# Note on Algebraic Effects

From Universal Algebra to Programming Language Design

July 25, 2025

**Sources and Acknowledgments**

The foundational material is loosely based on several existing sources, and may draw on the following (among others):

- A. Bauer, *What is algebraic about algebraic effects and handlers?* [1].
- A. Nuyts, *Understanding Universal Algebra Using KEML Diagrams* [11].
- X. Leroy, *Effect Theory: From Monads to Algebraic Effects* [8].

However, the structure, emphasis, and interpretation reflect my own understanding and pedagogical goals.

**Disclaimer:** These notes are still under construction and are being developed as part of my effort to compile and clarify in the same document some foundational and practical material that I find important and interesting. Sections may be incomplete or subject to revision, or may contain errors. Feedback is welcome.

## 1 Genealogy of Ideas

### 1.1 From early semantics to monads

- **Strachey & Scott (1960s–70s)** invented *denotational semantics*, but had to churn up *ad hoc* machinery for every new effect (explicit stores for state, exceptions, *etc.*.).
- **Moggi 1989** [10] observed that every effect can be captured semantically by an endofunctor $T$ equipped with a **ret** and bind (a *Kleisli triple*)— *computational* $\lambda$*-calculus*—marking one of the first appearances of *monads* in PL theory.

### 1.2 Monads in practice

Wadler (1990) [17] introduced the concept into Haskell, giving birth to familiar abstractions like `IO`, `State s`, and `Maybe` to *simulate* effect in a pseudo-imperative way without giving up on purity. The endofunctor underlying the monad describes a syntactic translation into

a target language in which the desired effectful operations (*e.g.* **get**, **set** or **raise** can be defined, and the underlying structure of **ret** and **bind** allow to sequence effects.

Such a syntactic translation is not new, in fact it dates back to the double-negation translation, or the CPS translation in landing a pure language in target language in which **call/cc** can be defined.

### 1.3 Algebraic theories for effects

The indirect approach of monadic programming (monad first, effectful operations second), despite its strengths in structuring some effects, does not explain how different effects compose algebraically and does not provide any recipe to answering the generic question: "What is the right monad?"

Plotkin & Power [12] took the direct algebraic theories road: taking effectful operations as primitives whose computational behaviour is captured by equations. Working their way from these first principles, they proved that the canonical monad $T_\Sigma$ for such an equational theory $\Sigma$ (signature + equations) corresponds Moggi's monads. Thus exceptions, state, nondeterminism, I/O, etc. are *algebraic*.

### 1.4 Effect Handlers

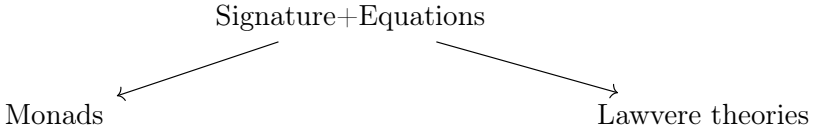
If algebraic theories provide the *syntax* of effects, we still need a way to interpret them. Plotkin & Pretnar introduced in [13] *effect handlers* as algebra homomorphisms that interpret syntax trees and produce results. Handlers generalise both `try/catch` and the categorical notion of *fold*.

## 2 Simple Algebraic Theories (SATs)

**Background and notation** We start with the standard denotational setup for effects:

- A *base category* $\mathcal{C}$ with finite products ( **Set** for the most part; more generally a symmetric monoidal $\mathcal{V}$-*enriched* category with cotensors).
- A *strong monad* $T : \mathcal{C} \to \mathcal{C}$ (strength is standard for call-by-value $\lambda$-calculus semantics). Computations of type $X$ live in $TX$ (*cf.* Moggi [10]).
- Two standard categories associated to $T$:
  - **Kleisli** $\mathcal{C}_T$: same objects as $\mathcal{C}$; morphisms $X \to Y$ are arrows $X \to TY$ in $\mathcal{C}$.
  - **Eilenberg–Moore** $T$-**Alg**: objects $(A, a)$ with $a : TA \to A$ satisfying the algebra axioms; homomorphisms $h : (A, a) \to (B, b)$ satisfy $h \circ a = b \circ Th$.

This section recalls how the following perspectives from the triangle encode the same notion of algebraic theory in three complementary ways.



## 2.1 Universal Algebra presentation

We start from the traditional data of operation symbols and equations, because that is how most working algebraists first meet the subject.

**Definition 2.1** (Algeraic Theory)**.** A *(simple) algebraic theory presentation* $\mathcal{A}$ comprises

(i) for each arity $n \in \mathbb{N}$ a set $O_{\mathcal{A}}(n)$ of $n$-ary operation symbols;

(ii) for each $n \in \mathbb{N}$ a set of equations $E_{\mathcal{A}}(n)$ whose elements are pairs of $n$-ary *terms*.

Terms $T_{\mathcal{A}}(n)$ are freely generated from variables $x_1, \ldots, x_n$ and the operations in $O_{\mathcal{A}}$.

**Definition 2.2** (Models/Algebras)**.** Given an algebraic theory $\mathcal{A}$, an $\mathcal{A}$-*model* $(A, [\![]\!])$ of comprises:

(i) a set $A$ that interprests the domain of discourse

(ii) an interpretion of every symbol $o \in O_{\mathcal{A}}(k)$ as a function $[\![o]\!] : A^k \to A$

such that each equation in $E_{\mathcal{A}}(n)$ holds point-wise. *i.e.* for every equation $\mathtt{t} =_{\mathcal{A}} \mathtt{u} \in E_{\mathcal{A}}(n)$,

$$\forall a_1, \cdots, a_n \in A.\ [\![\mathtt{t}]\!](a_1, \ldots, a_n) = [\![\mathtt{u}]\!](a_1, \ldots, a_n)$$

Homomorphisms between $\mathcal{A}$-models are the evident structure-preserving maps.

