Handling Fibred Algebraic Effects

DANEL AHMAN, Inria Paris, France

We study algebraic computational effects and their handlers in the dependently typed setting. We describe computational effects using a generalisation of Plotkin and Pretnar's effect theories, whose dependently typed operations allow us to capture precise notions of computation, e.g., state with location-dependent store types and dependently typed update monads. Our treatment of handlers is based on an observation that their conventional term-level definition leads to unsound program equivalences being derivable in languages that include a notion of homomorphism. We solve this problem by giving handlers a novel type-based treatment via a new computation type, the user-defined algebra type, which pairs a value type (the carrier) with a set of value terms (the operations), capturing Plotkin and Pretnar's insight that effect handlers denote algebras. We then show that the conventional presentation of handlers can be routinely derived, and demonstrate that this type-based treatment of handlers provides a useful mechanism for reasoning about effectful computations. We also equip the resulting language with a sound denotational semantics based on families fibrations.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Type theory**; **Control primitives**; **Type structures**; **Program specifications**; **Denotational semantics**;

Additional Key Words and Phrases: Dependent Types, Algebraic Effects, Handlers of Algebraic Effects

ACM Reference Format:
Danel Ahman. 2018. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (January 2018), 29 pages. https://doi.org/10.1145/3158095

1 INTRODUCTION

An important feature of many widely-used programming languages is their support for computational effects (e.g., raising exceptions, accessing memory, performing I/O), which allows programmers to write more efficient and conceptually clearer programs. Therefore, if dependently typed languages are to live up to their promise of providing a lightweight means for integrating formal verification and practical programming, we must first understand how to properly account for computational effects in such languages. While there already exists a range of work on combining these two fields (e.g., Ahman et al. [2016, 2017]; Brady [2013]; Casinghino [2014]; Hancock and Setzer [2000]; McBride [2011]; Nanevski et al. [2008]; Pédrot and Tabareau [2017]; Pitts et al. [2015]), there is still a gap between the rigorous and comprehensive understanding we have of computational effects in the simply typed setting, and what we know about them in the presence of dependent types. For example, in the mentioned works, either the mathematical foundations of the languages developed are not settled, the available effects are limited, or they lack a systematic treatment of (equational) effect specification. In this paper we contribute to the intersection of these two fields by investigating how to combine dependent types with algebraic effects and their handlers.

Algebraic effects form a wide class of computational effects that lend themselves to specification using operations and equations; examples include exceptions, state, input-output, nondeterminism, probability, etc. Their study originated with the pioneering work of Plotkin and Power [2001, 2002];

Author's address: Danel Ahman, Prosecco Team, Inria Paris, 2 rue Simone Iff, Paris, 75012, France, danel.ahman@inria.fr.

2475-1421/2018/1-ART7
https://doi.org/10.1145/3158095

and they have since been successfully applied to, e.g., modularly combining different effects [Hyland et al. 2006] and effect-dependent program optimisations [Kammar and Plotkin 2012]. A key insight of Plotkin and Power was that most of Moggi's monads [Moggi 1989, 1991] are determined by algebraic presentations, with the notable exception of continuations, which are not algebraic.

A major role in the recent rise of interest in algebraic effects can be attributed to their *handlers*. These were introduced by Plotkin and Pretnar [2013] as a generalisation of exception handlers to all algebraic effects, based on the insight that handlers denote user-defined algebras for the given notion of computation, and the handling construct denotes the homomorphism induced by the universal property of the free algebra. From a programming language perspective, an effect handler

$$\{\mathsf{op}_x(x') \mapsto N_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{eff}}}$$

provides redefinitions of the algebraic operations in the signature $\mathcal{S}_{\mathrm{eff}}$, and the handling construct

$$M \;\mathtt{handled\ with}\; \{\mathsf{op}_x(x') \mapsto N_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{eff}}} \;\mathtt{to}\; y{:}A \;\mathtt{in}\; N_{\mathrm{ret}}$$

then recursively traverses the given program M, replacing each algebraic operation op with the corresponding user-defined computation term N_{op}, e.g., as illustrated by the following β-equation:

$$\begin{aligned}&\Gamma \vdash (\mathsf{op}_V^{FA}(y'.M))\;\mathtt{handled\ with}\;\{\mathsf{op}_x(x') \mapsto N_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{eff}}}\;\mathtt{to}\;y{:}A\;\mathtt{in}\;N_{\mathrm{ret}}\\ &\quad = N_{\mathsf{op}}[V/x, \lambda y'{:}O[V/x].\mathtt{thunk}\;H/x'] : \underline{C}\end{aligned}$$

where, for better readability, we abbreviate the recursive call to the handling construct as

$$H \stackrel{\mathrm{def}}{=} M \;\mathtt{handled\ with}\; \{\mathsf{op}_x(x') \mapsto N_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{eff}}} \;\mathtt{to}\; y{:}A \;\mathtt{in}\; N_{\mathrm{ret}}$$

Plotkin and Pretnar [2013] also showed that handlers can be used to neatly implement timeouts, rollbacks, stream redirection, etc. More recently, handlers have also gained popularity as a practical and modular programming language abstraction, allowing one to write programs generically in terms of algebraic operations, and then use handlers to modularly provide different fit-for-purpose implementations for these programs. A prototypical example of this approach involves implementing the state operations (`get` and `put`) using their natural representation as state-passing functions $\mathsf{St} \to A \times \mathsf{St}$. In order to support this style of programming, existing languages have been extended with algebraic effects and their handlers [Hillerström and Lindley 2016; Kammar et al. 2013; Leijen 2017], and new languages have been built around them [Bauer and Pretnar 2015; Lindley et al. 2017]. However, being simply typed, these languages do not allow the programmer to (equationally) specify the intended behaviour of the computational effects at hand, and thus provide no guarantees that the defined handlers satisfy these specifications, a gap we aim to fill.

The main *contributions* of this paper are: i) a dependently typed generalisation of Plotkin and Pretnar's effect theories (§3.1); ii) an observation that the conventional term-level definition of effect handlers leads to unsound program equivalences being derivable in languages that include a notion of homomorphism (§4.1); iii) a new computation type, the user-defined algebra type, giving a type-based treatment of handlers and solving the above-mentioned problem with unsound program equivalences (§4.2); iv) a derivation of the conventional term-level definition of handlers from our type-based treatment (§4.3); v) a demonstration that such handlers provide a useful mechanism for reasoning about effectful computations (§7); and vi) a natural denotational semantics for the resulting language based on families fibrations and models of countable Lawvere theories (§8).

This paper is based on §6 and §7 of the author's PhD thesis [Ahman 2017] in which variants of these contributions are presented in a system where the equational proof obligations used in the type-based treatment of handlers are given by definitional equations instead of propositional ones.

2 EMLTT: THE UNDERLYING EFFECTFUL DEPENDENTLY TYPED LANGUAGE

We begin with an overview of the language we use as a basis for studying algebraic effects and their handlers in the dependently typed setting, namely, the effectful dependently typed language proposed by Ahman et al. [2016]. This language is a natural extension of Martin-Löf's [1975] intensional type theory (MLTT) with computational effects. It makes a clear distinction between values and computations, both at the level of types and terms, analogously to simply typed languages such as Call-By-Push-Value (CBPV) [Levy 2004] and the Enriched Effect Calculus (EEC) [Egger et al. 2014]. Specifically, we base our work in this paper on a minor extension of Ahman et al.'s language, as explained below. Following Ahman [2017], we refer to this extended language as EMLTT.

As usual for dependently typed languages, EMLTT's types and terms are defined mutually inductively. First, one assumes countable sets of *value variables* $x, y, \ldots$ and *computation variables* $z, \ldots$. Next, the grammar of *value types* $A, B, \ldots$ and *computation types* $\underline{C}, \underline{D}, \ldots$ is given by

$$A ::= \mathsf{Nat} \mid 1 \mid 0 \mid A + B \mid \Sigma x{:}A.B \mid \Pi x{:}A.B \mid V =_A W \mid U\underline{C} \mid \underline{C} \multimap \underline{D}$$
$$\underline{C} ::= FA \mid \Sigma x{:}A.\underline{C} \mid \Pi x{:}A.\underline{C}$$

Analogously to Ahman et al. [2016], we omit general inductive types and use natural numbers as a representative example. Compared to op. cit., our value types also include the empty type 0, the sum type $A + B$, and the homomorphic function type $\underline{C} \multimap \underline{D}$. We include the first two as to specify signatures of algebraic effects (see §3.2); and the latter as it is useful for writing effectful code without excessive thunking and forcing, and because it enables us to eliminate values into homomorphism terms, as discussed later in this section. Further, as standard, we write $A \times B$ and $A \to B$ for $\Sigma x{:}A.B$ and $\Pi x{:}A.B$ when the value variable x does not appear free in B. Finally, we note that FA is the type of possibly effectful computations that return values of type A.

Next, the grammar of EMLTT's *value terms* $V, W, \ldots$ is given by

$$\begin{aligned} V ::= {} & x \mid \star \mid \mathtt{zero} \mid \mathtt{succ}\ V \mid \mathtt{nat\text{-}elim}_{x.A}(V_z, y_1.y_2.V_s, V) \mid \mathtt{case}\ V\ \mathtt{of}_{x.A}\ () \\ \mid {} & \mathtt{inl}_{A+B}\ V \mid \mathtt{inr}_{A+B}\ V \mid \mathtt{case}\ V\ \mathtt{of}_{x.B}\ (\mathtt{inl}(y_1{:}A_1) \mapsto W_1, \mathtt{inr}(y_2{:}A_2) \mapsto W_2) \\ \mid {} & \langle V, W\rangle_{(x:A).B} \mid \mathtt{pm}\ V\ \mathtt{as}\ (x_1{:}A_1, x_2{:}A_2)\ \mathtt{in}_{y.B}\ W \mid \lambda x{:}A.V \mid V(W)_{(x:A).B} \mid \lambda z{:}\underline{C}.K \\ \mid {} & \mathtt{refl}\ V \mid \mathtt{eq\text{-}elim}_A(x_1.x_2.x_3.B, y.W, V_1, V_2, V_p) \mid \mathtt{fun\text{-}ext}_{(x:A).B}(V_1, V_2, W_p) \mid \mathtt{thunk}\ M \end{aligned}$$

Observe that in addition to the introduction and elimination forms for the types inherited from MLTT, value terms also include *thunks* of computations and *homomorphic lambda abstractions*. Compared to Ahman et al. [2016] and Ahman [2017], in this paper we further include the axiom of *function extensionality*, so as to enable reasoning about proof obligations given by propositional equalities between functions, e.g., as used in §4.2 for our type-based treatment of effect handlers.

Regarding effectful programs, EMLTT makes a distinction between *computation terms* $M, N, \ldots$ and *homomorphism terms* $K, L, \ldots$. The grammar of these two kinds of terms is given by

$$\begin{aligned} M ::= {} & \mathtt{force}_{\underline{C}}\ V \mid \mathtt{return}\ V \mid M\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{C}}\ N \\ \mid {} & \langle V, M\rangle_{(x:A).\underline{C}} \mid M\ \mathtt{to}\ (x{:}A, z{:}\underline{C})\ \mathtt{in}_{\underline{D}}\ K \\ \mid {} & \lambda x{:}A.M \mid M(V)_{(x:A).\underline{C}} \mid V(M)_{\underline{C},\underline{D}} \end{aligned} \qquad \begin{aligned} K ::= {} & z \mid K\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{C}}\ M \\ \mid {} & \langle V, K\rangle_{(x:A).\underline{C}} \mid K\ \mathtt{to}\ (x{:}A, z{:}\underline{C})\ \mathtt{in}_{\underline{D}}\ L \\ \mid {} & \lambda x{:}A.K \mid K(V)_{(x:A).\underline{C}} \mid V(K)_{\underline{C},\underline{D}} \end{aligned}$$

Computation terms include standard term formers: returning a value, sequential composition, lambda abstraction, and function application. They also include forcing of thunks, introduction and elimination forms for the computational Σ-type, and homomorphic function applications. Homomorphism terms differ from computation terms in two respects: on the one hand, they do not include $\mathtt{force}_{\underline{C}}\ V$ and $\mathtt{return}\ V$; on the other hand, they include computation variables z, which have to be used i) linearly and ii) in a way that ensures that the computation bound to z "happens

Value types:

$$\frac{}{\vdash \Gamma}\quad \frac{\Gamma \vdash \mathsf{Nat}}{}\quad \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : A}{\Gamma \vdash V =_A W}\quad \frac{\Gamma \vdash \underline{C}}{\Gamma \vdash U\underline{C}}\quad \frac{\Gamma \vdash \underline{C} \quad \Gamma \vdash \underline{D}}{\Gamma \vdash \underline{C} \multimap \underline{D}}$$

Computation types:

$$\frac{\Gamma \vdash A}{\Gamma \vdash FA}\quad \frac{\Gamma, x{:}A \vdash \underline{C}}{\Gamma \vdash \Sigma x{:}A.\underline{C}}\quad \frac{\Gamma, x{:}A \vdash \underline{C}}{\Gamma \vdash \Pi x{:}A.\underline{C}}$$

Value terms:

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{refl}\ V : V =_A V}\quad \frac{\begin{array}{c}\Gamma, x_1{:}A, x_2{:}A, x_3{:}x_1 =_A x_2 \vdash B \quad \Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : A \\ \Gamma \vdash V_p : V_1 =_A V_2 \quad \Gamma, y{:}A \vdash W : B[y/x_1, y/x_2, \mathtt{refl}\ y/x_3]\end{array}}{\Gamma \vdash \mathtt{eq\text{-}elim}_A(x_1.x_2.x_3.B, y.W, V_1, V_2, V_p) : B[V_1/x_1, V_2/x_2, V_p/x_3]}\quad \frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash \mathtt{thunk}\ M : U\underline{C}}$$

$$\frac{\Gamma \vdash V_1 : \Pi x{:}A.B \quad \Gamma \vdash V_2 : \Pi x{:}A.B \quad \Gamma \vdash W_p : \Pi x{:}A.(V_1(x)_{(x:A).B} =_B V_2(x)_{(x:A).B})}{\Gamma \vdash \mathtt{fun\text{-}ext}_{(x:A).B}(V_1, V_2, W_p) : V_1 =_{\Pi x:A.B} V_2}\quad \frac{\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash \lambda z{:}\underline{C}.K : \underline{C} \multimap \underline{D}}$$

Computation terms:

$$\frac{\Gamma \vdash V : U\underline{C}}{\Gamma \vdash \mathtt{force}_{\underline{C}}\ V : \underline{C}}\quad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{return}\ V : FA}\quad \frac{\Gamma \vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, x{:}A \vdash N : \underline{C}}{\Gamma \vdash M\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{C}}\ N : \underline{C}}\quad \frac{\Gamma \vdash V : \underline{C} \multimap \underline{D} \quad \Gamma \vdash M : \underline{C}}{\Gamma \vdash V(M)_{\underline{C},\underline{D}} : \underline{D}}$$

Homomorphism terms:

$$\frac{\Gamma \vdash \underline{C}}{\Gamma \mid z{:}\underline{C} \vdash z : \underline{C}}\quad \frac{\Gamma \mid z{:}\underline{C} \vdash K : FA \quad \Gamma \vdash \underline{D} \quad \Gamma, x{:}A \vdash M : \underline{D}}{\Gamma \mid z{:}\underline{C} \vdash K\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{D}}\ M : \underline{D}}\quad \frac{\Gamma \vdash V : A \quad \Gamma, x{:}A \vdash \underline{D} \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}[V/x]}{\Gamma \mid z{:}\underline{C} \vdash \langle V, K\rangle_{(x:A).\underline{D}} : \Sigma x{:}A.\underline{D}}$$

$$\frac{\Gamma \mid z_1{:}\underline{C} \vdash K : \Sigma x{:}A.\underline{D}_1 \quad \Gamma \vdash \underline{D}_2 \quad \Gamma, x{:}A \mid z_2{:}\underline{D}_1 \vdash L : \underline{D}_2}{\Gamma \mid z_1{:}\underline{C} \vdash K\ \mathtt{to}\ (x{:}A, z_2{:}\underline{D}_1)\ \mathtt{in}_{\underline{D}_2}\ L : \underline{D}_2}\quad \frac{\Gamma \vdash \underline{C} \quad \Gamma, x{:}A \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \mid z{:}\underline{C} \vdash \lambda x{:}A.K : \Pi x{:}A.\underline{D}}$$

$$\frac{\Gamma, x{:}A \vdash \underline{D} \quad \Gamma \mid z{:}\underline{C} \vdash K : \Pi x{:}A.\underline{D} \quad \Gamma \vdash V : A}{\Gamma \mid z{:}\underline{C} \vdash K(V)_{(x:A).\underline{D}} : \underline{D}[V/x]}\quad \frac{\Gamma \vdash V : \underline{D}_1 \multimap \underline{D}_2 \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}_1}{\Gamma \mid z{:}\underline{C} \vdash V(K)_{\underline{D}_1,\underline{D}_2} : \underline{D}_2}$$

Fig. 1. Selected formation and typing rules for EMLTT's types and terms.

first" in a term containing it. Omitting the two term formers and using computation variables in this way guarantees that every K denotes a homomorphism in the models we consider in §8.

As such, homomorphism terms were crucial for Ahman et al. [2016] in enabling the elimination form for the computational Σ-type $\Sigma x{:}A.\underline{C}$ to be defined correctly. Namely, a term of this type is eliminated into a pair of variables, one denoting a value and the other a computation. To preserve the intended left-to-right evaluation order of one's program, it is crucial that the variable denoting a computation is evaluated first, and not discarded or duplicated arbitrarily, which is guaranteed by how homomorphism terms are defined and how computation variables are used in them.

The *well-formed syntax* of EMLTT is defined using judgements (see Fig. 1) of well-formed value contexts $\vdash \Gamma$, value types $\Gamma \vdash A$, and computation types $\Gamma \vdash \underline{C}$; and well-typed value terms $\Gamma \vdash V : A$, computation terms $\Gamma \vdash M : \underline{C}$, and homomorphism terms $\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}$. *Contexts* Γ are lists of distinct value variables, each annotated with a value type; and with the *empty context* written as $\diamond$.

