l_2 = k_1'\ True\ (\lambda d_1 \to\\
&\qquad\quad l_2 = k_2'\ False\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)) && \qquad\qquad k_2'\ False\ (\lambda d_2 \to q\ (d_1 +_\theta d_2)))\\
&\quad\quad \mathbf{in}\ l_1 \cup l_2) && \qquad \mathbf{in}\ \ l_1 \cup l_2
\end{aligned} \tag{31}$$

where $k_1', k_2' :: Bool \to (Distr\ A \to Set\ (Distr\ A)) \to Set\ (Distr\ A)$. 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 k_1' and k_2' bear some properties not manifested in their types: they correspond to handled subterms of the computation, and therefore they must be built from $gen\ (ndetH \diamond probH)$ and $alg\ (ndetH \diamond probH)$. Indeed, if $f :: (Distr\ A \to Set\ (Distr\ A)) \to Set\ (Distr\ A)$ is built from $gen\ (ndetH \diamond probH)$ and $alg\ (ndetH \diamond probH)$, then it satisfies

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

and (32) for $f = k_1'\ True$ implies (31).

7.2 Generalising the Continuation Monad

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

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

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 in category theory that there is an adjunctive bijection for any semi-lattices A, B, and set X

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

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

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

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

and monad $X \mapsto C(B^X, B)$ for every object B in C [Hinze 2012]. 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 $C = SL$, 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.

In summary, in this section we have explored showing the correctness of $ndetH \diamond probH$ with respect to the distributive tensor $Prob \triangleright NDet$, and it turns out that our technique (Theorem 7.1) is not powerful enough to do so. As we analysed above, this limitation is caused by the fact that some application-specific invariants are lost in the types, which can be overcome by replacing the continuation monad used in Theorem 5.2 with a generalised form. Although we see no substantial difficulty in doing so, we leave adapting the theorems in this paper to the generalised form and a systematic study of interesting examples needing this generalisation 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] provide two calculi (one for defining handlers and one for using them) and, accordingly, two equational logics extending the logic by 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] present a type system in which computation types are tagged with a set of equations expected to hold, and the type system is parameterised by a reasoning logic that allows the programmer to actually prove that the equations are respected by a handler. Typical choices of the reasoning logic are those in [Plotkin and Pretnar 2008; Pretnar 2010]. In comparison, our paper is more about techniques for reducing the actual proof work of the correctness of composite handlers, and less about formalising languages and logics in which the proof can be done. However, the semi-formal way of working with equations used in our paper aligns well with Lukšič and Pretnar [2020]'s formalisms: computations are interpreted by free monads ignoring the equations, and equations are separately interpreted as a relation. Thus the systems in [Lukšič and Pretnar 2020] suits well for formalising our results, which is an important piece of future work.

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 priori*. 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 a language with effect polymorphism based on *tunneling* in which the parametricity theorem holds for effect-polymorphic functions. In comparison, we have focused exclusively on denotational semantics of effect handlers (presented in Haskell), and we achieve effect polymorphism by being polymorphic in the signature functors, utilising the polymorphic mechanisms of Haskell. Since our technique of handler fusion crucially relies on parametricity, we expect our results to hold only in an operational semantics admitting the parametricity theorem, such as Zhang and Myers [2019]'s and Brachthäuser et al. [2020]'s. Otherwise, a handler may accidentally intercept operations supposed to be handled by other handlers, breaking the modularity of handlers.

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.

Hillerström and Lindley [2018] introduce *shallow handlers* that semantically corresponds to case-splits on syntax trees of computations, whereas traditional deep handlers correspond to catamorphisms. When used with general recursion, shallow handlers can conveniently implement handling schemes that do not fit in the structure of deep handlers, such as handling two mutually dependent programs. However, this is at the cost of relinquishing structural recursion inherent in deep handlers that offers the programmer more reasoning principles. In particular, we have shown how the fusion property can be used to reason about composites of deep handlers.

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 those in [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 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 by Seynaeve et al. [2020] to eliminate intermediate lists when implementing nondeterminism with mutable state.

Proof Assistants for Equational Reasoning. The results of this paper allow the programmer to prove the correctness of composite handlers by doing equational reasoning about the clauses of handlers. However, all the proofs, both of our theorems and the use cases of the theorems, in this paper are done in a paper-and-pencil style. Thus we expect that the proposed technique would be more usable for programmers if there are mechanised tools for equational reasoning about programs when applying the proposed technique. This can be possibly achieved by resorting to existing proof assistants supporting equational reasoning about Haskell programs. Among them, LiquidHaskell [Vazou et al. 2018, 2017] seems promising, since it utilises SMT solvers to automatically verify the equality of programs. Another option is converting Haskell programs to Coq using the tools by Breitner et al. [2018]; Spector-Zabusky et al. [2018] and doing equational reasoning therein (with libraries like [Tesson et al. 2011] that aid with equational reasoning). Beyond formalising equational reasoning about handlers, a complete mechanised formalisation of the theorems in the paper is also an interesting piece of future work.

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 that can be determined at compile time and inline them into computations.
- We have only considered algebraic operations and we wish to extend this techniques in the paper to a broader family such as scoped operations [Piróg et al. 2018].

Effect handlers have proven to be a powerful construct for modelling language features modularly, but a powerful construct is only useful when powerful reasoning techniques are available. We hope that this paper can inspire more reasoning techniques for handlers to be developed in the future.

ACKNOWLEDGMENTS

This work has been supported by EPSRC grant number EP/S028129/1 on 'Scoped Contextual Operations and Effects'. The authors would like to thank Josh Ko, Marco Paviotti, Tom Schrijvers, Shin-Cheng Mu and the anonymous reviewers for their constructive feedback.

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:1807.05923 [cs.LO] 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* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://doi.org/10.1145/3236784

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* (San Antonio, Texas, USA) *(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* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. *Lecture Notes in Computer Science* 11275 LNCS (2018), 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

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. 1985a. *Communicating Sequential Processes.* Prentice-Hall, Inc., USA.

C. A. R. Hoare. 1985b. 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* (Boston, Massachusetts, USA) *(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

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

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

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and semantics for operations with scopes. *Proceedings - Symposium on Logic in Computer Science* 1 (2018), 809–818. https://doi.org/10.1145/3209108.3209166

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

Matija Pretnar. 2010. *Logic and handling of algebraic effects.* Ph.D. Dissertation. University of Edinburgh, UK.

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* (Berlin, Germany) *(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. https://doi.org/10.1007/978-3-030-57761-2_10

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. 2011. Program Calculation in Coq. In *Algebraic Methodology and Software Technology*, Michael Johnson and Dusko Pavlovic (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 163–179. https://doi.org/10.1007/978-3-642-17796-5_10

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 132–144. https://doi.org/10.1145/3242744.3242756

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

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. https://doi.org/10.1007/978-3-540-70594-9_20

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