## 2.2 Monadic Viewpoint

**Proposition 2.3.** *Every algebraic theory $\mathcal{A}$ determines a monad $M_{\mathcal{A}}$ on* **Set**.

*Construction.* Concretely, $M_{\mathcal{A}}X$ is the quotient of the absolutely free term algebra $\mathbf{T}_{\mathcal{A}}X$ by the congruence generated by the equations in $E_{\mathcal{A}}$, and the monad structure is induced by term substitution. □

**Definition 2.4** (Monadic SAT)**.** A *monadic simple algebraic theory* is a monad $M = (M, \eta, \mu)$ on **Set**; an $M$*-algebra* is an Eilenberg–Moore algebra $(A, \alpha : MA \to A)$.

### Signatures, theories, and the signature functor

**Definition 2.5** (Signature and signature functor)**.** A (finitary) *signature* $\Sigma$ consists of sets $\Sigma_n$ of $n$-ary operation symbols ($n \in \mathbb{N}$). Its associated *signature functor* on **Set** is the polynomial endofunctor

$$F_\Sigma(X) \ = \ \sum_{\sigma \in \Sigma_n} X^{\,n}.$$

**The syntax functor: direct construction**

**Definition 2.6** (Syntax functor $T_\Sigma$)**.** For a set $X$, define $T_\Sigma X$ inductively as the set of finite well-founded $\Sigma$-terms over $X$:

- if $x \in X$, then $\mathrm{Var}(x) \in T_\Sigma X$;
- if $\sigma \in \Sigma_n$ and $t_1, \ldots, t_n \in T_\Sigma X$, then $\mathrm{Node}(\sigma; t_1, \ldots, t_n) \in T_\Sigma X$.

For $g : X \to Y$, define $T_\Sigma g : T_\Sigma X \to T_\Sigma Y$ by renaming variables and recursing through nodes.

**Proposition 2.7** (Monad structure)**.** *There is a monad $(T_\Sigma, \eta, \mu)$ on* **Set** *where:*

- *The unit $\eta_X : X \to T_\Sigma X, \quad \eta_X(x) = \mathrm{Var}(x)$ (inject),*
- *The multiplication $\mu_X : T_\Sigma T_\Sigma X \to T_\Sigma X$ is $t \mapsto \mathtt{t} \triangleright \mathrm{id}$ (flatten).*

*with Kleisli binding operator $\triangleright$ is given by:*

$$\mathrm{Var}(x) \triangleright f := f(x), \qquad \mathrm{Node}(\sigma; \vec{t}) \triangleright f := \mathrm{Node}\big(\sigma;\, \mathtt{t}_1 \triangleright f, \ldots, \mathtt{t}_m \triangleright f\big).$$

*Proof sketch.* Well-founded recursion gives the operations; the monad axioms reduce to the associativity of substitution and identity laws, each checked by structural induction on terms. □

**The syntax functor as the free monad on the signature functor** Seen as functor, the set $\Sigma$-terms with variables in $X$ is contained in $T_\Sigma X$, which is satisfies the following:

$$T_\Sigma X \cong X + F_\Sigma(T_\Sigma)$$

i.e a term is either a variable or constructed from a constructor in $\Sigma$ and subterms in $T_\Sigma$.

$\Longrightarrow$ Hence, for $X \in$ **Set**, writing $H_X := X + F_\Sigma(-)$. $T_\Sigma X$is the intial algebra (fixpoint) of $H_X$.

The following theorem showcases the standard construction of free monads over polynomial functors. via initial algebras and $\omega$-chains.

**Theorem 2.8.** *If $F_\Sigma$ preserves colimits of $\omega$-chains (e.g. polynomial), then for each $X$ the endofunctor $H_X$ has an initial algebra*

$$\alpha_X : X + F_\Sigma(T_\Sigma X) \xrightarrow{\cong} T_\Sigma X,$$

*and $T_\Sigma X$ is the colimit of the $\omega$-chain*

$$X \longrightarrow X + F_\Sigma X \longrightarrow X + F_\Sigma(X + F_\Sigma X) \longrightarrow \cdots.$$

*The units $\eta_X$ and multiplications $\mu_X$ arise by initiality, and together $(T_\Sigma, \eta, \mu)$ is the free monad on $F_\Sigma$.*

**Adding equations: the term monad of a theory**

**Definition 2.9** (Congruence and quotient monad)**.** Given a theory $(\Sigma, E)$, let $\equiv_E$ be the least congruence on $T_\Sigma X$ that is closed under substitution and contains all instances of axioms in $E$. Let

$$T_{\Sigma,E}X \;:=\; T_\Sigma X \big/ \equiv_E .$$

Since $\equiv_E$ is substitution-stable, $\eta$ and $\mu$ descend, making $T_{\Sigma,E}$ a monad.

**Example 2.10** (Free monoid as a term monad)**.** Let $\Sigma = \{e : 0, m : 2\}$ and $E$ be associativity and unit laws. Then $F_\Sigma(X) = 1 + X^2$ and $T_\Sigma X$ is the set of finite lists over $X$; $\eta_X(x) = [x]$ and $\mu_X$ concatenates lists-of-lists. The free $(\Sigma, E)$-algebra on $X$ has underlying set $T_{\Sigma,E}X \cong T_\Sigma X$, multiplication by concatenation, and unit the empty list.

## 2.3 Lawverian formulation

In his seminal work [6], Lawvere introduced a syntax-free formulation of algebraic theories where they are presented as small categories.

**Definition 2.11** (Lawvere theory)**.** A *Lawvere theory* is a small category $\mathcal{L}$

- equipped with finite products,
- contains an object $\star$, called the *generic object*, such that every object $X \in \mathcal{L}$ is isomorphic to a finite cartesian power of $\star$, that is there exists $n \in \mathbb{N}$ and an isomorphism $X \cong \star^n$.

Intuition:

- $\star$ represents the domain of discourse, the set in which te variables live.
- $\mathcal{L}(\star^n, \star^m)$ is the set of $m$-tuples of $n$-ary terms that can be built out of the operations in $O_\mathcal{A}$ and quotiented by $E_\mathcal{A}$.