As one can readily use thunking and forcing (and homomorphic functions) to eliminate values into computation terms (resp. homomorphism terms), these elimination forms are not included primitively. For example, one can eliminate natural numbers into computation terms as follows:

$$\mathtt{nat\text{-}elim}_{x.\underline{C}}(M_z, y_1.y_2.M_s, V) \stackrel{\text{def}}{=} \mathtt{force}_{\underline{C}[V/x]}\left(\mathtt{nat\text{-}elim}_{x.U\underline{C}}(\mathtt{thunk}\ M_z, y_1.y_2.\mathtt{thunk}\ M_s, V)\right)$$

and (non-dependently) into homomorphism terms as follows:

$$\mathtt{nat\text{-}elim}_{\underline{C}}(K_z, y.K_s, V) \stackrel{\text{def}}{=} \left(\mathtt{nat\text{-}elim}_{x_1.\underline{C} \multimap \underline{C}}(\lambda z{:}\underline{C}.K_z, y.x_2.\lambda z{:}\underline{C}.K_s[x_2\ z/z], V)\right)\ z$$

Value terms:

$$\frac{\Gamma \vdash A \quad \Gamma, x_1{:}A, x_2{:}A, x_3{:}x_1 =_A x_2 \vdash B \quad \Gamma \vdash V : A \quad \Gamma, y{:}A \vdash W : B[y/x_1, y/x_2, \mathtt{refl}\ y/x_3]}{\Gamma \vdash \mathtt{eq\text{-}elim}_A(x_1.x_2.x_3.B, y.W, V, V, \mathtt{refl}\ V) = W[V/y] : B[V/x_1, V/x_2, \mathtt{refl}\ V/x_3]}$$

$$\frac{\Gamma \vdash V : U\underline{C}}{\Gamma \vdash \mathtt{thunk}\ (\mathtt{force}_{\underline{C}}\ V) = V : U\underline{C}} \qquad \frac{\Gamma \vdash V : \underline{C} \multimap \underline{D}}{\Gamma \vdash V = \lambda z{:}\underline{C}.V(z)_{\underline{C},\underline{D}} : \underline{C} \multimap \underline{D}}$$

Computation terms:

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash \underline{C} \quad \Gamma, x{:}A \vdash M : \underline{C}}{\Gamma \vdash (\mathtt{return}\ V)\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{C}}\ M = M[V/x] : \underline{C}} \qquad \frac{\Gamma \vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma \mid z{:}FA \vdash K : \underline{C}}{\Gamma \vdash M\ \mathtt{to}\ x{:}A\ \mathtt{in}_{\underline{C}}\ K[\mathtt{return}\ x/z] = K[M/z] : \underline{C}}$$

$$\frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash \mathtt{force}_{\underline{C}}\ (\mathtt{thunk}\ M) = M : \underline{C}} \qquad \frac{\Gamma \vdash M : \underline{C} \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash (\lambda z{:}\underline{C}.K)(M)_{\underline{C},\underline{D}} = K[M/z] : \underline{D}}$$

Homomorphism terms:

$$\frac{\Gamma \vdash V : A \quad \Gamma \mid z_1{:}\underline{C} \vdash K : \underline{D}_1[V/x] \quad \Gamma \vdash \underline{D}_2 \quad \Gamma, x{:}A \mid z_2{:}\underline{D}_1 \vdash L : \underline{D}_2}{\Gamma \mid z_1{:}\underline{C} \vdash \langle V, K\rangle_{(x:A).\underline{D}_1}\ \mathtt{to}\ (x{:}A, z_2{:}\underline{D}_1)\ \mathtt{in}_{\underline{D}_2}\ L = L[V/x][K/z_2] : \underline{D}_2}$$

$$\frac{\Gamma, x{:}A \vdash \underline{D}_1 \quad \Gamma \mid z_1{:}\underline{C} \vdash K : \Sigma x{:}A.\underline{D}_1 \quad \Gamma \vdash \underline{D}_2 \quad \Gamma \mid z_3{:}\Sigma x{:}A.\underline{D}_1 \vdash K : \underline{D}_2}{\Gamma \mid z_1{:}\underline{C} \vdash K\ \mathtt{to}\ (x{:}A, z_2{:}\underline{D}_1)\ \mathtt{in}_{\underline{D}_2}\ L[\langle x, z_2\rangle_{(x:A).\underline{D}_1}/z_3] = L[K/z_3] : \underline{D}_2}$$

$$\frac{\Gamma \vdash \underline{C} \quad \Gamma, x{:}A \mid z{:}\underline{C} \vdash K : \underline{D} \quad \Gamma \vdash V : A}{\Gamma \mid z{:}\underline{C} \vdash (\lambda x{:}A.K)(V)_{(x:A).\underline{D}} = K[V/x] : \underline{D}[V/x]} \qquad \frac{\Gamma, x{:}A \vdash \underline{D} \quad \Gamma \mid z{:}\underline{C} \vdash K : \Pi x{:}A.\underline{D}}{\Gamma \mid z{:}\underline{C} \vdash K = \lambda x{:}A.K(x)_{(x:A).\underline{D}} : \Pi x{:}A.\underline{D}}$$

Fig. 2. Selected definitional equations from EMLTT's equational theory.

where we assume that $FCV(K_z) = FCV(K_s) = z$, and where x_1 and x_2 are chosen fresh.[1]

Analogously to Ahman et al. [2016], we decorate value, computation, and homomorphism terms with a number of type annotations. We use these annotations to define the denotational semantics of EMLTT on raw expressions, so as to avoid well-known coherence problems arising in the interpretation of dependently typed languages; this is a standard technique in the literature [Hofmann 1997; Streicher 1991]. For better readability, we often omit these type annotations in examples.

The well-formed syntax of EMLTT is defined mutually inductively with its *equational theory* (see Fig. 2), consisting of a collection of mutually defined equivalence relations given by *definitional equations* between well-formed value contexts, written $\vdash \Gamma_1 = \Gamma_2$; well-formed types, written $\Gamma \vdash A = B$ and $\Gamma \vdash \underline{C} = \underline{D}$; and well-typed terms, written $\Gamma \vdash V = W : A$, $\Gamma \vdash M = N : \underline{C}$, and $\Gamma \mid z{:}\underline{C} \vdash K = L : \underline{D}$. These equations interact with well-formed syntax via conversion rules, such as

$$\frac{\vdash \Gamma_1 = \Gamma_2 \quad \Gamma_1 \vdash V : A_1 \quad \Gamma_1 \vdash A_1 = A_2}{\Gamma_2 \vdash V : A_2} \qquad \frac{\vdash \Gamma_1 = \Gamma_2 \quad \Gamma_1 \vdash M : \underline{C}_1 \quad \Gamma_1 \vdash \underline{C}_1 = \underline{C}_2}{\Gamma_2 \vdash M : \underline{C}_2}$$

Note that as EMLTT is based on Martin-Löf's intensional type theory, the elimination form for propositional equality $V =_A W$ supports a β-equation but not an η-equation (see Fig. 2). Similarly, the elimination form for natural numbers also only supports β-equations. In both cases, this is done so as to avoid known sources of undecidability for typechecking and the equational theory—for more details, see the analysis by Hofmann [1995], and by Okada and Scott [1999], respectively.

Regarding the meta-theory of EMLTT, one can readily prove standard weakening and substitution results, the latter for both value and computation variables. For example, we write $A[V/x]$ for the substitution of V for x in A. Analogously, we write $K[M/z]$ for the substitution of M for z in K.

[1] $FCV(K)$ denotes the free computation variable of the homomorphism term K.

The definitions of both kinds of substitution are straightforward: they proceed by recursion on the structure of the given type or term, making use of the standard convention of identifying types and terms that differ only in the names of bound variables, and assuming that in any definition, etc., the bound variables of types and terms are chosen to be different from any free variables.

We conclude by recalling from Ahman et al. [2016] that one of the notable features of EMLTT is the computational Σ-type $\Sigma x\!:\!A.\underline{C}$. This computation type provides a uniform means to account for type-dependency in sequential composition, allowing one to "close-off" the type of the the second computation with $\Sigma x\!:\!A.\underline{C}$ before using the typing rule for sequential composition that prohibits x to appear free in the type of the second computation. A similar restriction on free variables also appears in other typing rules for effectful programs. As a consequence, EMLTT lends itself to a very natural general denotational semantics based on *fibred adjunctions*, as studied in detail in Ahman [2017]; Ahman et al. [2016]. Thus, one says that the computational effects in EMLTT are *fibred*.

3 FIBRED ALGEBRAIC EFFECTS

In this section we develop a formal means for specifying computational effects in EMLTT using operations and equations, based on a natural dependently typed generalisation of the effect theories of Plotkin and Pretnar [2013]. We note that while algebraic effects were already discussed by the author in the context of EMLTT in Ahman et al. [2016], they were treated much more informally compared to this paper, e.g., without making precise any particular notion of effect theory.

3.1 Fibred Effect Theories

We begin by identifying the fragment of EMLTT which we use to define the types of our operations.

A value type is *pure* if it is built up from only Nat, 1, $\Sigma x\!:\!A.B$, $\Pi x\!:\!A.B$, 0, $A + B$, and $V =_A W$, where V, W, and A are all pure in propositional equality $V =_A W$. A value term is *pure* if it does not contain thunks of computations and homomorphic lambda abstractions, and all its type annotations are pure. This notion of pureness extends straightforwardly to contexts—a value context Γ is *pure* if A_i is pure for every $x_i\!:\!A_i \in \Gamma$. Note that this fragment of EMLTT corresponds precisely to MLTT.

Assuming a countable set of *effect variables* $w, \ldots$, we now define our notion of fibred effect theory. We begin by defining corresponding signatures of operation symbols and then add equations between derivable effect terms, so as to specify both the effects at hand and their behaviour.

A *fibred effect signature* $\mathcal{S}_{\text{eff}}$ consists of a finite set of typed operation symbols $\text{op} : (x\!:\!I) \longrightarrow O$, where $\diamond \vdash I$ and $x\!:\!I \vdash O$ are required to be pure value types, called the *input* and *output* type of op.

The *effect terms* T that one can derive from the given fibred effect signature $\mathcal{S}_{\text{eff}}$ are given by

$$\begin{aligned} T \;::=\;& w\,(V) \mid \text{op}_V(y.T) \mid \text{pm } V \text{ as } (x_1\!:\!A_1, x_2\!:\!A_2) \text{ in } T \\ \mid\;& \text{case } V \text{ of } (\text{inl}(x_1\!:\!A_1) \mapsto T_1, \text{inr}(x_2\!:\!A_2) \mapsto T_2) \end{aligned}$$

with the involved value types and value terms all required to be pure. We follow the convention of omitting V in $\text{op}_V(y.T)$ when the input type of op is 1, and y when the output type of op is 1. In future, if one were to discover computationally interesting examples needing the elimination forms of other value types (e.g., Nat), then these can be accommodated in effect terms straightforwardly.

An *effect context* Δ is a list of distinct effect variables annotated with pure value types. We say that Δ is *well-formed* in a pure value context Γ, written $\Gamma \vdash \Delta$, if $\vdash \Gamma$ and $\Gamma \vdash A_i$ for every $w_j\!:\!A_j \in \Delta$. Intuitively, each effect variable $w\!:\!A$ denotes a continuation that expects a value of type A.

Well-formed effect terms are then defined using the judgement $\Gamma \mid \Delta \vdash T$ as follows:

$$\frac{\Gamma \vdash \Delta_1, w\!:\!A, \Delta_2 \quad \Gamma \vdash V : A}{\Gamma \mid \Delta_1, w\!:\!A, \Delta_2 \vdash w\,(V)} \qquad \frac{\Gamma \vdash V : I \quad \Gamma \vdash \Delta \quad \Gamma, y\!:\!O[V/x] \mid \Delta \vdash T}{\Gamma \mid \Delta \vdash \text{op}_V(y.T)}\ (\text{op} : (x\!:\!I) \longrightarrow O \in \mathcal{S}_{\text{eff}})$$

$$\frac{\Gamma \vdash V : \Sigma x_1{:}A_1.A_2 \qquad \Gamma \vdash \Delta \quad \Gamma, x_1{:}A_1, x_2{:}A_2 \mid \Delta \vdash T}{\Gamma \mid \Delta \vdash \mathsf{pm}\ V\ \mathsf{as}\ (x_1{:}A_1, x_2{:}A_2)\ \mathsf{in}\ T} \qquad \frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma \vdash \Delta \quad \Gamma, x_1{:}A_1 \mid \Delta \vdash T_1 \quad \Gamma, x_2{:}A_2 \mid \Delta \vdash T_2}{\Gamma \mid \Delta \vdash \mathsf{case}\ V\ \mathsf{of}\ (\mathsf{inl}(x_1{:}A_1) \mapsto T_1, \mathsf{inr}(x_2{:}A_2) \mapsto T_2)}$$

Finally, a *fibred effect theory* $\mathcal{T}_{\text{eff}} = (\mathcal{S}_{\text{eff}}, \mathcal{E}_{\text{eff}})$ is given by a fibred effect signature $\mathcal{S}_{\text{eff}}$ and a finite set $\mathcal{E}_{\text{eff}}$ of equations $\Gamma \mid \Delta \vdash T_1 = T_2$ between well-formed effect terms $\Gamma \mid \Delta \vdash T_1$ and $\Gamma \mid \Delta \vdash T_2$.

In order to simplify the presentation of typing rules involving fibred effect theories, we assume $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ and $\Delta = w_1{:}A'_1, \ldots, w_m{:}A'_m$ when quantifying over the variables of Γ, Δ.

3.2 Examples of Fibred Effect Theories

As our fibred effect theories are a natural dependently typed generalisation of Plotkin and Pretnar's effect theories, we can capture all the effects they can, e.g., assuming a pure value type $\diamond \vdash \mathsf{Exc}$ of exception names, the *theory* $\mathcal{T}_{EXC}$ *of exceptions* is given by one operation symbol $\mathsf{raise} : \mathsf{Exc} \longrightarrow 0$ and no equations. Another standard example is the *theory* $\mathcal{T}_{ND}$ *of nondeterminism*, which is given by one operation symbol $\mathsf{or} : 1 \longrightarrow 1 + 1$ and three equations that make or into a semilattice operation.

On the other hand, as the operation symbols of our fibred effect theories are dependently typed, compared to Plotkin and Pretnar's, we can naturally capture more precise notions of computation. We discuss two such examples below: i) a variant of Plotkin and Power's [2002] theory of global state in which the type of stored values is allowed to be dependent on memory locations; and ii) Ahman and Uustalu's [2014] dependently typed update monads that model state in which the store is changed not by overwriting but instead by applying (store-dependent) updates to it, examples of which include non-overflowing buffers and non-underflowing stacks—see op. cit. for details.

Global state. First, assume given pure value types of *memory locations* and *values* stored at them:

$$\diamond \vdash \mathsf{Loc} \qquad x{:}\mathsf{Loc} \vdash \mathsf{Val}$$

such that the propositional equality on Loc is *decidable*, i.e., we assume given a pure value term:

$$\diamond \vdash \mathtt{isDec}_{\mathsf{Loc}} : \Pi x{:}\mathsf{Loc}.\Pi x'{:}\mathsf{Loc}.(x =_{\mathsf{Loc}} x') + (x =_{\mathsf{Loc}} x' \to 0)$$

We use this assumption below to specify the two fibred effect theory equations that describe the commutativity of reading and writing at different memory locations (4th and 5th equation below). We also use this assumption, and results one can derive from it, to construct proofs of equational proof obligations when we define predicates on effectful computations using effect handlers in §7.

The fibred effect signature $\mathcal{S}_{GS}$ of global state is then given by the two operation symbols

$$\mathsf{get} : (x{:}\mathsf{Loc}) \longrightarrow \mathsf{Val} \qquad \mathsf{put} : \Sigma x{:}\mathsf{Loc}.\mathsf{Val} \longrightarrow 1$$

The idea here is that get denotes an effectful command that returns the current value of the store at the given location; and put denotes a command that overwrites the store at the given location.

The corresponding fibred effect theory $\mathcal{T}_{\text{GS}}$ is then given by the following five equations:

$$x{:}\mathsf{Loc} \mid w{:}1 \vdash \mathsf{get}_x(y.\mathsf{put}_{\langle x,y\rangle}(w\,(\star))) = w\,(\star)$$

$$x{:}\mathsf{Loc}, y{:}\mathsf{Val} \mid w{:}\mathsf{Val} \vdash \mathsf{put}_{\langle x,y\rangle}(\mathsf{get}_x(y'.w\,(y'))) = \mathsf{put}_{\langle x,y\rangle}(w\,(y))$$

$$x{:}\mathsf{Loc}, y_1{:}\mathsf{Val}, y_2{:}\mathsf{Val} \mid w{:}1 \vdash \mathsf{put}_{\langle x,y_1\rangle}(\mathsf{put}_{\langle x,y_2\rangle}(w\,(\star))) = \mathsf{put}_{\langle x,y_2\rangle}(w\,(\star))$$

$$\begin{aligned} x_1{:}\mathsf{Loc}, x_2{:}\mathsf{Loc} \mid w{:}\mathsf{Val}[x_1/x] \times \mathsf{Val}[x_2/x] \vdash\ & \mathsf{get}_{x_1}(y_1.\mathsf{get}_{x_2}(y_2.w\,(\langle y_1, y_2\rangle))) \\ = \ & \mathsf{get}_{x_2}(y_2.\mathsf{get}_{x_1}(y_1.w\,(\langle y_1, y_2\rangle))) \qquad (x_1 \neq x_2) \end{aligned}$$

$$x_1 : \mathsf{Loc}, x_2 : \mathsf{Loc}, y_1 : \mathsf{Val}[x_1/x], y_2 : \mathsf{Val}[x_2/x] \mid w : 1 \vdash \begin{array}{l} \mathsf{put}_{\langle x_1, y_1 \rangle}(\mathsf{put}_{\langle x_2, y_2 \rangle}(w\,(\star))) \\ \quad = \mathsf{put}_{\langle x_2, y_2 \rangle}(\mathsf{put}_{\langle x_1, y_1 \rangle}(w\,(\star))) \end{array} \qquad (x_1 \neq x_2)$$

where the last two equations include a side-condition that requires the locations x_1 and x_2 to be different. Similarly to Plotkin and Pretnar's effect theories, this is simply an informal shorthand notation. Formally, the right-hand sides of these two equations are written using case analysis on the assumed term $\mathtt{isDec}_{\mathsf{Loc}}\, x_1\, x_2$, e.g., the right-hand side of the last equation is formally given by

$$\begin{array}{l} \mathsf{case}\ (\mathtt{isDec}_{\mathsf{Loc}}\, x_1\, x_2)\ \mathsf{of}\ (\mathsf{inl}(y_p : x_1 =_{\mathsf{Loc}} x_2) \mapsto \mathsf{put}_{\langle x_1, y_1 \rangle}(\mathsf{put}_{\langle x_2, y_2 \rangle}(w\,(\star))), \\ \qquad\qquad\qquad\qquad\quad \mathsf{inr}(y_p : x_1 =_{\mathsf{Loc}} x_2 \to 0) \mapsto \mathsf{put}_{\langle x_2, y_2 \rangle}(\mathsf{put}_{\langle x_1, y_1 \rangle}(w\,(\star)))) \end{array}$$

Observe how these five equations describe the expected behaviour of get and put: trivial store changes are not observable (1st equation); `get` returns the most recent value the store has been set to (2nd equation); `put` overwrites the content of the store (3rd equation); and `gets` and `puts` at different locations are independent and commute with each other (4th and 5th equation).

Dependently typed update monads. To begin with, we assume given pure value types

$$\diamond \vdash \mathsf{St} \qquad x : \mathsf{St} \vdash \mathsf{Upd}$$

of *store values* and *store updates*, respectively, together with well-typed closed pure value terms

$$\downarrow : \Pi x : \mathsf{St}.\mathsf{Upd} \to \mathsf{St} \qquad \mathsf{o} : \Pi x : \mathsf{St}.\mathsf{Upd} \qquad \oplus : \Pi x : \mathsf{St}.\Pi y : \mathsf{Upd}.\mathsf{Upd}[x \downarrow y/x] \to \mathsf{Upd}$$

satisfying the following propositional equalities (for better readability, we omit the type annotations on these equations, and write the first argument to $\oplus$ as a subscript; we also leave implicit the use of the standard transport operation (e.g., see §7.1) in transporting W and W_3 in the last two equations along the first two equations so as to ensure that both sides of these equations are well-typed):

$$V \downarrow (\mathsf{o}\, V) = V \qquad V \downarrow (W_1 \oplus_V W_2) = (V \downarrow W_1) \downarrow W_2$$

$$W \oplus_V (\mathsf{o}\, (V \downarrow W)) = W \qquad (\mathsf{o}\, V) \oplus_V W = W \qquad (W_1 \oplus_V W_2) \oplus_V W_3 = W_1 \oplus_V (W_2 \oplus_{V \downarrow W_1} W_3)$$

In the literature, this structure is commonly called a *directed container* [Ahman et al. 2014]. For dependently typed update monads, the high-level idea is that $(\mathsf{Upd}, \mathsf{o}, \oplus)$ forms a dependently typed monoid of store updates which can be applied to the store values via its action $\downarrow$ on St.

The signature $\mathcal{S}_{\mathrm{UPD}}$ of a dependently typed update monad is then given by two operation symbols:

$$\mathsf{lookup} : 1 \longrightarrow \mathsf{St} \qquad \mathsf{update} : \Pi x : \mathsf{St}.\mathsf{Upd} \longrightarrow 1$$

The idea here is that lookup denotes an effectful command that returns the current value of the store; and update denotes a command that applies an appropriate update to the current store (from the family of updates given as its input). The dependency of Upd on St provides fine-grain control over which updates are applicable to which store values, and allows this to be enforced statically.

The corresponding fibred effect theory $\mathcal{T}_{\mathrm{UPD}}$ is then given by the following three equations:

$$\diamond \mid w : 1 \vdash \mathsf{lookup}(x.\mathsf{update}_{\lambda y : \mathsf{St}.\, \mathsf{o}\, y}(w\,(\star))) = w\,(\star)$$

$$\begin{array}{l} x : (\Pi x' : \mathsf{St}.\mathsf{Upd}[x'/x]) \mid w : \mathsf{St} \times \mathsf{St} \vdash \mathsf{lookup}(y.\mathsf{update}_x(\mathsf{lookup}(y'.w\,(\langle y, y' \rangle)))) \\ \qquad\qquad\qquad\qquad\qquad\qquad = \mathsf{lookup}(y.\mathsf{update}_x(w\,(\langle y, y \downarrow (x\, y) \rangle)))) \end{array}$$

$$\begin{array}{l} x : (\Pi x' : \mathsf{St}.\mathsf{Upd}[x'/x]), y : (\Pi y' : \mathsf{St}.\mathsf{Upd}[y'/x]) \mid w : 1 \vdash \mathsf{update}_x(\mathsf{update}_y(w\,(\star))) \\ \qquad\qquad\qquad\qquad\qquad\qquad = \mathsf{update}_{\lambda x''.(x\, x'') \oplus_{x''} (y\, (x'' \downarrow (x\, x'')))}(w\,(\star)) \end{array}$$

$$
\begin{aligned}
&(\!| w_j\,(V) |\!) && \stackrel{\text{def}}{=} V_j'\ (V[\overrightarrow{V_i}/\overrightarrow{x_i}]) \\
&(\!| \mathsf{op}_V(y.T) |\!) && \stackrel{\text{def}}{=} W_{\mathsf{op}}\ \langle V[\overrightarrow{V_i}/\overrightarrow{x_i}], \lambda y{:}O[V[\overrightarrow{V_i}/\overrightarrow{x_i}]/x].(\!| T |\!)\rangle \\
&(\!| \mathtt{pm}\ V\ \mathtt{as}\ (y_1{:}B_1, y_2{:}B_2)\ \mathtt{in}\ T |\!) && \stackrel{\text{def}}{=} \mathtt{pm}\ V[\overrightarrow{V_i}/\overrightarrow{x_i}]\ \mathtt{as}\ (y_1{:}B_1[\overrightarrow{V_i}/\overrightarrow{x_i}], y_2{:}B_2[\overrightarrow{V_i}/\overrightarrow{x_i}])\ \mathtt{in}\ (\!| T |\!) \\
&(\!| \mathtt{case}\ V\ \mathtt{of}\ (\mathsf{inl}(y_1{:}B_1) \mapsto T_1, \mathsf{inr}(y_2{:}B_2) \mapsto T_2) |\!) && \stackrel{\text{def}}{=} \mathtt{case}\ V[\overrightarrow{V_i}/\overrightarrow{x_i}]\ \mathtt{of}\ (\mathtt{inl}(y_1{:}B_1[\overrightarrow{V_i}/\overrightarrow{x_i}]) \mapsto (\!| T_1 |\!), \\
&&& \qquad\qquad \mathtt{inr}(y_2{:}B_2[\overrightarrow{V_i}/\overrightarrow{x_i}]) \mapsto (\!| T_2 |\!))
\end{aligned}
$$

Fig. 3. Translation of effect terms into value terms.

These equations are similar to the first three equations of the global state theory $\mathcal{T}_{\mathrm{GS}}$, but instead of an overwriting behaviour, they describe how the store is changed using updates. In particular, observe how $\oplus$ is used to combine subsequent updates, and how o gives us "do nothing" updates.

3.3 Extending EMLTT with Fibred Algebraic Effects

Next, we show how to extend EMLTT with algebraic effects given by a fibred effect theory $\mathcal{T}_{\mathrm{eff}}$.

First, we extend the grammar of EMLTT's computation terms with *algebraic operations*:

$$M \ ::= \ \ldots \ \mid \ \mathsf{op}_V^{\underline{C}}(y.M)$$

for all operation symbols $\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}}$ and computation types $\underline{C}$.

Next, in order to extend the well-formed syntax of EMLTT with a corresponding typing rule and definitional equations, we first define a translation of effect terms into value terms. In particular, given an effect term $\Gamma \mid \Delta \vdash T$, a value type A, value terms V_i (for all $x_i{:}A_i \in \Gamma$), value terms V_j' (for all $w_j{:}A_j' \in \Delta$), and value terms W_{op} (for all $\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}}$), we define the *translation* of T into a value term $(\!| T |\!)_{A;\overrightarrow{V_i};\overrightarrow{V_j'};\overrightarrow{W_{\mathsf{op}}}}$ by recursion on the structure of T, as given in detail in Fig. 3. For better readability, we write $\overrightarrow{V_i}$ for $\{V_1, \ldots, V_n\}$ in the translation, and similarly for $\overrightarrow{V_j'}$ and $\overrightarrow{W_{\mathsf{op}}}$.

While we omit the subscripts in Fig. 3 so as to improve readability, it is important to note that in the cases where the given effect term involves variable bindings, the set of value terms $\overrightarrow{V_i}$ is extended with the corresponding variables in the right-hand side, e.g., in the second case we have

$$(\!| \mathsf{op}_V(y.T) |\!)_{A;\overrightarrow{V_i};\overrightarrow{V_j'};\overrightarrow{W_{\mathsf{op}}}} \stackrel{\text{def}}{=} W_{\mathsf{op}}\ \langle V[\overrightarrow{V_i}/\overrightarrow{x_i}], \lambda y{:}O[V[\overrightarrow{V_i}/\overrightarrow{x_i}]/x].(\!| T |\!)_{A;\overrightarrow{V_i},y;\overrightarrow{V_j'};\overrightarrow{W_{\mathsf{op}}}}\rangle$$

While it might be more intuitive and natural to translate effect terms directly into EMLTT's computation terms, giving the translation from effect terms into value terms allows us to later reuse this translation in §4 when we extend EMLTT with handlers of fibred algebraic effects.

We only translate well-formed effect terms $\Gamma \mid \Delta \vdash T$ because it makes it easier to account for substituting value terms V_j' (denoting continuations) for effect variables—various subsequent results refer to substituting value terms for all variables in Δ, not just for the free ones appearing in T.

Using this translation of effect terms into value terms, we can now define the typing rule and definitional equations for algebraic operations, as given in Fig. 4. It is worth noting that for presentational convenience, we include the equations given in $\mathcal{E}_{\mathrm{eff}}$ as definitional equations between value terms. The corresponding equations between computation terms are easily derivable, e.g.,

$$\Gamma \vdash \mathsf{get}_V^{\underline{C}}(y.\mathsf{put}_{\langle V, y\rangle}^{\underline{C}}(M)) = M : \underline{C}$$

can be derived from the translation of the corresponding equation in the global state theory $\mathcal{T}_{\mathrm{GS}}$.

Typing rule for algebraic operations:

$$\frac{\Gamma \vdash V : I \quad \Gamma \vdash \underline{C} \quad \Gamma, y{:}O[V/x] \vdash M : \underline{C}}{\Gamma \vdash \mathsf{op}_V^C(y.M) : \underline{C}} \; (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}})$$

Congruence equations:

$$\frac{\Gamma \vdash V = W : I \quad \Gamma \vdash \underline{C} = \underline{D} \quad \Gamma, y{:}O[V/x] \vdash M = N : \underline{C}}{\Gamma \vdash \mathsf{op}_V^C(y.M) = \mathsf{op}_W^D(y.N) : \underline{C}} \; (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}})$$

General algebraicity equations:

$$\frac{\Gamma \vdash V : I \quad \Gamma, y{:}O[V/x] \vdash M : \underline{C} \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash K[\mathsf{op}_V^C(y.M)/z] = \mathsf{op}_V^D(y.K[M/z]) : \underline{D}} \; (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}})$$

Equations from the given fibred effect theory:

$$\frac{\begin{array}{ll}\Gamma' \vdash V_i : A_i[V_1/x_1, \ldots, V_{i-1}/x_{i-1}] & (1 \leq i \leq n) \\ \Gamma' \vdash \underline{C} \quad \Gamma' \vdash V_j' : A_j'[\overrightarrow{V_i}/\overrightarrow{x_i}] \to U\underline{C} & (1 \leq j \leq m)\end{array}}{\Gamma' \vdash (\!|T_1|\!)_{U\underline{C};\overrightarrow{V_i};\overrightarrow{V_j'};\overrightarrow{W_{\mathrm{op}}}} = (\!|T_2|\!)_{U\underline{C};\overrightarrow{V_i};\overrightarrow{V_j'};\overrightarrow{W_{\mathrm{op}}}} : U\underline{C}} \; (\Gamma \mid \Delta \vdash T_1 = T_2 \in \mathcal{E}_{\mathrm{eff}})$$

where the well-typed value terms $\Gamma' \vdash W_{\mathrm{op}} : (\Sigma x{:}I.O \to U\underline{C}) \to U\underline{C}$ are defined as follows:

$$W_{\mathrm{op}} \stackrel{\mathrm{def}}{=} \lambda x' {:} (\Sigma x{:}I.O \to U\underline{C}).\, \mathtt{pm}\ x'\ \mathtt{as}\ (x{:}I, y{:}O \to U\underline{C})\ \mathtt{in}\ \mathtt{thunk}\ (\mathsf{op}_x^C(y'.\mathtt{force}_{\underline{C}}\,(y\ y')))$$

for all $\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}}$.

Fig. 4. Rules for extending EMLTT with fibred algebraic effects.

4 HANDLERS VIA THE USER-DEFINED ALGEBRA TYPE

4.1 A Problem with Adding Conventional Handlers to EMLTT

Before we show how to extend EMLTT with handlers of fibred algebraic effects using the user-defined algebra type, we first explain how extending EMLTT with the conventional term-level definition of handlers quickly leads to *unsound* program equivalences becoming derivable.

First, recall from §1 that Plotkin and Pretnar (and others since) include handlers in effectful languages by extending the syntax of computation terms with the following handling construct:

$$M\ \mathtt{handled}\ \mathtt{with}\ \{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathrm{eff}}}\ \mathtt{to}\ y{:}A\ \mathtt{in}\ N_{\mathrm{ret}}$$

whose semantics is given using the mediating homomorphism from the free algebra over A to the algebra denoted by the handler $\{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathrm{eff}}}$. However, when extending a language that includes a notion of homomorphism, such as EMLTT with its homomorphism terms, this algebraic understanding of handlers suggests that one ought to also extend the given notion of homomorphism with a corresponding handling construct. Unfortunately, if one simply adds

$$K\ \mathtt{handled}\ \mathtt{with}\ \{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathrm{eff}}}\ \mathtt{to}\ y{:}A\ \mathtt{in}\ N_{\mathrm{ret}}$$

to EMLTT, the combination of i) the β-equations associated with the handling construct (see §1) and ii) the general algebraicity equations (see Fig. 4) gives rise to unsound definitional equations.

To explain this problem in more detail, let us consider the theory $\mathcal{T}_{\mathrm{I/O}}$ of *interactive input-output of bits*, given by two operation symbols, $\mathsf{read} : 1 \longrightarrow 1 + 1$ and $\mathsf{write} : 1 + 1 \longrightarrow 1$, and no equations.

Next, we define a handler that negates all bits read from the input and written to the output:

$$\mathtt{read}(x') \mapsto \mathtt{read}^{F1}(y.\mathtt{force}\ (x'\ \neg y)) \qquad \mathtt{write}_x(x') \mapsto \mathtt{write}^{F1}_{\neg x}(\mathtt{force}\ (x'\ \star))$$

where $\neg : 1 + 1 \to 1 + 1$ denotes negation of bits, swapping the left and right injections into $1 + 1$.

Now, let us consider handling a simple program, $\mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star)$, using the handler we defined above.[2] On the one hand, using the β-equations for handling (see §1), we can prove that

$$\begin{aligned}
&\Gamma \vdash (\mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star))\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star \\
&\quad = \mathtt{write}^{F1}_{\neg(\mathtt{inl}\,\star)}((\mathtt{return}\ \star)\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star) \\
&\quad = \mathtt{write}^{F1}_{\neg(\mathtt{inl}\,\star)}(\mathtt{return}\ \star) \\
&\quad = \mathtt{write}^{F1}_{\mathtt{inr}\,\star}(\mathtt{return}\ \star) : F1
\end{aligned}$$

On the other hand, using the general algebraicity equations (see Fig. 4), which ensure that homomorphism terms indeed behave as if they were algebra homomorphisms, we can prove that

$$\begin{aligned}
&\Gamma \vdash (\mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star))\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star \\
&\quad = (z\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star)[\mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star)/z] \\
&\quad = \mathtt{write}^{F1}_{\mathtt{inl}\,\star}((z\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star)[\mathtt{return}\ \star/z]) \\
&\quad = \mathtt{write}^{F1}_{\mathtt{inl}\,\star}((\mathtt{return}\ \star)\ \mathtt{handled\ with}\ \{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y{:}1\ \mathtt{in}\ \mathtt{return}\ \star) \\
&\quad = \mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star) : F1
\end{aligned}$$

Finally, by combining these two sequences of equations using transitivity, we can prove that

$$\Gamma \vdash \mathtt{write}^{F1}_{\mathtt{inr}\,\star}(\mathtt{return}\ \star) = \mathtt{write}^{F1}_{\mathtt{inl}\,\star}(\mathtt{return}\ \star) : F1$$

which is clearly only valid in trivial models of interactive output, and thus should not be derivable.

The source of this problem lies in the term-level definition of handlers in their usual presentation. In particular, while the homomorphic behaviour of homomorphism terms is determined exclusively by the computation types involved (via the general algebraicity equations), the type of the above handling construct contains no trace of the algebra denoted by the handler $\{\mathtt{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{I/O}}}$.

It is worth noting that this problem is not inherent to EMLTT but also arises in the simply typed setting, e.g., when combining handlers of algebraic effects with CBPV and its stack terms, or with EEC and its linear (computation) terms. For example, CBPV with stack terms and exception handlers has been investigated by Levy [2006]. However, compared to the solution we propose in this paper, Levy follows the opposite direction: namely, while we aim to simultaneously accommodate effect handlers and make sure that homomorphism terms still denote algebra homomorphisms, Levy changes the syntax and equational theory of stack terms so that they can accommodate exception handlers, but with the cost of stack terms not denoting algebra homomorphisms any more.

Finally, we note that the reason why Plotkin and Pretnar [2013] were able to give a sound denotational semantics to their language was precisely due to their choice of using CBPV *without* stack terms, i.e., without a notion of homomorphism, only with value and computation terms.

4.2 Extending EMLTT with the User-Defined Algebra Type

In this section we solve the problems of §4.1 by giving handlers a novel type-based treatment that internalises Plotkin and Pretnar's insight that they denote algebras for the given effect theory.

[2] Note that for this illustrative example, we could have omitted read and only considered the signature of interactive output.

First, given a fibred effect theory $\mathcal{T}_{\text{eff}}$, we extend EMLTT with the *user-defined algebra type*:

$$\underline{C} \;::=\; \ldots \mid \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$$

which pairs a value type A (the carrier) with two sets of value terms, $\overrightarrow{V_{\text{op}}}$ (the operations) and $\overrightarrow{W_{\text{eq}}}$ (the proofs of equational proof obligations showing that $\overrightarrow{V_{\text{op}}}$ satisfy the equations given in $\mathcal{E}_{\text{eff}}$).

We also extend EMLTT's computation and homomorphism terms with two *composition operations*:

$$M \;::=\; \ldots \mid M \,\mathtt{as}\, x{:}U\underline{C} \,\mathtt{in}_{\overrightarrow{W_{\text{op}}};\underline{D}}\, N$$
$$K \;::=\; \ldots \mid K \,\mathtt{as}\, x{:}U\underline{C} \,\mathtt{in}_{\overrightarrow{W_{\text{op}}};\underline{D}}\, N$$

where $\overrightarrow{W_{\text{op}}}$ are proofs of equational proof obligations showing that the computation term N behaves as if it was a homomorphism term—we discuss this in detail below. These term formers also provide *elimination* forms for the user-defined algebra type when $\underline{C}$ is $\langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$; we also note that terms of this type are *introduced* by forcing thunks of type $U\langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$, i.e., value terms of type A.

For better readability, we write $\overrightarrow{\mathtt{refl}}$ for $\overrightarrow{W_{\text{eq}}}$ and $\overrightarrow{W_{\text{op}}}$ in the rest of this paper when all the proof witnesses are given by reflexivity, i.e., when the corresponding equational proof obligations hold definitionally. Also for better readability, we write $\langle A; \overrightarrow{V_{\text{op}}} \rangle$ for $\langle A; \overrightarrow{V_{\text{op}}}; \emptyset \rangle$ when $\mathcal{E}_{\text{eff}}$ is empty.

It is worth noting that in principle we could have restricted these composition operations to only the user-defined algebra type $\langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$, but then we would not have been able to derive a useful isomorphism of computation types to coerce computations between a general $\underline{C}$ and its canonical representation as a user-defined algebra type—see Prop. 4.1 for details of this type isomorphism.