**Definition 2.12** (Models)**.** A *model* of a Lawvere theory $\mathcal{L}$ is a product-preserving functor

$$F : \mathcal{L} \to \mathbf{Set}$$

Such models form the category $\mathsf{Mod}(\mathcal{L}) := [\mathcal{L}_M, \mathbf{Set}]_\Pi$ of product-preserving functors.

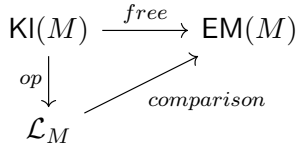
## 2.4 Relating the three formulations

**Proposition 2.13.** *Each monad $M$ on* **Set** *gives rise to a Lawvere theory $\mathcal{L}_M \;:=\; \mathsf{Kl}(M)^{\mathrm{op}}$ with a generic object $\star := 1 = \{*\}$.*

**Proposition 2.14.** *For any alhebraic theory $\mathcal{A}$, the category of $M_\mathcal{A}$-algebras is isomorphic to the category of $\mathcal{A}$-models.*

**Proposition 2.15.** *For every monad $M$, the functor category $[\mathcal{L}_M, \mathbf{Set}]_\Pi$ of product-preserving functors is equivalent to $\mathsf{EM}(M)$, compatibly with the respective forgetful functors to* **Set**.

Gathering the results above, we obtain the following KEML [11] commuting diagram: (*cf.* [11] for the full diagrammatic derivation)




### 2.5 Enriched perspective: monads $\leftrightarrow$ Lawvere theories

# Part I — Foundations of Algebraic Effects

This preliminary section is an introduction to the type of computational effects considered in the rest of the manuscript: *algebraic effects and handlers*. This programming abstraction is introduced from first principles taking the *algebraic lens* whereby an effect is presented by a signature of operation symbols together with equations; and *handlers* deconstruct such effectful computations by a universal fold principle. This lets us separate *what effectful programs say* (syntax) from *how an execution strategy interprets them* (handlers).

## 3 Minimal Universal-Algebra Preliminaries

**Signatures, terms, equations, models.** A (simple) signature is a family $(\Sigma(n))_{n\in\mathbb{N}}$ of $n$-ary operation symbols. Given a set $X$ of variables, $\Sigma$ determines a set of well-founded $\Sigma$-trees $\mathsf{Tm}_\Sigma X$ (''raw $\Sigma$-terms").

An *algebraic theory* is $(\Sigma, E)$ where $E$ is a set of axioms identifying equal open terms, inducing a congruence on $\Sigma$-trees that is stable by substitution.

A model $M$ fixes a *carrier set* $|M|$ and assigns to each symbol $\mathsf{op} \in \Sigma(n)$ a map $[\![\mathsf{op}]\!]_M : |M|^n \to |M|$ so that all equations hold pointwise. Homomorphisms are the evident structure-preserving maps between models. .

**From signatures to the term monads.** As a functor, the term construction $\mathsf{Tm}_\Sigma$ is the free monad on the *signature* endofunctor $F_\Sigma(X) = \sum_{n\in\mathbb{N},\, \mathsf{op}\in\Sigma(n)} X^n$; its unit $\eta_X : X \to \mathsf{Tm}_\Sigma X$ injects variables as one-leaf trees, and its multiplication $\mu_X : \mathsf{Tm}_\Sigma \mathsf{Tm}_\Sigma X \to \mathsf{Tm}_\Sigma X$ *flattens* a tree of trees by grafting subtrees for leaves (substitution). The monad laws reduce to the associativity and identity of substitution.

**Adding equations ⇒ the quotient monad.** Writing $\equiv_{\mathsf{E}}$ for the least substitution-stable congruence containing all instances of $E$, one can define $\mathsf{Tm}_{\Sigma,\mathsf{E}} X := \mathsf{Tm}_\Sigma X / \equiv_{\mathsf{E}}$, the quotiented set of $X$-labelled $\Sigma$-tree. Because $\equiv_{\mathsf{E}}$ is *closed under substitution*, both $\eta$ and $\mu$ factor through the quotient, so $\mathsf{Tm}_{\Sigma,\mathsf{E}}$ is again a monad and $\mathsf{Tm}_{\Sigma,\mathsf{E}} X$ is the *free model* of thealgebraic theory given by $(\Sigma, \mathsf{E})$.

## 4 Computational Effects as Algebraic Theories

> *Effectful behaviour is presented by an algebraic theory* $\mathcal{A} = (\Sigma_\mathcal{A}, \mathsf{E}_\mathcal{A})$. Operation symbols are the *constructors of effects* and equations are the *laws* governing their behaviour. The induced monad $\mathsf{Tm}_{\Sigma_\mathcal{A},\mathsf{E}_\mathcal{A}}$ is the *container* of computations, and models of $\mathcal{A}$ are exactly Eilenberg-Moore algebras for $\mathsf{Tm}_{\Sigma_\mathcal{A},\mathsf{E}_\mathcal{A}}$.

**From $n$-ary Symbols to Input-Output Arities** The minimal preesentation of simple algebraic theories above treats arity as a natural number, which is sufficient to cover for ... In order to give a programming-intuitive account for other effects, we need to generalise signatures alon two axes.

*infinite arities* Standard $n$-ary symbols ($\Sigma(n)$) and raw trees/quotients, as above; this suffices for effects with fixed finite fan-out .

$$\mathsf{op} : O_{\mathsf{op}}, \text{ with } O_{\mathsf{op}} \in \mathbf{Set}$$

so a node labelled $\mathsf{op}$ branches along "$O_{\mathsf{op}}$" subtrees, i.e a continuation from $O_{\mathsf{op}}$ (effect output) to $\Sigma$-trees.

*parameters* Many operations are more naturally formulated as carrying a parameter at each use site [1], we use a parametrised signature.

$$(\mathsf{op}_i)_{i \in I\mathsf{op}} : n, \text{ with } I_{\mathsf{op}} \in \mathbf{Set}$$

so a node labelled $\mathsf{op}_i \in \Sigma(n)$ stores a parameter $a \in I$ alongside $n$ subtrees. This legitimises speaking of an *input attached to a symbol* already at the signature level . State example uses exactly this: $\mathsf{sett}$ carries $s \in S$ as a parameter; **get** is parameterless .