Conceptually, these composition operations are a form of *explicit substitution* of thunked computations for value variables. For example, in this extension of EMLTT we will be able to show that $M \,\mathtt{as}\, x{:}U\underline{C} \,\mathtt{in}_{\overrightarrow{W_{\text{op}}};\underline{D}}\, N$ is definitionally equal to $N[\mathtt{thunk}\, M/x]$. As such, the value variable x refers to the whole of (the thunk of) M, compared to, e.g., sequential composition $M \,\mathtt{to}\, x{:}A \,\mathtt{in}\, N$, where the value variable x is used to bind only the value produced by M. However, it is also important to note that we do not allow arbitrary computation terms to be used in these composition operations. In particular, the typing rules of $M \,\mathtt{as}\, x{:}U\underline{C} \,\mathtt{in}_{\overrightarrow{W_{\text{op}}};\underline{D}}\, N$ and $K \,\mathtt{as}\, x{:}U\underline{C} \,\mathtt{in}_{\overrightarrow{W_{\text{op}}};\underline{D}}\, N$ require that the value variable x is used in the computation term N as if it was a computation variable, in that x must not be duplicated or discarded arbitrarily. We do this in order to ensure that N behaves as if it was a homomorphism term, so that in $N[\mathtt{thunk}\, M/x]$ the effects of M would be guaranteed to "happen before" those of N. However, rather than trying to extend EMLTT further with some form of linearity for such value variables, we impose these requirements via equational proof obligations, requiring that N commutes with algebraic operations (when substituted for x via thunking). As mentioned earlier, the proofs of these proof obligations are given by the value terms $\overrightarrow{W_{\text{op}}}$.

We make this discussion formal in Fig. 5 by giving the rules for extending EMLTT's well-formed syntax and equational theory with the user-defined algebra type and composition operations, using the following *auxiliary judgement* in the rules concerning the two composition operations:

$$\Gamma, y{:}U\underline{C} \vdash_{\text{hom}} N : \underline{D} \text{ witnessed by } \overrightarrow{W_{\text{op}}}$$

which holds iff N behaves like a *homomorphism* from the algebra denoted by $\underline{C}$ to that denoted by $\underline{D}$, *witnessed by* $\overrightarrow{W_{\text{op}}}$, i.e., $\Gamma, y{:}U\underline{C} \vdash N : \underline{D}$ and we have for each $\text{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\text{eff}}$ a proof

$$\Gamma \vdash W_{\text{op}} : \lambda x{:}I.\lambda x'{:}O \to U\underline{C}.\mathtt{thunk}\,(N[\mathtt{thunk}\,(\mathrm{op}_x^{\underline{C}}(x''.\mathtt{force}_{\underline{C}}\,(x'\,x'')))/y]) = \lambda x{:}I.\lambda x'{:}O \to U\underline{C}.\mathtt{thunk}\,(\mathrm{op}_x^{\underline{D}}(x''.N[x'\,x''/y]))$$

Formation rule for the user-defined algebra type:

$$\frac{\Gamma' \vdash A \quad \Gamma' \vdash V_{\mathsf{op}} : (\Sigma x{:}I.O \to A) \to A \qquad (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}} \text{ and } \Gamma \mid \Delta \vdash T_1 = T_2 \in \mathcal{E}_{\mathrm{eff}})}{\Gamma' \vdash W_{\mathsf{eq}} : \overrightarrow{\lambda x'_i{:}\widehat{A_i}.\, \lambda x_{w_j}{:}\widehat{A'_j} \to A.\, (\!|\, T_1 \,|\!)_{A;\overrightarrow{x'_i};\overrightarrow{x_{w_j}};\overrightarrow{V_{\mathsf{op}}}}} = \overrightarrow{\lambda x'_i{:}\widehat{A_i}.\, \lambda x_{w_j}{:}\widehat{A'_j} \to A.\, (\!|\, T_2 \,|\!)_{A;\overrightarrow{x'_i};\overrightarrow{x_{w_j}};\overrightarrow{V_{\mathsf{op}}}}}}{\Gamma' \vdash \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}$$

where $\widehat{A_i} \stackrel{\text{def}}{=} A_i[x'_1/x_1, \ldots, x'_{i-1}/x_{i-1}]$ and $\widehat{A'_j} \stackrel{\text{def}}{=} A'_j[x'_1/x_1, \ldots, x'_n/x_n]$; and where we write $\overrightarrow{\lambda x'_i{:}\widehat{A_i}.}$, $\overrightarrow{\lambda x_{w_j}{:}\widehat{A'_j} \to A.}$, $\overrightarrow{\Pi x'_i{:}\widehat{A_i}}$, and $\overrightarrow{\widehat{A'_j} \to A}$ for sequences of lambda abstractions and function types.

Typing rules for the composition operations:

$$\frac{\Gamma \vdash M : \underline{C} \quad \Gamma \vdash \underline{D} \qquad \Gamma, x{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D} \text{ witnessed by } \overrightarrow{W_{\mathsf{op}}}}{\Gamma \vdash M \text{ as } x{:}U\underline{C} \text{ in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}}\ N : \underline{D}} \qquad \frac{\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}_1 \quad \Gamma \vdash \underline{D}_2 \qquad \Gamma, x{:}U\underline{D}_1 \vdash_{\mathsf{hom}} N : \underline{D}_2 \text{ witnessed by } \overrightarrow{W_{\mathsf{op}}}}{\Gamma \mid z{:}\underline{C} \vdash K \text{ as } x : U\underline{D}_1 \text{ in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}_2}\ N : \underline{D}_2}$$

Congruence equations:

$$\frac{\Gamma \vdash \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{V_{\mathsf{eq}}} \rangle \quad \Gamma \vdash \langle B; \overrightarrow{W_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle \quad \Gamma \vdash A = B \qquad \Gamma \vdash V_{\mathsf{op}} = W_{\mathsf{op}} : (\Sigma x{:}I.O \to A) \to A \quad (\mathsf{op} : (x{:}I) \longrightarrow O)}{\Gamma \vdash \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{V_{\mathsf{eq}}} \rangle = \langle B; \overrightarrow{W_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}$$

plus similar equations for composition operations (note the proof irrelevant treatment of $\overrightarrow{V_{\mathsf{eq}}}$ and $\overrightarrow{W_{\mathsf{eq}}}$).

β-equation for the user-defined algebra type:

$$\frac{\Gamma \vdash \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}{\Gamma \vdash U\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle = A}$$

capturing the intuition that A denotes the carrier of the algebra denoted by $\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle$.

β- and η-equations for the composition operations:

$$\frac{\Gamma \vdash V : U\underline{C} \quad \Gamma \vdash \underline{D} \quad \Gamma, x{:}U\underline{C} \vdash_{\mathsf{hom}} M : \underline{D} \text{ witnessed by } \overrightarrow{W_{\mathsf{op}}}}{\Gamma \vdash (\mathtt{force}_{\underline{C}}\ V) \text{ as } x{:}U\underline{C} \text{ in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}}\ M = M[V/x] : \underline{D}}$$

$$\frac{\Gamma \vdash M : \underline{C} \quad \Gamma \mid z{:}\underline{C} \vdash K : \underline{D}}{\Gamma \vdash M \text{ as } x{:}U\underline{C} \text{ in}_{\overrightarrow{\mathtt{refl}};\underline{D}}\ K[\mathtt{force}_{\underline{C}}\ x/z] = K[M/z] : \underline{D}}$$

$$\frac{\Gamma \mid z_1{:}\underline{C} \vdash K : \underline{D}_1 \quad \Gamma \mid z_2{:}\underline{D}_1 \vdash L : \underline{D}_2}{\Gamma \mid z_1{:}\underline{C} \vdash K \text{ as } x{:}U\underline{D}_1 \text{ in}_{\overrightarrow{\mathtt{refl}};\underline{D}_2}\ L[\mathtt{force}_{\underline{D}_1}\ x/z_2] = L[K/z_2] : \underline{D}_2}$$

capturing the intuition that composition operations are a form of explicit substitution of thunks.

η-equation for algebraic operations:

$$\frac{\Gamma \vdash V : I \quad \Gamma \vdash \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle \quad \Gamma, y{:}O[V/x] \vdash M : \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}{\begin{array}{l}\Gamma \vdash \mathsf{op}_V^{\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}(y.M) \\ \quad = \mathtt{force}_{\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle}\ (V_{\mathsf{op}}\ \langle V, \lambda y{:}O[V/x].\mathtt{thunk}\ M \rangle) : \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle\end{array}} \ (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathrm{eff}})$$

capturing the intuition that $\overrightarrow{V_{\mathsf{op}}}$ denote the operations of the algebra denoted by $\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle$.

Fig. 5. Rules for extending EMLTT with the user-defined algebra type and the composition operations.

It is worth noting that by defining this extension of EMLTT using propositional proof obligations instead of definitional ones, as in Ahman [2017], we introduce a form of *equality reflection* into the language. Namely, in any model of this extension of EMLTT, the propositional proof obligations proved by $\overrightarrow{W_{\mathsf{eq}}}$ and $\overrightarrow{W_{\mathsf{op}}}$ have to also ensure that the corresponding definitional equations are validated in the given model, so as to validate the definitional equations given for algebraic effects in Fig. 4. Based on this extensional nature of these proof obligations, we have naturally chosen to treat them as proof irrelevant in the user-defined algebra type and the composition operations, namely, by not comparing their proofs for definitional equality in the congruence rules (see Fig. 5).

It is also useful to note that if one works exclusively with equation-free fibred effect theories, e.g., as used in the various simply typed languages mentioned in §1 that support algebraic effects and their handlers, then the equational proof obligations used in the typing rule of $\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}}\rangle$ hold vacuously, and thus do not put any additional burden on the programmer. However, the possibility of also being able to specify effects using equations ensures that the fit-for-purpose handler implementations of the given notion of computation, say, global state, are indeed correct.

We conclude this section by highlighting that we have not included an η-equation for $\langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}}\rangle$. We do so because it does not hold in the Lawvere theories based semantics we give to this extension of EMLTT in §8. Instead, as promised earlier, we can construct a corresponding type isomorphism.

Proposition 4.1. *Given a computation type* $\Gamma \vdash \underline{C}$*, there exists a computation type isomorphism* $\Gamma \vdash \underline{C} \cong \langle U\underline{C}; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{\mathtt{refl}}\rangle$*, where each value term* $\Gamma \vdash V_{\mathsf{op}} : (\Sigma x{:}I.O \to U\underline{C}) \to U\underline{C}$ *is given by*

$$V_{\mathsf{op}} \stackrel{def}{=} \lambda y{:}(\Sigma x{:}I.O \to U\underline{C}).\mathtt{pm}\ y\ \mathtt{as}\ (x{:}I, x'{:}O \to U\underline{C})\ \mathtt{in}\ \mathtt{thunk}\,(\mathsf{op}^{\underline{C}}_{x}(y'.\mathtt{force}_{\underline{C}}\,(x'\ y')))$$

Proof. This type isomorphism is witnessed by the following two homomorphic functions:

$$\Gamma \vdash \lambda z{:}\underline{C}.\ z\ \mathtt{as}\ x{:}U\underline{C}\ \mathtt{in}\ \mathtt{force}_{\langle U\underline{C};\overrightarrow{V_{\mathsf{op}}};\overrightarrow{\mathtt{refl}}\rangle}\ x : \underline{C} \multimap \langle U\underline{C}; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{\mathtt{refl}}\rangle$$

$$\Gamma \vdash \lambda z{:}\langle U\underline{C}; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{\mathtt{refl}}\rangle.\ z\ \mathtt{as}\ x{:}U\langle U\underline{C}; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{\mathtt{refl}}\rangle\ \mathtt{in}\ \mathtt{force}_{\underline{C}}\ x : \langle U\underline{C}; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{\mathtt{refl}}\rangle \multimap \underline{C}$$

□

4.3 Deriving the Conventional Term-Level Definition of Handlers

We now show how to derive the conventional term-level definition of handlers from our type-based treatment. In particular, we define the *handling construct* using sequential composition as follows:

$$M\ \mathtt{handled}\ \mathtt{with}\ (\{\mathsf{op}_x(x') \mapsto N_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathsf{eff}}}; \overrightarrow{W_{\mathsf{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_{\underline{C}}\ N_{\mathsf{ret}}$$

$$\stackrel{\mathsf{def}}{=}$$

$$\mathtt{force}_{\underline{C}}\,(\mathtt{thunk}\,(M\ \mathtt{to}\ y{:}A\ \mathtt{in}\ \mathtt{force}_{\langle U\underline{C};\overrightarrow{V_{\mathsf{op}}};\overrightarrow{W_{\mathsf{eq}}}\rangle}\,(\mathtt{thunk}\ N_{\mathsf{ret}})))$$

where the value terms V_{op} are defined as follows:

$$V_{\mathsf{op}} \stackrel{\mathsf{def}}{=} \lambda x''{:}(\Sigma x{:}I.O \to U\underline{C}).\,\mathtt{pm}\ x''\ \mathtt{as}\ (x{:}I, x'{:}O \to U\underline{C})\ \mathtt{in}\ \mathtt{thunk}\ N_{\mathsf{op}}$$

Observe that compared to the work of Plotkin and Pretnar [2013], who do not enforce the correctness of their handlers during typechecking and therefore have to give semantics to their language using Kleene equality, we require the set of user-defined computation terms N_{op} to satisfy the equations given in $\mathcal{E}_{\mathsf{eff}}$ (as witnessed by the value terms $\overrightarrow{W_{\mathsf{eq}}}$), so as to statically ensure that the given computation terms N_{op} indeed form an algebra. In particular, Plotkin and Pretnar do not enforce the correctness of their handlers during typechecking because it is in general an undecidable problem [Plotkin and Pretnar 2013, §6]. In contrast, by having defined the corresponding equational

proof obligations in this paper using propositional equality, we can naturally accommodate manual user-provided proofs of equations that one cannot establish through automatic means.

The expected typing rule and definitional β-equations are then routinely derivable for this definition of the handling construct.

Proposition 4.2. *The following typing rule is derivable:*

$$\frac{\Gamma \vdash M : FA \quad \Gamma \vdash \langle U\underline{C}; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle \quad \Gamma, y{:}A \vdash N_{\mathrm{ret}} : \underline{C}}{\Gamma \vdash M \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathit{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_{\underline{C}}\ N_{\mathrm{ret}} : \underline{C}}$$

where each value term V_{op} is derived from the corresponding computation term N_{op} as defined above.

Proposition 4.3. *The following definitional β-equations are derivable:*

$$\frac{\Gamma \vdash V : I \quad \Gamma, y'{:}O[V/x] \vdash M : FA \quad \Gamma \vdash \langle U\underline{C}; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle \quad \Gamma, y{:}A \vdash N_{\mathrm{ret}} : \underline{C}}{\begin{array}{l}\Gamma \vdash (\mathsf{op}_V^{FA}(y'.M)) \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathit{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_{\underline{C}}\ N_{\mathrm{ret}} \\ \quad = N_{\mathrm{op}}[V/x, \lambda y'{:}O[V/x].\mathtt{thunk}\ H/x'] : \underline{C}\end{array}}$$

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash \langle U\underline{C}; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle \quad \Gamma, y{:}A \vdash N_{\mathrm{ret}} : \underline{C}}{\begin{array}{l}\Gamma \vdash (\mathtt{return}\ V) \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathit{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_{\underline{C}}\ N_{\mathrm{ret}} \\ \quad = N_{\mathrm{ret}}[V/y] : \underline{C}\end{array}}$$

where

$$H \stackrel{\mathit{def}}{=} M \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto N_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathit{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_{\underline{C}}\ N_{\mathrm{ret}}$$

and each value term V_{op} is derived from the corresponding computation term N_{op} as defined above.

4.4 Handling Computations into Values

We conclude this section by noting that in addition to the standard "handle into computation terms" handling construct we derived in §4.3, we can also use the user-defined algebra type and sequential composition to define a handling construct that allows computations to be handled directly into value terms, e.g., as briefly discussed in Ahman and Staton [2013] in the context of Levy's [2004] fine-grain call-by-value language. This "handle into value terms" handling construct is given by

$$\begin{array}{c} M \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto V_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathrm{ret}} \\ \stackrel{\mathrm{def}}{=} \\ \mathtt{thunk}\ (M\ \mathtt{to}\ y{:}A\ \mathtt{in}\ \mathtt{force}_{\langle \mathsf{B}; \overrightarrow{W_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle}\ V_{\mathrm{ret}}) \end{array}$$

where the value terms W_{op} are defined as follows:

$$W_{\mathrm{op}} \stackrel{\mathrm{def}}{=} \lambda x'' {:} (\Sigma x{:}I.O \to B).\ \mathtt{pm}\ x''\ \mathtt{as}\ (x{:}I, x'{:}O \to B)\ \mathtt{in}\ V_{\mathrm{op}}$$

and it satisfies the expected typing rule and definitional β-equations:

$$\frac{\Gamma \vdash M : FA \quad \Gamma \vdash \langle \mathsf{B}; \overrightarrow{W_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle \quad \Gamma, y{:}A \vdash V_{\mathrm{ret}} : B}{\Gamma \vdash M \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto V_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathrm{ret}} : B}$$

$$\frac{\Gamma \vdash V : I \quad \Gamma, y'{:}O[V/x] \vdash M : FA \quad \Gamma \vdash \langle \mathsf{B}; \overrightarrow{W_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}}\rangle \quad \Gamma, y{:}A \vdash V_{\mathrm{ret}} : B}{\begin{array}{l}\Gamma \vdash (\mathsf{op}_V^{FA}(y'.M)) \,\mathtt{handled\ with}\, (\{\mathsf{op}_x(x') \mapsto V_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{S}_{\mathrm{eff}}}; \overrightarrow{W_{\mathrm{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathrm{ret}} \\ \quad = V_{\mathrm{op}}[V/x, \lambda y'{:}O[V/x].H/x'] : B\end{array}}$$

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash \langle \mathsf{B}; \overrightarrow{W_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle \quad \Gamma, y{:}A \vdash V_{\mathsf{ret}} : B}{\begin{array}{l}\Gamma \vdash (\mathtt{return}\ V)\ \mathtt{handled}\ \mathtt{with}\ (\{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathsf{eff}}}; \overrightarrow{W_{\mathsf{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathsf{ret}} \\ \quad = V_{\mathsf{ret}}[V/y] : B\end{array}}$$

where

$$H \stackrel{\text{def}}{=} M\ \mathtt{handled}\ \mathtt{with}\ (\{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathsf{eff}}}; \overrightarrow{W_{\mathsf{eq}}})\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathsf{ret}}$$

In the case the set of equations $\mathcal{E}_{\mathsf{eff}}$ is empty, we write this handling construct simply as

$$M\ \mathtt{handled}\ \mathtt{with}\ \{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{S}_{\mathsf{eff}}}\ \mathtt{to}\ y{:}A\ \mathtt{in}_B\ V_{\mathsf{ret}}$$

5 BASIC META-THEORY

We now discuss some properties of our extension of EMLTT with algebraic effects and their handlers.

5.1 Weakening and Substitution

We begin by noting that, as expected, weakening is admissible for value variables.

Theorem 5.1 (Weakening). *Given $\Gamma_1, \Gamma_2 \vdash B$, $\Gamma_1 \vdash A$, and x such that $x \notin Vars(\Gamma_1) \cup Vars(\Gamma_2)$, then $\Gamma_1, x{:}A, \Gamma_2 \vdash B$, and similarly for all other judgements of types, terms, and definitional equations.*

Next, we note that, as also expected, substitution is admissible for both value and computation variables. As various typing rules and definitional equations now include (translations of) effect terms, we also need to prove the corresponding property for the translation of effect terms.