**algebraic operations via *generic effects*.** Following the above generalisations, the signature $\Sigma$ of an effect is given by elements of the form:

$$\mathsf{op} : I \to O, \text{ with } I, O \in \mathbf{Set}$$

A similar $\Sigma$-terms construction works over the signature functor [2]

$$F_\Sigma(X) = \sum_{\mathsf{op} \in \Sigma} I_{\mathsf{op}} \times X^{O_{\mathsf{op}}},$$

In the induced monad $T$, an ary algebraic operation $\mathsf{op} : I \to O$ is a natural family $\mathsf{op}_X : (TX)^O \to (TX)^I$ homomorphic in each argument. Power & Plotkin show this is equivalent to the programmer-friendly *generic effect* $\overline{\mathsf{op}}_X : I \to TO$ from which $\mathsf{op}_X$ is obtained by substitution, *i.e.* through a Kleisli bind $(\mathsf{op}_X\, \kappa)\, i = (\overline{\mathsf{op}}_X\, i) \triangleright \kappa$. Intuition: the *output* arity sits in $O$ (or $(TX)^O$); the *input* side is what we substitute into its continuation slots. here and reserve the operational story for handlers.

**Example 4.1** (Global state)**.** We start by fixing a set of states $S$, and present the parameterised signature with two operations

$$\mathbf{get} : \mathbb{1} \twoheadrightarrow S \qquad \mathbf{set} : S \twoheadrightarrow \mathbb{1}$$

and the standard equations (using generic effects and the *kleisli* sequencing ;)

$$\overline{\mathbf{set}}(s)\ ;\ \overline{\mathbf{get}}()\ = \eta(s) \qquad \overline{\mathbf{get}}()\ ;\ \overline{\mathbf{set}}(s)\ =\ \overline{\mathbf{set}}(s) \qquad \overline{\mathbf{set}}(s)\ ;\ \overline{\mathbf{set}}(s')\ =\ \overline{\mathbf{set}}(s').$$

read returns the current store; write overwrites it; the last write trumps previous ones.

[1] The behaviour of global state presented with two operations **get**, and **set** is more naturally expressed as the inerplay of **get** and **set** $x$ where $x$ ranges over the set of states

[2] Categorically, this generalisation requires working over an enriched base with cotensors $(\_)^{O_{\mathsf{op}}}$, thus allowing the generalised arities $I \to O$ (that are not just finite powers).

# 5 Tensor Products: Combining Independent Effects

Given disjoint theories $(\Sigma_1, E_1)$ and $(\Sigma_2, E_2)$, their sum $\Sigma = \Sigma_1 \uplus \Sigma_2$ models independently generated effects.

## 5.1 Dual view: Comodels and Coalgebras

Every algebraic theory admits a dual semantics via *comodels*:

$$[\![\mathsf{op}]\!]_W : I \times W \to O \times W$$

This describes how a *world state* $W$ responds to each operation request.

# 6 Interpreting Operations: Handlers

So far we have only considered the initial semantics of an algebra in which the operations remain uninterpreted and the $\Sigma$-terms remain uneavaluated.

Handlers give meaning to programs by *folding* syntax trees into a target algebra. Concretely, given a $\mathsf{Tm}_\Sigma$-algebra $(A, \alpha)$, and an interpretation of $X$-variables $r : X \to B$, a handler is the unique homomorphism[3]

$$[\![-]\!]_H : \mathsf{Tm}_\Sigma X \longrightarrow A$$

determined by a return clause $r$ and one clause per operation

$$[\![\eta_X(x)]\!]_H \;=\; r(x) \qquad [\![\mathsf{op}(i;\, (\mathtt{t}_o)_{o\in O})]\!]_H \;=\; \mathsf{op}_A(i;\, \big([\![\mathtt{t}_o]\!]_H\big)_{o\in O}).$$

Operationally, this amounts to: replacing each variable $x$ by $r(x)$ to get a $\Sigma$-*tree* of $A$'s, then *folding* the tree using the *single-layer evaluaor* $\alpha$.

**Example 6.1. Nondeterminism $\to$ lists.** Let $\Sigma = \{\mathbf{fail} : 0,\ \mathbf{choose} : 2\}$ with semilattice laws. Handle to lists $B = \mathrm{List}\, X$.

$$r(x) = [x], \qquad \mathbf{fail}_B = [\,], \qquad \mathbf{choose}_B(u, v) = u{+\!\!+}v.$$

**Exceptions $\to$ option.** $\Sigma = \{\mathbf{raise}_e : 0 \mid e \in E\}$. With $B = \mathrm{Option}\, X$, let $r(x) = \mathrm{Some}(x)$ and $\mathbf{raise}_{e,B} = \mathrm{None}$.

The resumption clause has no continuations (nullary op), matching the algebraic picture.

**A handler calculus: reifying the homomorphism** We want a piece of syntax that *is* the algebra homomorphism $[\![-]\!]_H : \mathsf{Tm}_{\Sigma,\mathsf{E}} X \to B$ promised by initiality of the free monad[14].

We start with by the free $\Sigma$-term syntax

$$\text{Terms} \quad \mathsf{Tm}_\Sigma X \ni t \;::=\; \mathbf{ret}\ x \;\mid\; \mathsf{op}\big(i;\ (\mathtt{t}_o)_{o\in O}\big) \qquad (\mathsf{op} : I \to O).$$

Here $\mathbf{ret}\ x$ stands for the leaf $\eta_X(x)$, and each node $\mathsf{op}(i; \kappa)$ carries a parameter $i$ and a continuation $\kappa$ in $O \to \mathsf{Tm}_{\Sigma,\mathsf{E}} X$ (*i.e.* "O" subtrees).