Proposition 5.2. *Given $\Gamma \mid \Delta \vdash T$, a value type A, value terms V_i (for all $x_i{:}A_i \in \Gamma$), V'_j (for all $w_j{:}A'_j \in \Delta$), and W_{op} (for all $\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathit{eff}}$), a value variable y, and a value term W, then*

$$(\!|\, T \,|\!)_{A;\overrightarrow{V_i};\overrightarrow{V'_j};\overrightarrow{W_{\mathsf{op}}}}[W/y] = (\!|\, T \,|\!)_{A[W/y];\overrightarrow{V_i[W/y]};\overrightarrow{V'_j[W/y]};\overrightarrow{W_{\mathsf{op}}[W/y]}}$$

Theorem 5.3 (Substitution).

- *Given $\Gamma_1, x{:}A, \Gamma_2 \vdash B$ and $\Gamma_1 \vdash V : A$, then $\Gamma_1, \Gamma_2[V/x] \vdash B[V/x]$, and similarly for other judgements of types, terms, and definitional equations.*
- *Given $\Gamma \mid z{:}\underline{C} \vdash K : \underline{D}$ and $\Gamma \vdash M : \underline{C}$, then $\Gamma \vdash K[M/z] : \underline{D}$.*
- *Given $\Gamma \mid z_2{:}\underline{D}_1 \vdash L : \underline{D}_2$ and $\Gamma \mid z_1{:}\underline{C} \vdash K : \underline{D}_1$, then $\Gamma \mid z_1{:}\underline{C} \vdash L[K/z_2] : \underline{D}_2$.*

Finally, we note that judgements of well-formed types, etc. only refer to well-formed contexts, etc. To this end, we also need to show that under suitable assumptions, $(\!|\, T \,|\!)_{A;\overrightarrow{V_i};\overrightarrow{V'_j};\overrightarrow{W_{\mathsf{op}}}}$ is well-typed.

Proposition 5.4. *Given $\Gamma \mid \Delta \vdash T$ and Γ', such that $\Gamma' \vdash A$ and the value terms in the subscripts are well-typed in Γ' (as required in Fig. 4), then we have $\Gamma' \vdash (\!|\, T \,|\!)_{A;\overrightarrow{V_i};\overrightarrow{V'_j};\overrightarrow{W_{\mathsf{op}}}} : A$.*

Proposition 5.5. *Given $\Gamma \mid \Delta \vdash T$ and Γ', such that $\Gamma' \vdash A = B$ and the corresponding value terms in the subscripts are definitionally equal in Γ' (analogously to the typing in Fig. 4), then we have*

$$\Gamma' \vdash (\!|\, T \,|\!)_{A;\overrightarrow{V_i};\overrightarrow{V'_j};\overrightarrow{V_{\mathsf{op}}}} = (\!|\, T \,|\!)_{B;\overrightarrow{W_i};\overrightarrow{W'_j};\overrightarrow{W_{\mathsf{op}}}} : A$$

Theorem 5.6. *Given $\Gamma \vdash V : A$, then $\vdash \Gamma$ and $\Gamma \vdash A$, and similarly for all other judgements.*

5.2 Derivable Definitional Equations

To begin with, one can derive specialised versions of the general algebraicity equations from Fig. 4.

Proposition 5.7. *We can derive the following specialised algebraicity equations:*

$$\frac{\Gamma \vdash V : I \quad \Gamma, y{:}O[V/x] \vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, y'{:}A \vdash N : \underline{C}}{\Gamma \vdash \mathsf{op}_V^{FA}(y.M)\ \mathtt{to}\ y'{:}A\ \mathtt{in}\ N = \mathsf{op}_V^{\underline{C}}(y.M\ \mathtt{to}\ y'{:}A\ \mathtt{in}\ N) : \underline{C}}\ (\mathsf{op} : (x{:}I) \longrightarrow O \in \mathcal{S}_{\mathit{eff}})$$

and analogously for other homomorphism term formers.

Next, we present some useful derivable equations for the composition operations.

Proposition 5.8. *We can derive the following unitality and associativity equations:*

$$\frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash M\ \mathtt{as}\ x{:}U\underline{C}\ \mathtt{in}\ \mathsf{force}_{\underline{C}}\, x = M : \underline{C}}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \underline{C}_1 \quad \Gamma \vdash \underline{C}_2 \quad \Gamma \vdash \underline{D} \\ \Gamma, x_1{:}U\underline{C}_1 \vdash_{\mathsf{hom}} N_1 : \underline{C}_2\ \text{witnessed by}\ \overrightarrow{W_{\mathsf{op}}} \quad \Gamma, x_2{:}U\underline{C}_2 \vdash_{\mathsf{hom}} N_2 : \underline{D}\ \text{witnessed by}\ \overrightarrow{W'_{\mathsf{op}}}\end{array}}{\Gamma \vdash M\ \mathtt{as}\ x_1{:}U\underline{C}_1\ \mathtt{in}\ (N_1\ \mathtt{as}\ x_2{:}U\underline{C}_2\ \mathtt{in}\ N_2) = (M\ \mathtt{as}\ x_1{:}U\underline{C}_1\ \mathtt{in}\ N_1)\ \mathtt{as}\ x_2{:}U\underline{C}_2\ \mathtt{in}\ N_2 : \underline{D}}$$

and analogously for the composition operation for homomorphism terms.

Proposition 5.9. *We can derive the following interaction equations:*

$$\frac{\Gamma \vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma \vdash \underline{D} \quad \Gamma, x_1{:}A \vdash N_1 : \underline{C} \quad \Gamma, x_2{:}U\underline{C} \vdash_{\mathsf{hom}} N_2 : \underline{D}\ \text{witnessed by}\ \overrightarrow{W_{\mathsf{op}}}}{\Gamma \vdash M\ \mathtt{to}\ x_1{:}A\ \mathtt{in}\ (N_1\ \mathtt{as}\ x_2{:}U\underline{C}\ \mathtt{in}\ N_2) = (M\ \mathtt{to}\ x_1{:}A\ \mathtt{in}\ N_1)\ \mathtt{as}\ x_2{:}U\underline{C}\ \mathtt{in}\ N_2 : \underline{D}}$$

$$\frac{\Gamma \vdash M : \underline{C} \quad \Gamma, x_1{:}U\underline{C} \vdash_{\mathsf{hom}} N_1 : FA\ \text{witnessed by}\ \overrightarrow{W_{\mathsf{op}}} \quad \Gamma \vdash \underline{D} \quad \Gamma, x_2{:}A \vdash N_2 : \underline{D}}{\Gamma \vdash M\ \mathtt{as}\ x_1{:}U\underline{C}\ \mathtt{in}\ (N_1\ \mathtt{to}\ x_2{:}A\ \mathtt{in}\ N_2) = (M\ \mathtt{as}\ x_1{:}U\underline{C}\ \mathtt{in}\ N_1)\ \mathtt{to}\ x_2{:}A\ \mathtt{in}\ N_2 : \underline{D}}$$

and analogously for computational pattern-matching, and the corresponding homomorphism terms.

Proposition 5.10. *The composition operations commute with computational pairing, computational lambda abstraction, and computational and homomorphic function applications, e.g., we have*

$$\frac{\begin{array}{c}\Gamma \vdash M : \underline{C} \quad \Gamma \vdash V : A \quad \Gamma, y{:}A \vdash \underline{D} \\ \Gamma, x{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D}[V/y]\ \text{witnessed by}\ \overrightarrow{W_{\mathsf{op}}}\end{array}}{\begin{array}{l}\Gamma \vdash M\ \mathtt{as}\ x{:}U\underline{C}\ \mathtt{in}\ \langle V, N\rangle \\ \quad = \langle V, M\ \mathtt{as}\ x{:}U\underline{C}\ \mathtt{in}\ N\rangle : \Sigma y{:}A.\underline{D}\end{array}} \qquad \frac{\begin{array}{c}\Gamma \vdash M : \underline{C} \quad \Gamma \vdash V : \underline{D}_1 \multimap \underline{D}_2 \\ \Gamma, y_1{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D}_1\ \text{witnessed by}\ \overrightarrow{W_{\mathsf{op}}}\end{array}}{\begin{array}{l}\Gamma \vdash M\ \mathtt{as}\ y_1{:}U\underline{C}\ \mathtt{in}\ V\, N \\ \quad = V\ (M\ \mathtt{as}\ y_1{:}U\underline{C}\ \mathtt{in}\ N) : \underline{D}_2\end{array}}$$

To improve the readability of the last three propositions, we have omitted the proofs of the equational proof obligations from the terms in the conclusions. These proofs are all constructed straightforwardly, by combining equational reasoning using definitional equations with the axiom of function extensionality $\mathtt{fun\text{-}ext}_{(x:A).B}(V_1, V_2, W_p)$, the transitivity and congruence rules we can derive from the elimination form for propositional equality, and the proofs given in the premises.

6 ALTERNATIVE PRESENTATIONS OF THE LANGUAGE

We now briefly discuss some alternative presentations of EMLTT and our extensions to it.

6.1 Different Definition of the Auxiliary Judgement

First, for typing the composition operations, we could have defined the auxiliary judgement

$$\Gamma, y{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D} \text{ witnessed by } \overrightarrow{W_{\mathsf{op}}}$$

not using a set of proof witnesses $\overrightarrow{W_{\mathsf{op}}}$ typed as

$$\begin{aligned}\Gamma \vdash W_{\mathsf{op}} : \lambda x{:}I.\lambda x'{:}O \to U\underline{C}.\mathtt{thunk}\,(N[\mathtt{thunk}\,(\mathsf{op}^{\underline{C}}_{x}(x''.\mathtt{force}_{\underline{C}}\,(x'\,x'')))/y]) = \\ \lambda x{:}I.\lambda x'{:}O \to U\underline{C}.\mathtt{thunk}\,(\mathsf{op}^{\underline{D}}_{x}(x''.N[x'\,x''/y]))\end{aligned}$$

but instead using a single witness W based on Munch-Maccagnoni's [2013] notion of linearity:[3]

$$\begin{aligned}\Gamma \vdash W : \lambda x{:}UFA.\lambda x'{:}A \to U\underline{C}.\mathtt{thunk}\,(N[\mathtt{thunk}\,((\mathtt{force}_{FA}\,x)\ \mathtt{to}\ x''{:}A\ \mathtt{in}_{\underline{C}}\ \mathtt{force}_{\underline{C}}\,(x'\,x''))/y]) = \\ \lambda x{:}UFA.\lambda x'{:}A \to U\underline{C}.\mathtt{thunk}\,((\mathtt{force}_{FA}\,x)\ \mathtt{to}\ x''{:}A\ \mathtt{in}_{\underline{D}}\ N[x'\,x''/y])\end{aligned}$$

The former definition of this auxiliary judgement follows from the latter by straightforward equational reasoning (taking A to be equal to O), while latter definition can be shown to follow from the former by using Plotkin and Pretnar's principle of computational induction for algebraic computational effects, which states that every computation term of type FA is either a returned value or built from computation terms using algebraic operations [Plotkin and Pretnar 2008].

While the latter definition is also applicable in languages with computational effects other than algebraic (e.g., as used by Levy [2017] to characterise general isomorphisms between computation types), we chose the former due to its more intuitive reading in the setting of algebraic effects.

6.2 Replacing Homomorphism Terms with the Auxiliary Judgement

Second, note that we could have omitted computation variables and homomorphism terms from EMLTT altogether. Instead, we could have used value variables and the auxiliary judgement $\Gamma, x{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D}$ witnessed by $\overrightarrow{W_{\mathsf{op}}}$ (see §4.2 and §6.1) to define and type the elimination form for the computational Σ-type, analogously to the composition operations introduced in §4.2. In more detail, this alternative presentation would involve the following elimination form for $\Sigma x{:}A.\underline{C}$:

$$\frac{\Gamma \vdash M : \Sigma x{:}A.\underline{C} \quad \Gamma \vdash \underline{D} \quad \Gamma, x{:}A, y{:}U\underline{C} \vdash_{\mathsf{hom}} N : \underline{D} \text{ witnessed by } \overrightarrow{W_{\mathsf{op}}}}{\Gamma \vdash M\ \mathtt{to}\ (x{:}A, y{:}U\underline{C})\ \mathtt{in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}}\ N : \underline{D}}$$

where M is now eliminated into a pair of values, with the auxiliary judgement ensuring that the computation term N behaves in the value variable y as if it was a homomorphism term. To add to this, we would also need to include specialised algebraicity equations from Prop. 5.7 in the equational theory of the language, so as to replace the general algebraicity equations from Fig. 4.

In this paper we chose to include both homomorphism terms and the above-mentioned auxiliary judgement for two reasons. First, as one of the main aims of this paper was to show how to extend the language in Ahman et al. [2016] with handlers of fibred algebraic effects, we wanted to keep the underlying language close to op. cit. Second, aesthetically, using homomorphism terms provides a cleaner presentation of the elimination form for $\Sigma x{:}A.\underline{C}$, compared to equational proof obligations.

7 USING HANDLERS TO REASON ABOUT EFFECTFUL COMPUTATIONS

In this section we demonstrate that in addition to being a practical programming abstraction, handlers also provide a useful mechanism for reasoning about effectful computations. Namely, we show that the "handle into value terms" handling construct we defined in §4.4 provides the programmer with a convenient alternative to using propositional equality on thunks for defining

[3] Note that the value variables x, x', and x'' are assigned different types in the two definitions of the auxiliary judgement.

predicates on effectful computations. Specifically, we consider two kinds of natural examples of this approach: i) lifting predicates from return values to effectful computations (§7.1); and ii) specifying patterns of allowed effects (§7.2). These examples are also accompanied by a *formalisation*[4], based on a shallow embedding of the relevant value fragment of EMLTT in Agda [The Agda Team 2017].

In order to facilitate reasoning based on the "handle into value terms" handling construct, we first introduce a universe à la Tarski [Martin-Löf 1984] by extending EMLTT with a *universe* $\mathcal{U}$ of value types, a corresponding *decoding function* $\mathsf{El}(V)$, and the corresponding *codes* of value types:

$$A ::= \ldots \mid \mathcal{U} \mid \mathsf{El}(V)$$

$$V ::= \ldots \mid \mathtt{nat\text{-}c} \mid \mathtt{one\text{-}c} \mid \mathtt{zero\text{-}c} \mid \mathtt{sum\text{-}c}(\mathtt{V},\mathtt{W}) \mid \mathtt{sig\text{-}c}(V, x.W) \mid \mathtt{pi\text{-}c}(V, x.W)$$

We also extend EMLTT with corresponding typing rules and definitional equations, e.g., we include

$$\frac{\Gamma \vdash V : \mathcal{U} \quad \Gamma, x{:}\mathsf{El}(V) \vdash W : \mathcal{U}}{\Gamma \vdash \mathtt{pi\text{-}c}(V, x.W) : \mathcal{U}} \qquad \frac{\Gamma \vdash V : \mathcal{U} \quad \Gamma, x{:}\mathsf{El}(V) \vdash W : \mathcal{U}}{\Gamma \vdash \mathsf{El}(\mathtt{pi\text{-}c}(V, x.W)) = \Pi x{:}\mathsf{El}(V).\mathsf{El}(W)}$$

Using this universe, we can now define predicates on effectful computations (of type FA) as value terms of the form $\Gamma \vdash V : UFA \to \mathcal{U}$, with the aim of using these predicates to refine (thunks of) computations using the value Σ-type, i.e., as $\Sigma x{:}UFA.\mathsf{El}(Vx)$. In detail, we define the predicates $\Gamma \vdash V : UFA \to \mathcal{U}$ by i) equipping $\mathcal{U}$ (or a type we define using it) with an algebra for the given effect theory, and ii) by using the "handle into value terms" handling construct we defined in §4.4.

It is worth noting that our approach of defining type-theoretic predicates on effectful computations by equipping the universe $\mathcal{U}$ with an algebra structure (essentially, we are defining types that depend on effectful computations in a "well-behaved" manner) is reminiscent of the recent work by Pédrot and Tabareau [2017]. In particular, their monadic translation of dependent type theories crucially relies on equipping universes with an algebra structure for a given monad.

7.1 Lifting Predicates from Return Values to Effectful Computations

Input-output. Lifting predicates from return values to computations is easiest when the given effect theory does not contain equations. Thus, let us consider the theory $\mathcal{T}_{\mathrm{I/O}}$ of *input-output of bits* from §4.1 for our first example; other equation-free fibred algebraic effects admit similar reasoning.

Assuming given a predicate $\Gamma \vdash V_P : A \to \mathcal{U}$ on A, we can lift V_P to a predicate $V_{\widehat{P}}$ on UFA by

$$V_{\widehat{P}} \stackrel{\text{def}}{=} \lambda y{:}UFA.\ (\mathtt{force}_{FA}\, y)\ \mathtt{handled\ with}\ \{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y'{:}A\ \mathtt{in}_{\mathcal{U}}\ (V_P\, y')$$

where we define the code of bits as $\mathtt{bit\text{-}c} \stackrel{\text{def}}{=} \mathtt{sum\text{-}c}(\mathtt{one\text{-}c}, \mathtt{one\text{-}c})$, and where

$$x{:}1, x'{:}1 + 1 \to \mathcal{U} \vdash V_{\mathsf{read}} \stackrel{\text{def}}{=} \mathtt{sig\text{-}c}(\mathtt{bit\text{-}c}, y'.\, x'\, y') \quad : \mathcal{U}$$

$$x{:}1 + 1, x'{:}1 \to \mathcal{U} \vdash V_{\mathsf{write}} \stackrel{\text{def}}{=} x'\, \star \quad : \mathcal{U}$$

On closer inspection, we can see that $V_{\widehat{P}}$ agrees with the *possibility modality* from Evaluation Logic [Pitts 1991], in that a computation term satisfies $V_{\widehat{P}}$ if there *exists* a return value that satisfies the given predicate V_P. Further, observe that if we were to replace `sig-c` (code for value Σ-type, i.e., existential quantification) with `pi-c` (code for value Π-type, i.e., universal quantification), we would get a *necessity modality* that holds when *all* the return values of the given computation satisfy V_P.

Global state. For our second example of lifting predicates from return values to computations, we consider an effect theory that also includes equations, the theory $\mathcal{T}_{\mathrm{GS}}$ of *global state* from §3.2.