[3] By initiality property

*Handler syntax* We first expose the *algebraic* clauses exactly as $[r, h] : X + \mathsf{Tm}A \to A$:

$$\mathtt{h} ::= \begin{cases} \textbf{ret}\ \ x \mapsto r(x) \\ \mathsf{op}\big(i;\ (a_o)_{o\in O}\big) \mapsto \mathsf{op}_A\big(i;\ (a_o)_{o\in O}\big) \quad \text{for each } \mathsf{op} : I_{\mathsf{op}} \to O_{\mathsf{op}} \in \Sigma \end{cases}$$

where $r : X \to B$ and each $\mathsf{op}_A : I_{\mathsf{op}} \times A^{O_{\mathsf{op}}} \to A$. The syntax is accordingly extended with the *application* of the handler to a term.

$$\text{Terms} \quad \mathsf{Tm}_\Sigma X \ni t \quad ::= \quad \cdots \mid \{\mathtt{t}\}\ \textbf{with}\ \mathtt{h}$$

*computation rules* The operational behaviour is already determined by the underlying equational theory.

(i) *handle-return* passes the value to the return clause;

(ii) *handle-op* captures the delimited continuation $\kappa$ and runs the matching clause, possibly resuming $\kappa$ zero, or several times.

$$\{\textbf{ret}\ x\}\ \textbf{with}\ \mathtt{h}\ =\ r(x) \qquad \frac{\forall i.\ \{\mathtt{t}_i\}\ \textbf{with}\ \mathtt{h} = \mathtt{u}_i}{\{\mathsf{op}(i;\ \kappa)\}\ \textbf{with}\ \mathtt{h} = \mathsf{op}_A(i;\ \kappa)}$$

**Relating the two classic lenses**

- *Moggi.* Pick a strong monad $T$ first. in which a denotation $[\![\mathsf{op}]\!]$ of *constructors of effects* can be defined. the language is interpreted in the associated Kleisli category, in which the monad $T$ is the *container* of computations [10].

  $\Longrightarrow$ Monadic programming: Programs are translated into $T$ in which effect constructors have a *concrete implementation.*[17].

- *Power & Plotkin.* Specify directly the desired effectful behaviour by $(\Sigma, E)$. $\mathsf{Tm}_{\Sigma,\mathsf{E}}$ is *the* monad whose Eilenberg-Moore algebras coincide with $\Sigma$-models, and each $\mathsf{op}$ induces a natural family $\mathsf{op}_X : (\mathsf{Tm}_{\Sigma,\mathsf{E}}X)^n \to \mathsf{Tm}_{\Sigma,\mathsf{E}}X$

  $\Longrightarrow$ Algebraic effects and handlers: Programs remain in the free syntax exposing an *abstract interface* to handlers that provide an implementation.

TODO: fix notations of the following and ..

*state monad* Given a fixed a set of states $S$. The *global state monad* on **Set** is given by the triple:

$$St\,X := S \to (X \times S)$$

$$\eta_X : X \to St\,X \qquad \eta_X(x) := \lambda s.\,\langle x, s\rangle$$

$$\mu_X\ :\ StStX \to StX, \qquad \mu_X(m) = s \mapsto m'(s') \text{ where } m(s) = \langle m', s'\rangle$$

# 7 Programming with algebraic effects and deep handlers

We adopt a minimal call-by-value core calculus with an explicit interface for algebraic operations and *deep* handlers. The calculus is intentionally small: just variables, lambdas, application, **let**, $\langle\rangle$.

**Definition 7.1** (Syntax of expressions)**.**

$$\begin{array}{rrcl} \text{values} & \mathtt{v} & ::= & x \mid \lambda x.\, \mathtt{e} \mid \langle\rangle \mid \langle \mathtt{v}_1, \mathtt{v}_2 \rangle \mid \mathbf{inl}\mathtt{v} \mid \mathbf{inr}\mathtt{v} \\ \text{computations} & \mathtt{e} & ::= & \mathtt{v} \mid \mathtt{e}\,\mathtt{e} \mid \mathbf{let}\, x \Leftarrow \mathtt{e}\ \mathbf{in}\ \mathtt{e} \mid \overline{\mathsf{op}}(\mathtt{v}) \mid \{\mathtt{e}\}\ \mathbf{with}\ \mathtt{h} \\ \text{handlers} & \mathtt{h} & ::= & \{\, \mathbf{ret}\, x \mapsto \mathtt{e};\ \ (\mathsf{op}_i(x;k) \mapsto \mathtt{e}_i)_i \,\} \\ \\ \text{evaluation contexts} & F & ::= & [\!] \mid \mathtt{e}\,F \mid F\,\mathtt{v} \mid \mathbf{let}\, x \Leftarrow F\ \mathbf{in}\ \mathtt{e} \\ & E & ::= & [\!] \mid \{E\}\ \mathbf{with}\ \mathtt{h} \mid F \end{array}$$

**eplain how this syntax is more general?** A hndler, In each operation clause the variable $k$ is a *resumption*, a function that, when applied to a value, continues the suspended computation under the *same* handler (deep handlers), no contraints

**Definition 7.2** (Small-step semantics (deep handlers))**.** We write $E[\mathtt{e}]$ for plugging $\mathtt{e}$ into context $E$. Reduction rules:

$$\begin{array}{ll} (\beta_v) & (\lambda x.\, e)\ v \ \to\ e[x := v] \\ (\text{let}) & \mathbf{let}\, x \Leftarrow \mathtt{v}\ \mathbf{in}\ \mathtt{e} \ \to\ \mathtt{e}[x := \mathtt{v}] \\ (\text{return}) & \{\mathtt{v}\}\ \mathbf{with}\ \{\mathbf{ret}\, x \mapsto \mathtt{e}_0; \dots\} \ \to\ \mathtt{e}_0[x := \mathtt{v}] \\ (\text{perform}) & \{E[\overline{\mathsf{op}}(\mathtt{v})]\}\ \mathbf{with}\ \{\mathbf{ret}\, x \mapsto \mathtt{e}_0; \dots, \mathsf{op}(x;k) \mapsto \mathtt{e}_{\mathsf{op}}, \dots\} \\ & \qquad \to\ \mathtt{e}_{\mathsf{op}}\big[x := \mathtt{v},\ k := \lambda y.\, \{E[y]\}\ \mathbf{with}\ \mathtt{h}\big] \end{array}$$

**Remarks.**

(i) If a surrounding handler does not have a clause for $\mathsf{op}$, the operation is propagated outwards to the next handler.

(ii) We do not restrict uses of the resumption $\kappa$ (no scoped resumptions here). When a clause ends with $\kappa\mathtt{e}$, we may call it *tail-resumptive*; such clauses can be implemented without rewrapping the dynamic context

# 8 Examples, more examples

We now specialise the general picture to common effects. Each is given by operations and equations, its induced monad, and a canonical handler reading.

### Finite Nondeterminism

*Presentation.* A binary **choose** and a nullary **fail**, with the semilattice laws (associativity, commutativity, idempotence) and **fail** as unit . Trees are binary **choose**-nodes and **fail** leaves; different bracketings/permutations collapse under the axioms.

*in the induced monad.* $TX \cong \mathcal{P}_{\text{fin}}(X)$; $\eta(x) = \{x\}$, $\mu$ is union. The operations are $\mathbf{choose}_{TX}(U,V) = U \cup V$ and $\mathbf{fail}_{TX} = \emptyset$, and their generic counterparts are

$$\overline{\mathbf{choose}} : \mathbb{1} \to T\{0,1\} \cong T\mathbb{B} \qquad\qquad \overline{\mathbf{fail}} : \mathbb{1}$$

where $\overline{\mathbf{choose}}() = \{\mathtt{tt}, \mathtt{ff}\}$ and $\overline{\mathbf{fail}}()$ is undefined.

*list-collecting handler.* Evaluate an effectful computation into a list by resuming the continuation twice at each choice and concatenating the results; This is the archetypal "resume more than once" example.

**Exceptions**

*Presentation.* For a set $E$ of exceptions, constants $\mathbf{raise}(e)$, typically no extra equations . In trees, exception nodes are 0-ary and absorb their context under evaluation.

*and generic operation.* $TX \cong X + E$ with $\eta_X(x) = \mathtt{inj}_1(x)$ and $\mu_X$ collapsing nested sums

$$\mu_X(x) = \mathbf{match}\, c\, \mathbf{with}\{\mathtt{inj}_2(e) \mapsto \mathtt{inj}_2(e) \mid \mathtt{inj}_1(y) \mapsto y\}$$

The generic operation is the right injection $\overline{\mathbf{raise}}_{TX} = \mathtt{inj}_2$.

*Catchers as handlers.* A catcher $X+E \to X$ is exactly a $T$-algebra, hence a handler is ?

**Global State**

*Presentation.* For a fixed set $S$ of states, $\mathbf{get} : 1 \to S$ and $\mathbf{set} : S \to 1$ with the usual laws: $\overline{\mathbf{set}}(s); \overline{\mathbf{get}} \equiv \eta(s)$, $\overline{\mathbf{get}}; \overline{\mathbf{set}}(s) \equiv \overline{\mathbf{set}}(s)$, $\overline{\mathbf{set}}(s); \overline{\mathbf{set}}(t) \equiv \overline{\mathbf{set}}(t)$.

# 9 Handler Variants

The *deep* handlers we have described so far are the natural maps out free algebras that enjoy ideal algebraic properties. Programming practice is not so ideal in general, and once handlers are introduced they need to be considered in all their arbitrary generality, *i.e.* folds over syntax trees, hence several variants arise, each with distinct implications for typing, operational behaviour. In this section, we briefly outline some of them, thus fixing the terminology we use later.

**Deep vs. Shallow Handlers.** A *deep handler* traverses the entire syntax tree — reinstalling itself when a captured continuation is resumed to intercepts further effects invoked by the resumed computation. A shallow handler handles only the next operation;

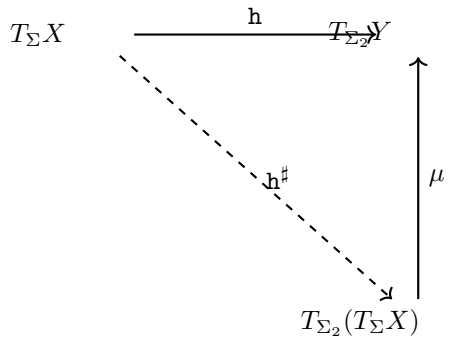



Figure 1: Deep vs. shallow handlers.

A *shallow handler*, by contrast, handles only the outermost operation node — resuming a continuation does not reinstall the handler. If further effects are be handled, the calling computation or the the continuation itself must provide a new handler. Mathematically,

this is not a catamorphism, but rather a morphism in the *Kleisli category* of the remaining operations' monad:

$$\mathtt{h}^\sharp : T_{\Sigma_1+\Sigma_2}X \longrightarrow T_{\Sigma_2}(T_{\Sigma_1 \uplus \Sigma_2}X)$$

The corresponding computational rule of such a handler $\mathtt{h} = \{\mathbf{ret}\, x \mapsto \mathtt{e}_0; \ldots, \mathtt{op}(x; k) \mapsto \mathtt{e}_{\mathtt{op}}, \ldots\}$ (syntax of the handler remains the same, semantics change) is as displayed

$$\{E[\overline{\mathtt{op}}(\mathtt{v})]\}\ \mathbf{with}\ \mathtt{h}^\sharp \to \mathtt{e}_{\mathtt{op}}\big[x := \mathtt{v},\ k := \lambda y.\ E[y]\big] \qquad ((\textit{shallow}))$$

**One vs. Multi-shot** Corresponds to the *linearity* of captured continutation seen as a *resource*. A one-shot continuation may be resumed at most once (typical in systems optimized for speed and interaction with mutation). A multi-shot continuation may be resumed multiple times (useful for backtracking/nondeterminism), but requires copying or reifying continuations.