[4]The Agda formalisation of the examples presented in §7 is available at **https://github.com/danelahman/POPL18/**

In particular, when we define the type of stores as $\mathsf{S} \stackrel{\text{def}}{=} \Pi x{:}\mathsf{Loc}.\mathsf{Val}$, then assuming given a predicate $\Gamma \vdash V_Q : A \to \mathsf{S} \to \mathcal{U}$ on return values and *final stores*, we can define a predicate

$$V_{\widehat{Q}} \stackrel{\text{def}}{=} \lambda y{:}UFA.\, \lambda x_S{:}\mathsf{S}.\, \mathtt{fst}\,\Big(\big((\mathtt{force}_{FA}\, y)\ \mathtt{handled\ with}\ (\{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathsf{GS}}}; \overrightarrow{W_{\mathsf{eq}}})\ \mathtt{to}\ y'{:}A\ \mathtt{in}_{\mathsf{S}\to\mathcal{U}\times\mathsf{S}}\ V_{\mathsf{ret}}\big)\, x_S\Big)$$

on (thunks of) computations and *initial stores*, with V_{get}, V_{put}, and V_{ret} defined as follows:

$$\begin{aligned}
\Gamma, x{:}\mathsf{Loc}, x'{:}\mathsf{Val} \to \mathsf{S} \to \mathcal{U} \times \mathsf{S} \vdash V_{\mathsf{get}} &\stackrel{\text{def}}{=} \lambda x_S{:}\mathsf{S}.\, x'\,(\mathtt{sel}\, x_S\, x)\, x_S &&: \mathsf{S} \to \mathcal{U} \times \mathsf{S}\\
\Gamma, x{:}(\Sigma x{:}\mathsf{Loc}.\mathsf{Val}), x'{:}1 \to \mathsf{S} \to \mathcal{U} \times \mathsf{S} \vdash V_{\mathsf{put}} &\stackrel{\text{def}}{=} \lambda x_S{:}\mathsf{S}.\, x' \star (\mathtt{upd}\, x_S\, x) &&: \mathsf{S} \to \mathcal{U} \times \mathsf{S}\\
\Gamma, y'{:}A \vdash V_{\mathsf{ret}} &\stackrel{\text{def}}{=} \lambda x_S{:}\mathsf{S}.\, \langle V_Q\, y'\, x_S, x_S\rangle &&: \mathsf{S} \to \mathcal{U} \times \mathsf{S}
\end{aligned}$$

and where we define the selection and update operations on stores as $\mathtt{sel}\, V_S\, V \stackrel{\text{def}}{=} V_S\, V$ and

$$\begin{aligned}
\mathtt{upd}\, V_S\, V \stackrel{\text{def}}{=}\ & \mathtt{pm}\ V\ \mathtt{as}\ (x{:}\mathsf{Loc}, y{:}\mathsf{Val})\ \mathtt{in}\\
& \lambda x'{:}\mathsf{Loc}.\, \mathtt{case}\ (\mathtt{isDec}_{\mathsf{Loc}}\, x\, x')\ \mathtt{of}\ (\mathtt{inl}(y_p{:}x =_{\mathsf{Loc}} x') \mapsto \mathtt{transport}_{\mathsf{Val}}\, x\, x'\, y_p\, y,\\
& \qquad\qquad \mathtt{inr}(y_p{:}x =_{\mathsf{Loc}} x' \to 0) \mapsto \mathtt{sel}\, V_S\, x')
\end{aligned}$$

with $\mathtt{transport}_{\mathsf{Val}} : \Pi x_l{:}\mathsf{Loc}.\Pi x'_l{:}\mathsf{Loc}.\Pi y_p{:}x_l =_{\mathsf{Loc}} x'_l.\mathsf{Val}[x_l/x] \to \mathsf{Val}[x'_l/x]$ being the standard transport operation for propositional equality that one can derive from its elimination principle.

While we omit details of the witnesses of the proof obligations corresponding to the equations given in §3.2, referring the interested reader to the Agda formalisation for details, we want to highlight that the use of $\mathtt{transport}_{\mathsf{Val}}$ in the definition of upd means that in order to construct the proof witness W_{eq} for the second of the five global state equations, namely, for $\mathsf{put}_{\langle x,y\rangle}(\mathsf{get}_x(y'.w\,(y'))) = \mathsf{put}_{\langle x,y\rangle}(w\,(y))$, we need Loc to be a *set* in the sense of Homotopy Type Theory [The Univalent Foundations Program 2013]. That is, we need there to be a pure value term

$$\diamond \vdash \mathtt{isSet}_{\mathsf{Loc}} : \Pi x{:}\mathsf{Loc}.\Pi x'{:}\mathsf{Loc}.\Pi y{:}x =_{\mathsf{Loc}} x'.\Pi y'{:}x =_{\mathsf{Loc}} x'.y =_{x=_{\mathsf{Loc}}x'} y'$$

Using Hedberg's theorem [Hedberg 1998], we can readily derive this property from $\mathtt{isDec}_{\mathsf{Loc}}$ that we assumed in §3.2. Returning to the proof witness W_{eq}, we note that its definition amounts to having to prove $\mathtt{transport}_{\mathsf{Val}}\, x\, x\, y_p\, y =_{\mathsf{Val}} y$ for an arbitrary equality proof $y_p : x =_{\mathsf{Loc}} x$, which we can do using the derived property $\mathtt{isSet}_{\mathsf{Loc}}$, i.e., using $\mathtt{isSet}_{\mathsf{Loc}}\, x\, x\, y_p\, (\mathtt{refl}\, x) : y_p =_{x=_{\mathsf{Loc}}x} \mathtt{refl}\, x$.

From the perspective of programming with computational effects, having to require Loc to have decidable equality (and thus to be a set) is not a significant drawback because the natural choice for Loc, namely, the finite coproduct of 1s denoting a finite set of memory locations, can be easily shown to have decidable equality. It is also worth noting that if Val were not dependent on Loc, then we could define the case of upd that currently uses $\mathtt{transport}_{\mathsf{Val}}$ simply as $\mathtt{inl}(y_p{:}x =_{\mathsf{Loc}} x') \mapsto y$.

Finally, on closer inspection, we can see that $V_{\widehat{Q}}$ corresponds to Dijkstra's weakest precondition semantics of stateful programs [Dijkstra 1975], e.g., the following definitional equations hold:

$$\begin{aligned}
\Gamma \vdash V_{\widehat{Q}}\ (\mathtt{thunk}\ (\mathtt{return}\ V))\ V_S &= V_Q\, V\, V_S : \mathcal{U}\\
\Gamma \vdash V_{\widehat{Q}}\ (\mathtt{thunk}\ (\mathtt{get}^{FA}_{V_l}(y.M)))\ V_S &= V_{\widehat{Q}}\ (\mathtt{thunk}\ M[\mathtt{sel}\, V_S\, V_l/y])\ V_S : \mathcal{U}\\
\Gamma \vdash V_{\widehat{Q}}\ (\mathtt{thunk}\ (\mathtt{put}^{FA}_{\langle V_l, V\rangle}(M)))\ V_S &= V_{\widehat{Q}}\ (\mathtt{thunk}\ M)\ (\mathtt{upd}\, V_S\, \langle V_l, V\rangle) : \mathcal{U}
\end{aligned}$$

That is, e.g., $V_{\widehat{Q}}$ holds of the term $\mathtt{put}^{FA}_{\langle V_l, V\rangle}(M)$ in state V_S iff it holds of M in state $\mathtt{upd}\, V_S\, \langle V_l, V\rangle$.

7.2 Specifying Patterns of Allowed Effects in Computations

Analogously to lifting predicates from return values to effectful computations, specifying patterns of allowed effects is easiest when the given fibred effect theory does not contain any equations. Thus, for simplicity, we again consider the theory $\mathcal{T}_{\mathrm{I/O}}$ of *input-output of bits* for our examples.

Disallowing writes. We begin by considering a coarse grained example of disallowing all writes:

$$V_{\text{no-w}} \stackrel{\text{def}}{=} \lambda y{:}UFA.\ (\mathtt{force}_{FA}\ y)\ \mathtt{handled\ with}\ \{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y'{:}A\ \mathtt{in}_{\mathcal{U}}\ \mathtt{one\text{-}c}$$

where V_{read} and V_{write} are defined as follows:

$$\begin{aligned} x{:}1, x'{:}1+1 \to \mathcal{U} \vdash V_{\mathsf{read}} &\stackrel{\text{def}}{=} \mathtt{pi\text{-}c}(\mathtt{bit\text{-}c}, y'.\ x'\ y') &&: \mathcal{U} \\ x{:}1+1, x'{:}1 \to \mathcal{U} \vdash V_{\mathsf{write}} &\stackrel{\text{def}}{=} \mathtt{zero\text{-}c} &&: \mathcal{U} \end{aligned}$$

For example, the computation term $\mathtt{read}^{FA}(x.\mathtt{write}^{FA}_{V}(M))$ does not satisfy $V_{\text{no-w}}$ because we have

$$\Gamma \vdash \mathsf{El}(V_{\text{no-w}}(\mathtt{thunk}(\mathtt{read}^{FA}(y.\mathtt{write}^{FA}_{V}(M))))) = \Pi y{:}1{+}1.0 \cong 0$$

Patterns of reads and writes. As a more fine grained example, we consider specifications on effectful computations in the style of session types [Honda et al. 1998], given by allowed patterns of I/O-effects. First, we assume an inductive type Protocol with the following three constructors:

$$\mathtt{r} : (1{+}1 \to \mathsf{Protocol}) \to \mathsf{Protocol} \qquad \mathtt{w} : (1{+}1 \to \mathcal{U}) \to \mathsf{Protocol} \to \mathsf{Protocol} \qquad \mathtt{e} : \mathsf{Protocol}$$

describing *patterns of allowed I/O-effects*. Intuitively, r specifies that the next allowed I/O-effect is reading; w specifies that the next allowed I/O-effect is writing, with the value written required to satisfy the predicate given as an argument to w; and e specifies that no further communication must happen (i.e., end of communication). The Protocol-valued arguments of the constructors r and w specify how the computation is allowed to evolve after reading and writing, respectively.

Then, given some particular protocol $\Gamma \vdash V_{\mathsf{pr}} : \mathsf{Protocol}$, we can define a corresponding predicate

$$V_{\widehat{\mathsf{pr}}} \stackrel{\text{def}}{=} \lambda y{:}UFA.\ \Big((\mathtt{force}_{FA}\ y)\ \mathtt{handled\ with}\ \{\mathsf{op}_x(x') \mapsto V_{\mathsf{op}}\}_{\mathsf{op}\in\mathcal{S}_{\mathrm{I/O}}}\ \mathtt{to}\ y'{:}A\ \mathtt{in}_{\mathsf{Protocol}\to\mathcal{U}}\ V_{\mathsf{ret}}\Big)\ V_{\mathsf{pr}}$$

where the value terms V_{read}, V_{write}, and V_{ret} are defined as follows (for better readability, we give their structural-recursive definitions by pattern-matching on their arguments of type Protocol):

$$\begin{aligned} \Gamma, x{:}1, x'{:}1+1 \to \mathsf{Protocol} \to \mathcal{U} \vdash V_{\mathsf{read}}\ (\mathtt{r}\ V'_{\mathsf{pr}}) &\stackrel{\text{def}}{=} \mathtt{pi\text{-}c}(\mathtt{bit\text{-}c}, y'.(x'\ y')\ (V'_{\mathsf{pr}}\ y')) &&: \mathcal{U} \\ \Gamma, x{:}1+1, x'{:}1 \to \mathsf{Protocol} \to \mathcal{U} \vdash V_{\mathsf{write}}\ (\mathtt{w}\ V_P\ V'_{\mathsf{pr}}) &\stackrel{\text{def}}{=} \mathtt{sig\text{-}c}(V_P\ x, y'.\ x' \star V'_{\mathsf{pr}}) &&: \mathcal{U} \\ \Gamma, y'{:}A \vdash V_{\mathsf{ret}}\quad \mathtt{e} &\stackrel{\text{def}}{=} \mathtt{one\text{-}c} &&: \mathcal{U} \end{aligned}$$

with all other cases defined as zero-c. As a result, a computation satisfies the predicate $V_{\widehat{\mathsf{pr}}}$ if and only if its I/O-effects precisely follow the specific pattern of I/O-effects allowed by the protocol V_{pr}.

This example can be easily extended to also account for *sets* of patterns of allowed I/O-effects. For instance, we could extend the inductive type Protocol with a fourth constructor or, typed as $\mathtt{or} : \mathsf{Protocol} \to \mathsf{Protocol} \to \mathsf{Protocol}$, and correspondingly extend the above definitions of value terms $V \in \{V_{\mathsf{read}}, V_{\mathsf{write}}, V_{\mathsf{ret}}\}$ each with a new case given by $V\ (V'_{\mathsf{pr}}\ \mathtt{or}\ V''_{\mathsf{pr}}) \stackrel{\text{def}}{=} \mathtt{sum\text{-}c}(V\ V'_{\mathsf{pr}}, V\ V''_{\mathsf{pr}})$.

Finally, we highlight that it is easy combine these specifications with those discussed in §7.1, namely, by replacing one-c in the definition of V_{ret} with a suitable predicate V_P on return values.

8 SEMANTICS

We conclude by describing how to give a natural denotational semantics to our extension of EMLTT with fibred algebraic effects and their handlers. This semantics is an instance of a more general class of models, based on *fibrations* (functors with extra structure for modelling substitution, Σ- and Π-types, etc.) and *adjunctions* between them, as studied by Ahman [2017]; Ahman et al. [2016].

We proceed in three steps. First, we recall how the pure fragment of EMLTT is interpreted in the families of sets fibration, a prototypical model of dependent types. Next, we show how to derive a

countable Lawvere theory $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ from the given fibred effect theory $\mathcal{T}_{\text{eff}}$. Finally, we show how to define the interpretation of the rest of our extension of EMLTT using families of models of $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$.

8.1 Families Fibrations

We begin by giving a brief overview of the kinds of fibrations we use for defining the denotational semantics of our extension of EMLTT. For a more detailed treatment of fibrations and their use in modelling various type theories and logics, we refer the reader to the book by Jacobs [1999].

Given a category C, it is well-known that one can define a new category $\mathsf{Fam}(C)$ of C-valued families. Its objects are pairs (X, A) of a set X and a functor $A : X \longrightarrow C$ (treating X as a discrete category); and the morphisms $(X, A) \to (Y, B)$ are pairs of a function $f : X \longrightarrow Y$ and a natural transformation $g : A \longrightarrow B \circ f$. The corresponding C-valued *families fibration* $\mathsf{fam}_C : \mathsf{Fam}(C) \longrightarrow \mathsf{Set}$ is then defined on objects as $\mathsf{fam}_C(X, A) \stackrel{\text{def}}{=} X$ and on morphisms as $\mathsf{fam}_C(f, g) \stackrel{\text{def}}{=} f$.

Next, for any set X, the category $\mathsf{Fam}_X(C)$ is called the *fibre* over X; this is a subcategory of $\mathsf{Fam}(C)$ whose objects and morphisms are of the form (X, A) and (id_X, g). Given a function $f : X \longrightarrow Y$, the corresponding *reindexing functor* $f^* : \mathsf{Fam}_Y(C) \longrightarrow \mathsf{Fam}_X(C)$ is given on objects by $f^*(Y, A) \stackrel{\text{def}}{=} (X, A \circ f)$, and analogously on morphisms. As standard in the literature, we write $\overline{f}(Y, A) \stackrel{\text{def}}{=} (f, (\mathsf{id}_{A(f(x))})_x) : f^*(Y, A) \longrightarrow (Y, A)$ for the *Cartesian morphism* over $f : X \longrightarrow Y$.

We get a prototypical model of dependent types when we take $C \stackrel{\text{def}}{=} \mathsf{Set}$, the category of sets and functions. In this case, there also exists a pair of adjunctions $\mathsf{fam}_{\mathsf{Set}} \dashv 1 \dashv \{-\}$, where the *terminal object functor* $1 : \mathsf{Set} \longrightarrow \mathsf{Fam}(\mathsf{Set})$ is given by $1(X) \stackrel{\text{def}}{=} (X, x \mapsto \{\star\})$, and the *comprehension functor* $\{-\} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$ is given by $\{(X, A)\} \stackrel{\text{def}}{=} \coprod_{x \in X} A(x)$. The latter functor provides a natural semantics for context extension, together with the canonical *projection maps* $\pi_{(X,A)} : \{(X, A)\} \longrightarrow X$.

8.2 Interpretation of the Pure Fragment of EMLTT

We now recall how the pure fragment of EMLTT (i.e., MLTT) is interpreted in the families of sets fibration $\mathsf{fam}_{\mathsf{Set}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$. For more details about the interpretation of this fragment of EMLTT, we refer the reader to the relevant sections in Ahman [2017]; Ahman et al. [2016], and to Streicher [1991] whose treatment of the categorical semantics of MLTT the former two build on.

In detail, the interpretation of the pure fragment of EMLTT is defined as a *partial interpretation function* $[\![-]\!]$, which, if defined, maps a context Γ to a set $[\![\Gamma]\!]$; a context Γ and value type A to an object $[\![\Gamma; A]\!]$ in $\mathsf{Fam}_{[\![\Gamma]\!]}(\mathsf{Set})$; and a context Γ and value term V to $[\![\Gamma; V]\!] : 1([\![\Gamma]\!]) \longrightarrow ([\![\Gamma]\!], A)$ in $\mathsf{Fam}_{[\![\Gamma]\!]}(\mathsf{Set})$, for some $A : [\![\Gamma]\!] \longrightarrow \mathsf{Set}$. For better readability, we denote the first and second components of $[\![\Gamma; A]\!]$ and $[\![\Gamma; V]\!]$ using subscripts 1, 2, i.e., we write $([\![\Gamma; A]\!]_1, [\![\Gamma; A]\!]_2)$ for $[\![\Gamma; A]\!]$.

First, the value types Nat, 1, 0, and $A + B$ are interpreted using the corresponding categorical structure in the fibres of $\mathsf{Fam}(\mathsf{Set})$ as follows, assuming that $[\![\Gamma]\!]$, $[\![\Gamma; A]\!]$, and $[\![\Gamma; B]\!]$ are defined:

$$[\![\Gamma; \mathsf{Nat}]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \mathbb{N}) \qquad [\![\Gamma; 1]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \{\star\}) = 1([\![\Gamma]\!]) \qquad [\![\Gamma; 0]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \emptyset)$$

$$[\![\Gamma; A + B]\!] \stackrel{\text{def}}{=} [\![\Gamma; A]\!] + [\![\Gamma; B]\!] = ([\![\Gamma]\!], \gamma \mapsto [\![\Gamma; A]\!]_2(\gamma) + [\![\Gamma; B]\!]_2(\gamma))$$

Next, assuming that $[\![\Gamma; A]\!]$ and $[\![\Gamma, x : A; B]\!]$ are defined, with $[\![\Gamma, x : A; B]\!]_1 = \coprod_{\gamma \in [\![\Gamma]\!]} [\![\Gamma; A]\!]_2(\gamma)$, then

$$[\![\Gamma; \Sigma x : A.B]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \coprod_{a \in [\![\Gamma; A]\!]_2(\gamma)} [\![\Gamma, x : A; B]\!]_2(\langle \gamma, a \rangle))$$

$$[\![\Gamma; \Pi x : A.B]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \prod_{a \in [\![\Gamma; A]\!]_2(\gamma)} [\![\Gamma, x : A; B]\!]_2(\langle \gamma, a \rangle))$$

Finally, propositional equality is interpreted as the equality between the interpretations of terms:

$$[\![\Gamma; V =_A W]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \{ \star \mid ([\![\Gamma; V]\!]_2)_\gamma(\star) = ([\![\Gamma; W]\!]_2)_\gamma(\star) \})$$

assuming that $[\![\Gamma; V]\!] : 1([\![\Gamma]\!]) \longrightarrow ([\![\Gamma; A]\!])$ and $[\![\Gamma; W]\!] : 1([\![\Gamma]\!]) \longrightarrow ([\![\Gamma; A]\!])$ are both defined. In the rest of this paper, we often leave such routine preconditions to the definedness of $[\![-]\!]$ implicit.

In §8.5, we rely on this extensional nature of the semantics of propositional equality for showing that the interpretations of the proof witnesses involved in the typing of the user-defined algebra type and the composition operations are enough to validate the corresponding definitional equations.