**Parameterised Handlers.** A *parameterised* handler [3] carries adn additional state across the interpretation of operations, such as a log accumulator, or heap cell. These are homomorphisms in the Kleisli category of the state monad:

$$\mathtt{h}^\sharp : T_\Sigma X \longrightarrow (S \to T_\Sigma Y)$$

## 10 Static vs. dynamic effect instances

From Exposing all statically known operations, to exposing dynamically generated instances (namespaces for operations). New instances may be created at run time; handlers can target an instance or a the whole "known"signature.

**Instance-based Effects.** Sometimes, we wish to install multiple versions of the same operation, e.g., two separate references, or two file descriptors. This requires operations to be indexed by an *instance* (a name or label), yielding signatures like:

$$\mathbf{get}_r : 1 \to S, \quad \mathbf{set}_r : S \to 1$$

Handlers then eliminate operations with *matching* instance tags. The type system must guarantee that no aliasing or escape occurs; this leads to systems like Eff, which track instances using either affine types or singleton types.

## 11 Free and Freer Monads: Effects as Syntax Trees

WE have seen that every algebraic theory $\Sigma$ possesses a free monad $T_\Sigma$. If we want to write effectful programs and later interpret them with different handlers, we need a *first-order, inspectable syntax tree*. This can be done using *Free* monads (without introducing a full calculus for handlers)—a datatype that is:

- Inductively defined (so we can pattern-match and write folds);
- Parametric in the signature functor (so one datatype covers every theory $\Sigma$);

- The initial $\Sigma$-algebra, hence enjoys the same universal property as $T_\Sigma$.

In functional programming, the *free monad* on a functor $f$ is defined as:

$$\textbf{type}\ \texttt{Free f a} := \texttt{Pure a} \mid \texttt{Impure (f (Free f a))}$$

with $\texttt{Pure} = \eta$, and bind performing substitution then flattening ($\mu$). Its universal property is exactly that of the categorical free monad.

O. Kiselyov and H. Ishii [5] observed that the `Functor` instance required for `sig` might be cumborsome and resists scaling. Their `Freer` construction works by storing the continuation externally [5] (*à la generic effects*), thus doing away with the *functoriality* constraint (on `sig`) that would otherwise be indispensable to recurse over `f (Free f a)`.

```
type  Freer f a :=
  | Pure    : a → Freer f a
  | Impure  : f b → (b → Freer f a) -> Freer f a
```

## 11.1 Equational Reasoning with Handlers

Given $\mathcal{A} = (\Sigma_\mathcal{A}, \mathsf{E}_\mathcal{A})$ nd its its associated term monad $\mathsf{Tm}_{\Sigma_\mathcal{A},\mathsf{E}_\mathcal{A}}$. If `h` is handler interpreting $\mathsf{Tm}_{\Sigma_\mathcal{A},\mathsf{E}_\mathcal{A}} X$ in $M$, then because it is an algebra homomorphism, every equation $\texttt{t} \cong_{\mathsf{E}_\mathcal{A}} \texttt{u}$ in $\mathsf{E}_\mathcal{A}$ implies:

$$\texttt{h(t)} = \texttt{h(u)}$$

within the model model $M$. Thus, the equational theory on $\Sigma_\mathcal{A}$ lifts to the *observational theory* of the target interpretation.

| **Handler Law** | **Algebraic explanation** | **Preconditions** |
|---|---|---|
| *Fusion:* $\texttt{h}_{\Sigma_1} \circ \texttt{h}_{\Sigma_2} = \texttt{h}_{\Sigma_2 \circ \Sigma_1}$ | homomorphism composition | Both handlers are deep |
| *Commutation:* $\texttt{h}_{\Sigma_1} \circ \texttt{h}_{\Sigma_2} = \texttt{h}_{\Sigma_2} \circ \texttt{h}_{\Sigma_1}$ | tensor product of disjoint theories | disjoint signatures |

Table 1: Equational properties of handlers via algebra

# Part II — Practical Design of Algebraic Effects and Handlers

By the end of the first Part I, we possessed a purely semantic notion of a handler: an algebra homomorphism folding an abstract syntax tree, which we have generalised in the untyped minimal calculus $\Lambda_{\mathsf{eff}}$. However, the syntax alone cannot prevent the programmer from mistakenly forgetting to supply such a handler, or from installing one that handles an operation it never actually receives. It also cannot warn us that a seemingltly pure function quietly performs an effect, or that a delimited continuation will be duplicated.

Safety guarantees ($\Longrightarrow$ Type and effect systems), modularity: multiple handlers (order of installation matters $\Longrightarrow$ how to guarantee intended behaviour (i.e algebraic fusion laws)), multiple occurrences of the same effect (requires multiple handlers $\Longrightarrow$ how to get each occurence to be handled by the intended handler?)

## 12 Type-and-Effect Systems

The main issue type-and-effect systems adress is that of *type safety*: guaranteeing that a well-typed program do not *go wrong*, and most importantly do not leave effects unhandled. A weaker version is type safety is the gurantee that any effect footprint the program may leave at *run-time* is captured by its type.

To this end, several approaches have been proposed and explored, we outline the two main philosophies on reading effet annotations on types.

*Traditional* : Rows and row polymorphism: type specify what effects the computation might perform.

*Contextual reading* : capabilities: types specify what what capabilities the context must provide.

A central aim is to track, at the type level, which operations a computation may invoke.

**Modularity and repeated effects.** Handling multiple *instances* of the same effect while retaining modularity motivates either a generative, instance-scoped approach (fresh names as first-class values) or lexical/ capability schemes.

### 12.1 Row-Polymorphism

#### 12.1.1 From sets to rows

Take the set of operation names occurring in a program body, $\{\mathbf{get}, \mathbf{set}, \mathbf{raise}\}$ in a toy example. A row is simply that finite set, written in type syntax as

$$\langle \mathbf{get}, \mathbf{set}, \mathbf{raise} \rangle \quad \text{– closed row}$$

$$\langle \mathbf{get}, \mathbf{set}, \rho \rangle \quad \text{– row variable } \rho \text{ keeps it open}$$

where $\rho$ is a row variable ranging over sets of labels. Row variables endow the calculus with polymorphism: the same function can work for an arbitrary superset of effects [7, 2, 9].

The operational intuition is captured by the following; row variables as "placeholders/open slots"in the effect set waiting to be filled.


Whenever a handler eliminates an operation, say **get**, the type system enforces a corresponding set-difference

$$\langle \mathbf{get}, \mathbf{set}, \rho \rangle \ominus \langle \mathbf{get} \rangle = \langle \mathbf{set}, \rho \rangle,$$

reflecting the fact that a deep handler is a fold landing in the free syntax of the residual signature.

**Typing judgement and rules** A judgement is written

$$\Gamma \vdash \mathtt{e} : \tau \,!\, \Delta$$

meaning that `e` returns a value of type $\tau$ and may perform effects in $\{ = . \}$

TODO: insert selected typing rules?

- **ret** `v` is pure, hence row $\Delta = \emptyset$.:
- **Performing** sn effect adds a singleton row containing the invoked operation.
- **Sequential composition** merges rows by union.
- **Handle** subtracts the handled operations from the row of the body.

The type checker never needs to know the complete set of effects up-front, only that the handler removes a subset.

**Principal types and decidable inference** In [7], Leijen proves that for a Hindley-Milner core plus effect rows:

- Every typable term possesses a principal type scheme.
- That scheme is found by a variant of Algorithm W in which ordinary type variables coexist with row variables, and unification is extended to union-of-sets constraints.

**Polymorphism and the value-restriction** It is common knowledge that polymorphism combined with mutable state is a soudness pitfall. In thesetting of algebraic effects and handlers, however, the classic ML value restriction is not required: Kammar & Pretnar show in [4] that unrestricted HM let-polymorphism is sound so long as operations are monomorphic and we do not have ML-style reference cells (handlers can't express them; they only simulate dynamically scoped state). What does go wrong is the naïve extension where effect operations themselves are made polymorphic and combined with ordinary HM generalisation: later work by Sekiyama-Tsukada-Igarashi [16] shows this breaks type safety and fixes it by signature restriction, *i.e.*, constraining where quantified type variables may

appear in an operation's interface (which also supports "private effects" via public-signature restriction). An orthogonal line by Sekiyama-Igarashi [15] restores safety by restricting handlers (ruling out interfering resumptions). This reconciles "no value restriction needed" in the monomorphic operations setting with the need for extra discipline once operations become polymorphic.

# 13 Modularity, Scope, Effect instances *etc.*

## 13.1 Static vs. dynamic effect instances

From Exposing all statically known operations, to exposing dynamically generated instances (namespaces for operations). New instances may be created at run time; handlers can target an instance or a the whole "known"signature.

**Instance-based Effects.** Sometimes, we wish to install multiple versions of the same operation, e.g., two separate references, or two file descriptors. This requires operations to be indexed by an *instance* (a name or label), yielding signatures like:

$$\mathbf{get}_r : 1 \to S, \quad \mathbf{set}_r : S \to 1$$

Handlers then eliminate operations with *matching* instance tags. The type system must guarantee that no aliasing or escape occurs; this leads to systems like Eff, which track instances using either affine types or singleton types.

## 13.2 Lexically scoped instances

## 13.3 Capabilities

## 13.4 Abstraction-safe tunnelling

# 14 Graded Effect Systems

> Grading switches the perspective of effect types from "which effects?" to "how much and in what shape?", by threading a small algebra (monoid + an order) through the type rules so that sequencing composes grades and weakening tracks safe approximations.

## 14.1 Grading — measuring how much happens

relevant: [18, 19]

## 14.2 Affine and Linear Effect Tracking — controlling duplication

## References


[1] Andrej Bauer. What is algebraic about algebraic effects and handlers? 2018.

[2] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In *Algebra and Coalgebra in Computer Science: 5th International Conference, CALCO 2013, Warsaw, Poland, September 3-6, 2013. Proceedings 5*, pages 1–16. Springer, 2013.

[3] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. 30:e5.

[4] Ohad Kammar and Matija Pretnar. No value restriction is needed for algebraic effects andhandlers. *Journal of Functional Programming*, 27:e7, January 2017.

[5] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. *Journal of Functional Programming*, 25:e21, 2015.

[6] F. WILLIAM Lawvere. ALGEBRAIC THEORIES, ALGEBRAIC CATEGORIES, AND ALGEBRAIC FUNCTORS. In J. W. Addison, LEON Henkin, and ALFRED Tarski, editors, *The Theory of Models*, Studies in Logic and the Foundations of Mathematics, pages 413–418. North-Holland, January 2014.

[7] Daan Leijen. Koka: Programming with row-polymorphic effect types. *Journal of Functional Programming*, 27:e20, 2017.

[8] Xavier Leroy. Effect theory: From monads to algebraic effects, 2024. Lecture 9, Collège de France course: "Control Structures: From ‘goto‘ to Algebraic Effects".

[9] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml System: Documentation and User's Manual*. INRIA, 2025. Version 5.2, accessed August 2025.

[10] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[11] Andreas Nuyts. Understanding universal algebra using kleisli–eilenberg–moore–lawvere diagrams, 2022. v0.2, KU Leuven, Feb 21.

[12] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[13] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4):1–41, 2013.

[14] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical methods in computer science*, 9, 2013.

[15] Taro Sekiyama and Atsushi Igarashi. Handling Polymorphic Algebraic Effects. In Luís Caires, editor, *Programming Languages and Systems*, pages 353–380, Cham, 2019. Springer International Publishing.

[16] Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. Signature restriction for polymorphic algebraic effects. *Journal of Functional Programming*, 34:e7, January 2024.

[17] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, 1st International School (AFP '95)*, volume 925 of *LNCS*, pages 24–52. Springer, 1995. Based on earlier (1990) draft circulated as "Comprehending Monads".

[18] Shin ya Katsumata. Parametric effect monads and semantics of effect systems. In *POPL*, pages 633–645, 2014.

[19] Shin ya Katsumata, Dylan McDermott, Tarmo Uustalu, and Nicolas Wu. Flexible presentations of graded monads. *Proceedings of the ACM on Programming Languages*, 6(ICFP), 2022.