Next, the interpretation of the universe $\mathcal{U}$ is based on the semantics of induction-recursion [Dybjer and Setzer 1999; Ghani et al. 2013]: we first construct the initial algebra $\mu\Phi$ of an endofunctor Φ on $\mathsf{Fam}(|\mathsf{Set}|)$[5] corresponding to the signature of codes of value types given in §7, and then define

$$[\![\Gamma; \mathcal{U}]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \mu\Phi_1) \qquad [\![\Gamma; \mathsf{El}(V)]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \mu\Phi_2\,([\![\Gamma; V]\!]_2)_\gamma(\star)))$$

Finally, as an example of the interpretation of value terms, $\mathtt{inl}_{A+B}\ V$ is interpreted as follows:

$$[\![\Gamma; \mathtt{inl}_{A+B}\ V]\!]_1 \stackrel{\text{def}}{=} \mathsf{id}_{[\![\Gamma]\!]} \qquad ([\![\Gamma; \mathtt{inl}_{A+B}\ V]\!]_2)_\gamma \stackrel{\text{def}}{=} \mathsf{inl}_{[\![\Gamma;A]\!]_2(\gamma)+[\![\Gamma;B]\!]_2(\gamma)} \circ ([\![\Gamma; V]\!]_2)_\gamma$$

The soundness theorems for MLTT and EMLTT then tell us that the *a priori* partial $[\![-]\!]$ is in fact defined on well-formed pure syntax and that it validates the corresponding definitional equations.

8.3 Deriving a Countable Lawvere Theory from a Fibred Effect Theory

Next, we show how to derive a countable Lawvere theory from the given fibred effect theory $\mathcal{T}_{\text{eff}}$.

We begin by recalling some basic definitions and results about countable Lawvere theories and their models. We refer the reader to Power [2006] for a more detailed overview of the area.

A *countable Lawvere theory* consists of a small category $\mathcal{L}$ with countable products and a strict countable-product preserving identity-on-objects functor $I : \aleph_1^{\text{op}} \longrightarrow \mathcal{L}$, where $\aleph_1$ is the skeleton of the category of countable sets and all functions between them. In other words, an object of $\aleph_1$ is either a natural number or the distinguished element ω denoting the cardinality of countable sets.

A *model* of a countable Lawvere theory $\mathcal{L}$ in a category C with countable products is a countable-product preserving functor $\mathcal{M} : \mathcal{L} \longrightarrow C$. A *morphism* from $\mathcal{M}_1 : \mathcal{L} \longrightarrow C$ to $\mathcal{M}_2 : \mathcal{L} \longrightarrow C$ is given by a natural transformation $h : \mathcal{M}_1 \longrightarrow \mathcal{M}_2$. This gives us the category $\mathsf{Mod}(\mathcal{L}, C)$.

Conceptually, a countable Lawvere theory is nothing but an abstract category-theoretic description of the clone of countable equational theories [Grätzer 1979]. In particular, one often thinks of the morphisms $n \longrightarrow 1$ in $\mathcal{L}$ as terms in n free variables, and of the morphisms $n \longrightarrow m$ as m-tuples of terms in n free variables. Analogously, the models of a countable Lawvere theory correspond to the models of the countable equational theories whose clone this countable Lawvere theory is.

We also recall that there exists a canonical forgetful functor $U_{\mathcal{L}} : \mathsf{Mod}(\mathcal{L}, C) \longrightarrow C$, given on objects by $U_{\mathcal{L}}(\mathcal{M}) \stackrel{\text{def}}{=} \mathcal{M}(1)$. It is then well-known that if C is locally countably presentable [Adamek and Rosicky 1994], the functor $U_{\mathcal{L}}$ has a left adjoint $F_{\mathcal{L}} : C \longrightarrow \mathsf{Mod}(\mathcal{L}, C)$. Importantly for the purposes of this paper, the category Set of sets and functions is locally countably presentable.

Next, in order to be able to derive a countable Lawvere theory $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ from the given fibred effect theory $\mathcal{T}_{\text{eff}}$, we require $\mathcal{T}_{\text{eff}}$ to be *countable*. In particular, this means that for all $\mathsf{op} : (x : I) \longrightarrow O \in \mathcal{S}_{\text{eff}}$, we require $[\![x : I; O]\!]_2$ to be a family of countable sets. Furthermore, we also require $[\![\Gamma; A'_j]\!]_2$ to be a family of countable sets, for all equations $\Gamma \mid \Delta \vdash T_1 = T_2 \in \mathcal{E}_{\text{eff}}$ and effect variables $w_j : A'_j \in \Delta$.

We can then construct $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ by expanding $\mathcal{T}_{\text{eff}}$ into a countable equational theory [Grätzer 1979], analogously to how Plotkin and Pretnar [2013] expanded their effect theories. More specifically, $\mathcal{S}_{\text{eff}}$ determines a countable signature consisting of operation symbols $\mathsf{op}_i : |[\![x : I; O]\!]_2\,(\langle\star, i\rangle)|$, for all $\mathsf{op} : (x : I) \longrightarrow O \in \mathcal{S}_{\text{eff}}$ and $i \in [\![\diamond; I]\!]_2\,(\star)$. Every effect term $\Gamma \mid \Delta \vdash T$ then naturally determines terms $\Delta^\gamma \vdash T^\gamma$ derivable from this countable signature (for all $\gamma \in [\![\Gamma]\!]$), where Δ^γ consists of

[5] $|\mathsf{Set}|$ denotes the discrete variant of Set; it is used to accommodate the contravariance arising from decoding the Π-type.

variables $x^a_{w_j}$ for all $w_j : A_j \in \Delta$ and $a \in [\![\Gamma; A'_j]\!]_2(\gamma)$. In detail, the terms T^γ are defined as follows:

$$
\begin{array}{lll}
(w_j(V))^\gamma & \stackrel{\text{def}}{=} x^{([\![\Gamma;V]\!]_2)_\gamma(\star)}_{w_j} & \\
(\mathsf{op}_V(y.T))^\gamma & \stackrel{\text{def}}{=} \mathsf{op}_{([\![\Gamma;V]\!]_2)_\gamma(\star)}(T^{\langle\gamma,o\rangle})_{1\le o\le |[\![x:I;O]\!]_2\,(\langle\star,([\![\Gamma;V]\!]_2)_\gamma(\star)\rangle)|} & \\
(\mathsf{pm}\ V\ \mathsf{as}\ (y_1:B_1, y_2:B_2)\ \mathsf{in}\ T)^\gamma & \stackrel{\text{def}}{=} T^{\langle\langle\gamma,b_1\rangle,b_2\rangle} & (\textit{when}\ ([\![\Gamma;V]\!]_2)_\gamma(\star) = \langle b_1, b_2\rangle) \\
(\mathsf{case}\ V\ \mathsf{of}\ (\mathsf{inl}(y_1)\mapsto T_1, \mathsf{inr}(y_2)\mapsto T_2))^\gamma & \stackrel{\text{def}}{=} T_1^{\langle\gamma,b\rangle} & (\textit{when}\ ([\![\Gamma;V]\!]_2)_\gamma(\star) = \mathsf{inl}\, b) \\
(\mathsf{case}\ V\ \mathsf{of}\ (\mathsf{inl}(y_1)\mapsto T_1, \mathsf{inr}(y_2)\mapsto T_2))^\gamma & \stackrel{\text{def}}{=} T_2^{\langle\gamma,b\rangle} & (\textit{when}\ ([\![\Gamma;V]\!]_2)_\gamma(\star) = \mathsf{inr}\, b)
\end{array}
$$

We get a countable equational theory by taking equations $\Delta^\gamma \vdash T_1^\gamma = T_2^\gamma$, for all $\Gamma \mid \Delta \vdash T_1 = T_2 \in \mathcal{E}_{\text{eff}}$ and $\gamma \in [\![\Gamma]\!]$, and closing them under the standard rules of reflexivity, symmetry, transitivity, etc.

The countable Lawvere theory $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ is then given by taking the morphisms $n \longrightarrow m$ in $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ to be m-tuples $(\overrightarrow{x_i} \vdash t_j)_{1\le j\le m}$ of equivalence classes of terms in n variables (in the countable equational theory defined above). The identity morphisms are given by tuples of variables, while the composition of morphisms is given by substitution. We define the functor $I_{\mathcal{T}_{\text{eff}}} : \aleph_1^{\text{op}} \longrightarrow \mathcal{L}_{\mathcal{T}_{\text{eff}}}$ by $I_{\mathcal{T}_{\text{eff}}}(n) \stackrel{\text{def}}{=} n$ and $I_{\mathcal{T}_{\text{eff}}}(f) \stackrel{\text{def}}{=} (\overrightarrow{x_i} \vdash x_{f(j)})_{1\le j\le m} : n \to m$. It is then easy to verify both that $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ has countable products (given using the cardinal sum in $\aleph_1$) and that $I_{\mathcal{T}_{\text{eff}}}$ strictly preserves them.

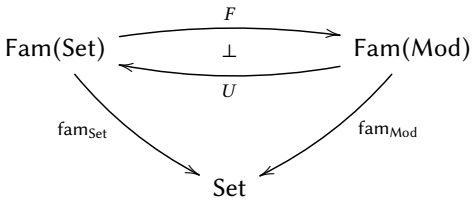
Proposition 8.1. *$\mathcal{L}_{\mathcal{T}_{\text{eff}}}$ is a countable Lawvere theory.*

For better readability, we write Mod for $\mathsf{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \mathsf{Set})$ in the rest of this paper. In addition, a useful well-known property of Mod that we use later is that it is both complete and cocomplete.

We conclude by highlighting that for any set A, one can intuitively think of the *free model* $F_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}}(A)$ as being given by the set of equivalence classes of computation trees built from the operations of the given fibred effect theory $\mathcal{T}_{\text{eff}}$, with the leaves of these trees given by elements of the set A.

8.4 Interpretation of the Non-Pure Fragment of EMLTT

Next, we show how to extend the interpretation of the pure fragment of EMLTT to the rest of our extension of EMLTT with fibred algebraic effects and their handlers, based on the fibred adjunction



where the two fibred functors F and U are defined (on objects) as

$$F(X, A) \stackrel{\text{def}}{=} (X, F_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}} \circ A) \qquad U(X, \underline{C}) \stackrel{\text{def}}{=} (X, U_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}} \circ \underline{C})$$

That the functors F and U indeed form a fibred adjunction suitable for modelling EMLTT is an instance of a more general result about the models of EMLTT—see Ahman et al. [2016, Thm. 3].

Using this fibred adjunction, we can now extend the definition of $[\![-]\!]$ that we gave in §8.2 so that, if defined, it maps a context Γ and a computation type $\underline{C}$ to an object $[\![\Gamma; \underline{C}]\!]$ in $\mathsf{Fam}_{[\![\Gamma]\!]}(\mathsf{Mod})$; a context Γ and a computation term M to a morphism $[\![\Gamma; M]\!] : 1([\![\Gamma]\!]) \longrightarrow U([\![\Gamma]\!], \underline{C})$ in $\mathsf{Fam}_{[\![\Gamma]\!]}(\mathsf{Set})$, for some $\underline{C} : [\![\Gamma]\!] \longrightarrow \mathsf{Mod}$; and a context Γ, a variable z, a computation type $\underline{C}$, and a homomorphism term K to a morphism $[\![\Gamma; z:\underline{C}; K]\!] : [\![\Gamma; \underline{C}]\!] \longrightarrow ([\![\Gamma]\!], \underline{D})$ in $\mathsf{Fam}_{[\![\Gamma]\!]}(\mathsf{Mod})$, for some $\underline{D} : [\![\Gamma]\!] \longrightarrow \mathsf{Mod}$.

In particular, the interpretation of the types $\underline{C} \multimap \underline{D}$, $U\underline{C}$, and FA is defined as follows:

$$[\![\Gamma; \underline{C} \multimap \underline{D}]\!] \stackrel{\text{def}}{=} \Big([\![\Gamma]\!], \gamma \mapsto \mathsf{Hom}_{\mathsf{Mod}}\big([\![\Gamma; \underline{C}]\!]_2(\gamma), [\![\Gamma; \underline{D}]\!]_2(\gamma)\big)\Big)$$

$$[\![\Gamma; U\underline{C}]\!] \stackrel{\text{def}}{=} U([\![\Gamma; \underline{C}]\!]) \qquad [\![\Gamma; FA]\!] \stackrel{\text{def}}{=} F([\![\Gamma; A]\!])$$

The computational Π- and Σ-types are interpreted similarly to their value counterparts, using the set-indexed products and coproducts in Mod, which exist because Mod is complete and cocomplete.

We omit the definition of $[\![-]\!]$ for the cases (of computation and homomorphism terms) that are already covered in great detail in Ahman et al. [2016], and instead concentrate on demonstrating how to define the interpretation function $[\![-]\!]$ for algebraic operations, the user-defined algebra type, and the two composition operations. Diagrammatic and more detailed treatment of these cases of the definition of $[\![-]\!]$ can be found in §6 and §7 of the author's PhD thesis [Ahman 2017].

First, we define $[\![-]\!]$ on algebraic operations $\mathsf{op}^{\underline{C}}_{V}(y.M)$ as follows:

$$([\![\Gamma; \mathsf{op}^{\underline{C}}_{V}(y.M)]\!]_2)_\gamma \stackrel{\text{def}}{=} \mathsf{op}^{[\![\Gamma;C]\!]_2(\gamma)}_{([\![\Gamma;V]\!]_2)_\gamma(\star)} \circ \prod_o \Big(([\![\Gamma, y{:}O[V/x]; M]\!]_2)_{\langle\gamma,o\rangle}\Big) \circ \langle \mathsf{id}_1 \rangle_o : 1 \longrightarrow U_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}}([\![\Gamma; \underline{C}]\!]_2(\gamma))$$

where $\mathsf{op}^{[\![\Gamma;C]\!]_2(\gamma)}_{([\![\Gamma;V]\!]_2)_\gamma(\star)}$ is the corresponding *operation* of $[\![\Gamma; \underline{C}]\!]_2(\gamma)$, given by the following composite:

$$([\![\Gamma; \underline{C}]\!]_2(\gamma))(\overrightarrow{x_o} \vdash \mathsf{op}_{([\![\Gamma;V]\!]_2)_\gamma(\star)}(x_o)_o) \circ \iota : \prod_{o \in [\![\Gamma;O[V/x]]\!]_2(\gamma)} ([\![\Gamma; \underline{C}]\!]_2(\gamma))(1) \longrightarrow ([\![\Gamma; \underline{C}]\!]_2(\gamma))(1)$$

where ι is the canonical countable-product preservation isomorphism

$$\prod_{o \in [\![\Gamma;O[V/x]]\!]_2(\gamma)} ([\![\Gamma; \underline{C}]\!]_2(\gamma))(1) \cong ([\![\Gamma; \underline{C}]\!]_2(\gamma))(|[\![\Gamma; O[V/x]]\!]_2(\gamma)|)$$

Next, we define $[\![-]\!]$ on the user-defined algebra type $\langle A, \{V_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{S}_{\text{eff}}}\rangle$ as follows:

$$[\![\Gamma; \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}}\rangle]\!] \stackrel{\text{def}}{=} ([\![\Gamma]\!], \gamma \mapsto \mathcal{M}^\gamma)$$

where the functors $\mathcal{M}^\gamma : \mathcal{L}_{\mathcal{T}_{\text{eff}}} \longrightarrow \mathsf{Set}$ are defined on morphisms of the form $n \longrightarrow 1$ as follows:[6]

$$\mathcal{M}^\gamma(n) \stackrel{\text{def}}{=} \prod_{1 \le j \le n} [\![\Gamma; A]\!]_2(\gamma) \qquad \mathcal{M}^\gamma(\overrightarrow{x_j} \vdash x_j) \stackrel{\text{def}}{=} \mathsf{proj}_j$$

$$\mathcal{M}^\gamma(\Delta \vdash \mathsf{op}_i(t_o)_{1 \le o \le |[\![x:I;O]\!]_2(\langle\star,i\rangle)|}) \stackrel{\text{def}}{=} f^\gamma_{\mathsf{op}_i} \circ \langle \mathcal{M}^\gamma(\Delta \vdash t_o)\rangle_{o \in [\![x:I;O]\!]_2(\langle\star,i\rangle)}$$

where the function $f^\gamma_{\mathsf{op}_i}$ is derived from $[\![\Gamma; V_{\mathsf{op}}]\!]$ as follows:

$$f^\gamma_{\mathsf{op}_i} \stackrel{\text{def}}{=} f \mapsto \mathsf{proj}_{\langle i,f\rangle}\Big(([\![\Gamma; V_{\mathsf{op}}]\!]_2)_\gamma(\star)\Big) : \prod_{o \in [\![x:I;O]\!]_2(\langle\star,i\rangle)} [\![\Gamma; A]\!]_2(\gamma) \longrightarrow [\![\Gamma; A]\!]_2(\gamma)$$

It is worth noting that each $\mathcal{M}^\gamma$ extends straightforwardly to m-tuples of terms, so as to account for all morphisms in $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$, i.e., those of the form $n \longrightarrow m$ for a general m. Specifically, we have that

$$\mathcal{M}^\gamma((\Delta \vdash t)_{1 \le j \le m}) = \langle \mathcal{M}^\gamma(\Delta \vdash t)\rangle_{1 \le j \le m}$$

We also note that for $[\![\Gamma; \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}}\rangle]\!]$ to be defined, we additionally require that $\mathcal{M}^\gamma$ validates the equations given in $\mathcal{E}_{\text{eff}}$. Namely, we require for all $\Gamma' \mid \Delta \vdash T_1 = T_2 \in \mathcal{E}_{\text{eff}}$ and $\gamma' \in [\![\Gamma']\!]$ that

$$\mathcal{M}^\gamma(\Delta^{\gamma'} \vdash T_1^{\gamma'}) = \mathcal{M}^\gamma(\Delta^{\gamma'} \vdash T_2^{\gamma'})$$

Finally, we define $[\![-]\!]$ on the two composition operations as follows:

$$([\![\Gamma; M \text{ as } x{:}U\underline{C} \text{ in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}} N]\!]_2)_\gamma \stackrel{\text{def}}{=} f^\gamma \circ ([\![\Gamma; M]\!]_2)_\gamma : 1 \longrightarrow U_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}}([\![\Gamma; \underline{D}]\!]_2(\gamma))$$

$$([\![\Gamma; z{:}\underline{C}'; K \text{ as } x{:}U\underline{C} \text{ in}_{\overrightarrow{W_{\mathsf{op}}};\underline{D}} N]\!]_2)_\gamma \stackrel{\text{def}}{=} \mathsf{hom}(f^\gamma) \circ ([\![\Gamma; z{:}\underline{C}'; K]\!]_2)_\gamma : [\![\Gamma; \underline{C}']\!]_2(\gamma) \longrightarrow [\![\Gamma; \underline{D}]\!]_2(\gamma)$$

[6] We use the notation $\prod_{1 \le j \le n} [\![\Gamma; A]\!]_2(\gamma)$ to mean the finite product $[\![\Gamma; A]\!]_2(\gamma) \times \ldots \times [\![\Gamma; A]\!]_2(\gamma)$ when n is a natural number, and $\prod_{m \in \mathbb{N}} [\![\Gamma; A]\!]_2(\gamma)$ when n is the distinguished symbol ω. In particular, we have $\mathcal{M}^\gamma(1) = [\![\Gamma; A]\!]_2(\gamma)$.

where the function f^γ is derived from $[\![\Gamma, x:U\underline{C}; N]\!]$ as follows:

$$f^\gamma \stackrel{\text{def}}{=} c \mapsto ([\![\Gamma, x:U\underline{C}; N]\!]_2)_{\langle\gamma, c\rangle}(\star) : U_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}}([\![\Gamma; \underline{C}]\!]_2(\gamma)) \longrightarrow U_{\mathcal{L}_{\mathcal{T}_{\text{eff}}}}([\![\Gamma; \underline{D}]\!]_2(\gamma))$$

and where $\mathsf{hom}(f^\gamma)$ is a morphism of models of $\mathcal{L}_{\mathcal{T}_{\text{eff}}}$, given by components

$$(\mathsf{hom}(f^\gamma))_n \stackrel{\text{def}}{=} \iota_{[\![\Gamma;\underline{D}]\!]_2(\gamma)} \circ \textstyle\prod_{1\leq j\leq n}(f^\gamma) \circ \iota^{-1}_{[\![\Gamma;\underline{C}]\!]_2(\gamma)} : ([\![\Gamma; \underline{C}]\!]_2(\gamma))(n) \longrightarrow ([\![\Gamma; \underline{D}]\!]_2(\gamma))(n)$$

where $\iota_{[\![\Gamma;\underline{C}]\!]_2(\gamma)}$ and $\iota_{[\![\Gamma;\underline{D}]\!]_2(\gamma)}$ are the canonical countable-product preservation isomorphisms

$$\textstyle\prod_{1\leq j\leq n}([\![\Gamma; \underline{C}]\!]_2(\gamma))(1) \cong ([\![\Gamma; \underline{C}]\!]_2(\gamma))(n) \qquad \prod_{1\leq j\leq n}([\![\Gamma; \underline{D}]\!]_2(\gamma))(1) \cong ([\![\Gamma; \underline{D}]\!]_2(\gamma))(n)$$

We note that for these two cases of $[\![-]\!]$ to be defined, we require that f^γ commutes with the operations of $[\![\Gamma; \underline{C}]\!]_2(\gamma)$ and $[\![\Gamma; \underline{D}]\!]_2(\gamma)$, for all $i \in [\![\diamond; I]\!]_2(\star)$, as depicted in the following diagram:

$$\begin{array}{ccc} \prod_o([\![\Gamma; \underline{C}]\!]_2(\gamma))(1) & \xrightarrow{\prod_{o\in[\![x:I;O]\!]_2(\langle\star,i\rangle)}(f^\gamma)} & \prod_o([\![\Gamma; \underline{D}]\!]_2(\gamma))(1) \\ \Big\downarrow{\scriptstyle \mathsf{op}_i^{[\![\Gamma;C]\!]_2(\gamma)}} & & \Big\downarrow{\scriptstyle \mathsf{op}_i^{[\![\Gamma;D]\!]_2(\gamma)}} \\ ([\![\Gamma; \underline{C}]\!]_2(\gamma))(1) & \xrightarrow[f^\gamma]{} & ([\![\Gamma; \underline{D}]\!]_2(\gamma))(1) \end{array}$$

8.5 Soundness

In order to establish the soundness of the interpretation $[\![-]\!]$ we defined above in §8.2 and §8.4, we first formulate and prove standard semantic weakening and substitution results. In particular, we begin by defining standard *a priori* partial semantic *projection* and *substitution* morphisms

$$\mathsf{proj}_{\Gamma_1;x:A;\Gamma_2} : [\![\Gamma_1, x:A, \Gamma_2]\!] \longrightarrow [\![\Gamma_1, \Gamma_2]\!] \qquad \mathsf{subst}_{\Gamma_1;x:A;\Gamma_2;V} : [\![\Gamma_1, \Gamma_2[V/x]]\!] \longrightarrow [\![\Gamma_1, x:A, \Gamma_2]\!]$$

by induction on the size of Γ_2 as follows:

$$\begin{array}{lll} \mathsf{proj}_{\Gamma_1;x:A;\diamond} \stackrel{\text{def}}{=} \pi_{[\![\Gamma_1;A]\!]} & \mathsf{proj}_{\Gamma_1;x:A;\Gamma_2,y:B} \stackrel{\text{def}}{=} \{\overline{\mathsf{proj}_{\Gamma_1;x:A;\Gamma_2}}([\![\Gamma_1, \Gamma_2; B]\!])\} \\ \mathsf{subst}_{\Gamma_1;x:A;\diamond;V} \stackrel{\text{def}}{=} [\![\Gamma; V]\!] & \mathsf{subst}_{\Gamma_1;x:A;\Gamma_2,y:B;V} \stackrel{\text{def}}{=} \{\overline{\mathsf{subst}_{\Gamma_1;x:A;\Gamma_2;V}}([\![\Gamma_1, x:A, \Gamma_2; B]\!])\} \end{array}$$

Next, we show that both morphisms are defined if the interpretations of the involved contexts, types, and terms are defined; and that reindexing along them models weakening and substitution. These results also extend to account for weakening and substitution with multiple value variables.

Proposition 8.2. *Given value contexts Γ_1 and Γ_2, a value type A, and a value variable x such that $[\![\Gamma_1, \Gamma_2]\!] \in \mathsf{Set}$ and $[\![\Gamma_1, x:A, \Gamma_2]\!] \in \mathsf{Set}$, then i) the semantic projection morphism $\mathsf{proj}_{\Gamma_1;x:A;\Gamma_2}$ is defined; and ii) given a value type B such that $[\![\Gamma_1, \Gamma_2; B]\!] \in \mathsf{Fam}_{[\![\Gamma_1,\Gamma_2]\!]}(\mathsf{Set})$, we have*

$$[\![\Gamma_1, x:A, \Gamma_2; B]\!] = \mathsf{proj}^*_{\Gamma_1;x:A;\Gamma_2}([\![\Gamma_1, \Gamma_2; B]\!])$$

and similarly for computation types, and value, computation, and homomorphism terms.

Proposition 8.3. *Given value contexts Γ_1 and Γ_2, a value type A, a value variable x, and a value term V such that $[\![\Gamma_1; V]\!] : 1([\![\Gamma_1]\!]) \longrightarrow [\![\Gamma_1; A]\!]$, $[\![\Gamma_1, x:A, \Gamma_2]\!] \in \mathsf{Set}$, and $[\![\Gamma_1, \Gamma_2[V/x]]\!] \in \mathsf{Set}$, then i) the semantic substitution morphism $\mathsf{subst}_{\Gamma_1;x:A;\Gamma_2;V}$ is defined; and ii) given a value type B such that $[\![\Gamma_1, x:A, \Gamma_2; B]\!] \in \mathsf{Fam}_{[\![\Gamma_1, x:A, \Gamma_2]\!]}(\mathsf{Set})$, we have*

$$[\![\Gamma_1, \Gamma_2[V/x]; B[V/x]]\!] = \mathsf{subst}^*_{\Gamma_1;x:A;\Gamma_2;V}([\![\Gamma_1, x:A, \Gamma_2; B]\!])$$

and similarly for computation types, and value, computation, and homomorphism terms.

In addition, we show that substituting computation and homomorphism terms for computation variables corresponds to the composition of the morphisms that the given terms denote.

Proposition 8.4. *Given a value context* Γ*, a computation variable* z*, a computation type* $\underline{C}$*, a computation term* M*, and a homomorphism term* K *such that* $[\![\Gamma; M]\!] : 1([\![\Gamma]\!]) \longrightarrow U([\![\Gamma; \underline{C}]\!])$ *and* $[\![\Gamma; z{:}\underline{C}; K]\!] : [\![\Gamma; \underline{C}]\!] \longrightarrow ([\![\Gamma]\!], \underline{D})$*, for some* $\underline{D} : [\![\Gamma]\!] \longrightarrow \mathrm{Mod}$*, then*

$$[\![\Gamma; K[M/z]]\!] = U([\![\Gamma; z{:}\underline{C}; K]\!]) \circ [\![\Gamma; M]\!]$$

Proposition 8.5. *Given a value context* Γ*, computation variables* z_1 *and* z_2*, computation types* $\underline{C}_1$ *and* $\underline{C}_2$*, and homomorphism terms* K *and* L *such that* $[\![\Gamma; z_1{:}\underline{C}_1; K]\!] : [\![\Gamma; \underline{C}_1]\!] \longrightarrow [\![\Gamma; \underline{C}_2]\!]$ *and* $[\![\Gamma; z_2{:}\underline{C}_2; L]\!] : [\![\Gamma; \underline{C}_2]\!] \longrightarrow ([\![\Gamma]\!], \underline{D})$*, for some* $\underline{D} : [\![\Gamma]\!] \longrightarrow \mathrm{Mod}$*, then*

$$[\![\Gamma; z_1{:}\underline{C}_1; L[K/z_2]]\!] = [\![\Gamma; z_2{:}\underline{C}_2; L]\!] \circ [\![\Gamma; z_1{:}\underline{C}_1; K]\!]$$

Finally, we prove the main soundness theorem for our language.

Theorem 8.6 (Soundness). $[\![-]\!]$ *is defined on all well-formed contexts, well-formed types, and well-typed terms, and it identifies definitionally equal contexts, types, and terms.*

Proof. We prove this theorem by induction on the given derivations. For the definitional equations that correspond to the equations given in $\mathcal{E}_{\mathrm{eff}}$ (see Fig. 4), we recall that these equations hold in the countable Lawvere theory $\mathcal{L}_{\mathcal{T}_{\mathrm{eff}}}$ by construction. Therefore, all models of $\mathcal{L}_{\mathcal{T}_{\mathrm{eff}}}$ validate them, including those modelling our computation types. For computation types, we prove that $[\![\Gamma; \langle A; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}} \rangle]\!]$ is defined by observing that the interpretations of the proof witnesses $\overrightarrow{W_{\mathrm{eq}}}$ ensure that each $\mathcal{M}^\gamma$ validates the equations given in $\mathcal{E}_{\mathrm{eff}}$, as required in §8.4. For computation terms, we prove that $[\![\Gamma; M \ \mathtt{as}\ x{:}U\underline{C} \ \mathtt{in}_{\overrightarrow{W_{\mathrm{op}}}; \underline{D}}\ N]\!]$ is defined by observing that the interpretations of the proof witnesses $\overrightarrow{W_{\mathrm{op}}}$ ensure that the functions f^γ derived from $[\![\Gamma, x{:}U\underline{C}; N]\!]$ commute with the operations of $[\![\Gamma; \underline{C}]\!]_2(\gamma)$ and $[\![\Gamma; \underline{D}]\!]_2(\gamma)$, as required in §8.4; the homomorphism term case is analogous. □

9 CONCLUSION AND FUTURE WORK

In this paper we have given a comprehensive account of algebraic effects and their handlers in the dependently typed setting. In particular, we gave handlers a novel type-based treatment and demonstrated that being able to handle computations into values provides a useful mechanism for reasoning about effectful computations. We also showed how to equip the resulting language with a denotational semantics based on families fibrations and models of countable Lawvere theories.

In future, we plan to combine our type-based treatment of handlers with effect-typing [Kammar et al. 2013] and multi-handlers [Lindley et al. 2017]. We also plan to investigate extending EMLTT with local effects, such as local state (e.g., Staton [2013]). In particular, while we could use our fibred effect theories to express the operations and equations of the theory of local state, our current strategy of giving semantics to our language using the corresponding countable Lawvere theories and their (set-theoretic) models would not give a desired result for local effects. Namely, it is well-known that there are no non-trivial set-theoretic models of the theory of local state [Staton 2013, Prop. 6]. Instead, we plan to generalise from fibrations based on models of countable Lawvere theories to fibrations based on the presheaf models of Staton's parameterised algebraic theories.

ACKNOWLEDGMENTS

The bulk of this work was done when I was a PhD student at the University of Edinburgh. I am grateful to James Cheney, Paul Levy, Sam Lindley, Gordon Plotkin, Tarmo Uustalu, and the anonymous reviewers for their helpful comments and useful discussions. I also thank Neel Krishnaswami, in particular, for reminding me of Hedberg's theorem for §7.1. This work was supported, in part, by the Estonian scholarship program Kristjan Jaak, which is funded and managed by the Archimedes Foundation; and by the European Research Council under ERC Starting Grant SECOMP (715753).

REFERENCES

J. Adamek and J. Rosicky. 1994. *Locally Presentable and Accessible Categories.* Number 189 in London Mathematical Society Lecture Note Series. Cambridge Univ. Press.

D. Ahman. 2017. *Fibred Computational Effects.* Ph.D. Dissertation. School of Informatics, University of Edinburgh.

D. Ahman, J. Chapman, and T. Uustalu. 2014. When is a container a comonad? *Logical Methods in Computer Science* 10, 3 (2014).

D. Ahman, N. Ghani, and G. D. Plotkin. 2016. Dependent Types and Fibred Computational Effects. In *Proc. of 19th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2016 (LNCS)*, Vol. 9634. Springer, 1–19.

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. 2017. Dijkstra Monads for Free. In *Proc. of 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2017.* ACM, 515–529.

D. Ahman and S. Staton. 2013. Normalization by Evaluation and Algebraic Effects. In *Proc. of 29th Conf. on the Mathematical Foundations of Programming Semantics, MFPS XXIX (ENTCS)*, Vol. 298. Elsevier, 51–69.

D. Ahman and T. Uustalu. 2014. Update Monads: Cointerpreting Directed Containers. In *Post-proc. of 19th Meeting "Types for Proofs and Programs", TYPES 2013 (LIPIcs)*, Vol. 26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 1–23.

A. Bauer and M. Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

E. Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013.* ACM, 133–144.

C. Casinghino. 2014. *Combining Proofs and Programs.* Ph.D. Dissertation. University of Pennsylvania.

E. W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM* 18, 8 (1975), 5.

P. Dybjer and A. Setzer. 1999. A Finite Axiomatization of Inductive-Recursive Definitions. In *Proc. 4th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'99 (LNCS)*, Vol. 1581. Springer, 129–146.

J. Egger, R. E. Møgelberg, and A. Simpson. 2014. The enriched effect calculus: syntax and semantics. *J. Log. Comput.* 24, 3 (2014), 615–654.

N. Ghani, L. Malatesta, F. N. Forsberg, and A. Setzer. 2013. Fibred Data Types. In *Proc. of 28th Ann. Symp. on Logic in Computer Science, LICS 2013.* IEEE Computer Society, 243–252.

G. A. Grätzer. 1979. *Universal Algebra* (2nd ed.). Springer.

P. Hancock and A. Setzer. 2000. Interactive programs in dependent type theory. In *Proc. of 14th Ann. Conf. of the EACSL on Computer Science Logic, CSL 2000 (LNCS)*, Vol. 1862. Springer, 317–331.

M. Hedberg. 1998. A Coherence Theorem for Martin-Löf's Type Theory. *J. Funct. Program.* 8, 4 (1998), 413–436.

D. Hillerström and S. Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proc. of 1st Wksh. on Type-Driven Development, TyDe 2016.* ACM, 15–27.

M. Hofmann. 1995. *Extensional concepts in intensional type theory.* Ph.D. Dissertation. Laboratory for Foundations in Computer Science, University of Edinburgh.

M. Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation.* Cambridge Univ. Press, 79–130.

K. Honda, V. T. Vasconcelos, and M. Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. of 7th European Symp. on Programming, ESOP 1998 (LNCS)*, Vol. 1381. Springer, 122–138.

M. Hyland, G. Plotkin, and J. Power. 2006. Combining effects: Sum and tensor. *Theor. Comput. Sci.* 357, 1–3 (2006), 70–99.

B. Jacobs. 1999. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier.

O. Kammar, S. Lindley, and N. Oury. 2013. Handlers in Action. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013.* ACM, 145–158.

O. Kammar and G. D. Plotkin. 2012. Algebraic Foundations for Effect-dependent Optimisations. In *Proc. of 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2012.* ACM, 349–360.

D. Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proc. of 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2017.* ACM, 486–499.

P. B. Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis.* Semantics Structures in Computation, Vol. 2. Springer.

P. B. Levy. 2006. Monads and Adjunctions for Global Exceptions. *ENTCS* 158 (2006), 261–287.

P. B. Levy. 2017. Contextual isomorphisms. In *Proc. of 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2017.* ACM, 400–414.

S. Lindley, C. McBride, and C. McLaughlin. 2017. Do Be Do Be Do. In *Proc. of 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2017.* ACM, 500–514.

P. Martin-Löf. 1975. An intuitionisitc theory of types, Predicative part. In *Proc. of Logic Colloquium 1973.* North-Holland, 73–118.

P. Martin-Löf. 1984. *Intuitionistic Type Theory.* Bibliopolis.
C. McBride. 2011. Functional Pearl: Kleisli arrows of outrageous fortune. *J. Funct. Program.* (2011). (To appear).
E. Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proc. of 4th Ann. Symp. on Logic in Computer Science, LICS 1989*. IEEE, 14–23.
E. Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.
G. Munch-Maccagnoni. 2013. *Syntax and Models of a non-Associative Composition of Programs and Proofs.* Ph.D. Dissertation. Univ. Paris Diderot.
A. Nanevski, G. Morrisett, and L. Birkedal. 2008. Hoare Type Theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911.
M. Okada and P. J. Scott. 1999. A Note on Rewriting Theory for Uniqueness of Iteration. *Theory Appl. Categ.* 6, 4 (1999), 47–64.
P.-M. Pédrot and N. Tabareau. 2017. An Effectful Way to Eliminate Addiction to Dependence. In *Proc. of 32nd Ann. Symp. on Logic in Computer Science, LICS 2017*. 1–12.
A. M. Pitts. 1991. Evaluation Logic. In *Proc. IVth Higher Order Workshop (Workshops in Computing)*. Springer, 162–189.
A. M. Pitts, J. Matthiesen, and J. Derikx. 2015. A Dependent Type Theory with Abstractable Names. In *Proc. of 9th Wksh. on Logical and Semantic Frameworks, with Applications, LSFA 2014 (ENTCS)*, Vol. 312. Elsevier, 19–50.
G. Plotkin and J. Power. 2001. Semantics for Algebraic Operations. In *Proc. of 17th Conf. on the Mathematical Foundations of Programming Semantics, MFPS XVII (ENTCS)*, Vol. 45. Elsevier, 332–345.
G. D. Plotkin and J. Power. 2002. Notions of Computation Determine Monads. In *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2002 (LNCS)*, Vol. 2303. Springer, 342–356.
G. D. Plotkin and M. Pretnar. 2008. A Logic for Algebraic Effects. In *Proc. of 23th Ann. IEEE Symp. on Logic in Computer Science, LICS 2008*. IEEE, 118–129.
G. D. Plotkin and M. Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4:23 (2013).
J. Power. 2006. Countable Lawvere Theories and Computational Effects. In *Proc. of 3rd Irish Conf. on the Mathematical Foundations of Computer Science and Information Technology, MFCSIT 2004 (ENTCS)*, Vol. 161. Elsevier, 59–71.
S. Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *Proc. of 28th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2013*. IEEE, 519–519.
T. Streicher. 1991. *Semantics of Type Theory. Correctness, Completeness and Independence Results.* Birkhäuser Boston.
The Agda Team. 2017. The Agda Wiki. Available at http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php. (2017).
The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study. Available at: https://homotopytypetheory.org/book/.