Cambria: a language for parametrized handlers

Jack Liell-Cock

Draft of 15-08-2025

1 Introduction

Algebraic effects and handlers [Pre15] have emerged as a compelling paradigm for structuring programs with computational effects like I/O, non-determinism, and state. By separating the specification of an effect (an operation) from its implementation (a handler), they provide a high degree of modularity and composability.

However, this approach encounters a fundamental limitation when faced with effects that require multiple, dynamically created instances of a resource. For example, how can a program create an arbitrary number of independent memory cells at runtime? The standard algebraic framework, which typically deals with a fixed set of global operations, does not offer a natural solution to this ”problem of instances”.

An answer comes from Parametrized Algebraic Theories (PATs). PATs extend algebraic theories by allowing operations to be parametrized by resource identifiers. Crucially, they formally distinguish between operations that *use* existing parameters and those that *bind* new ones, thereby providing a denotational account of resource creation.

The goal of this project is to connect the highly practical, operational framework of effect handlers with the expressive, denotational semantics of PATs. We present an extended programming language which builds upon a standard effect calculus. The core extension is the introduction of a primitive computational form, new that generates a fresh parameter. This construct provides the operational tool necessary to implement handlers that act as models for PATs. By formalizing this language and demonstrating its use for key examples, we show how the operational world of handlers can be unified with the equational world of parametrized theories, creating a practical and expressive tool for programming with sophisticated computational effects.

2 A First Language for Effects and Handlers

To build our extended language, we first establish a formal foundation based on the effect calculus in *An Introduction to Algebraic Effects and Handlers* [Pre15]. This language, employs a fine-grain call-by-value strategy, separating inert *values* from potentially effectful *computations.* This separation is crucial for reasoning about the order of evaluation in the presence of effects.

2.1 Syntax

The syntax is defined below. Values v include variables, constants, functions, and handlers themselves. Computations c include returning a value, calling an operation, sequencing, conditionals, application, and handling.

$$\begin{array}{ll}
\text{Value} & v := x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{fun}\ x \mapsto c \mid h \\
\text{Handler} & h := \mathsf{handler}\ \{\ \mathsf{return}\ x \mapsto c_r, \mathsf{op}_1(x;k) \mapsto c_1, \ldots, \mathsf{op}_n(x;k) \mapsto c_n\ \} \\
\text{Computation} & c := \mathsf{return}\ v \mid \mathsf{op}(v; x.c) \mid \mathsf{do}\, x \leftarrow c_1 \,\mathsf{in}\, c_2 \mid \mathsf{if}\, v\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2 \mid v_1\ v_2 \mid \mathsf{with}\, h\, \mathsf{handle}\, c
\end{array}$$

An operation call $\mathsf{op}(v; x.c)$ passes a parameter v to the operation op and continues with c after completion, where the result is bound to x. In the actual syntax of the language, we only expose the generic effect $!\mathsf{op}\ v$ which is syntactic sugar for $\mathsf{op}(v; x.\mathsf{return}\ x)$. We can recover an arbitrary continuation using do binding.

$$\mathsf{op}(v; x.c) = \mathsf{do}\, x \leftarrow\ !\mathsf{op}\ v\, \mathsf{in}\, c$$

2.2 Operational Semantics

The behaviour is defined by a small-step operational semantics, denoted by the relation $c \rightsquigarrow c'$. Operation calls that are not handled by any enclosing handler propagate outwards, called *forwarding.* If a call escapes the top level, the computation stops.

$$\frac{c_1 \rightsquigarrow c_1'}{\mathsf{do}\, x \leftarrow c_1 \,\mathsf{in}\, c_2 \rightsquigarrow \mathsf{do}\, x \leftarrow c_1' \,\mathsf{in}\, c_2} \qquad \frac{}{\mathsf{do}\, x \leftarrow \mathsf{return}\ v \,\mathsf{in}\, c_2 \rightsquigarrow c_2[v/x]}$$

$$\frac{}{\mathsf{do}\, x \leftarrow \mathsf{op}(v; y.c_1) \,\mathsf{in}\, c_2 \rightsquigarrow \mathsf{op}(v; y.\mathsf{do}\, x \leftarrow c_1 \,\mathsf{in}\, c_2)} \qquad \frac{}{\mathsf{if\ true\ then}\, c_1 \,\mathsf{else}\, c_2 \rightsquigarrow c_1}$$

$$\frac{}{\mathsf{if\ false\ then}\, c_1 \,\mathsf{else}\, c_2 \rightsquigarrow c_2} \qquad \frac{}{(\mathsf{fun}\ x \mapsto c)\ v \rightsquigarrow c[v/x]}$$

Let $h = \mathsf{handler}\ \{\ \mathsf{return}\ x \mapsto c_r, \ldots, \mathsf{op}(x; k) \mapsto c_{\mathsf{op}}, \ldots\ \}$:

$$\frac{c \rightsquigarrow c'}{\mathsf{with}\, h \,\mathsf{handle}\, c \rightsquigarrow \mathsf{with}\, h \,\mathsf{handle}\, c'} \qquad \frac{}{\mathsf{with}\, h \,\mathsf{handle\ return}\ v \rightsquigarrow c_r[v/x]}$$

$$\frac{}{\mathsf{with}\, h \,\mathsf{handle\, op}(v; y.c) \rightsquigarrow c_{\mathsf{op}}[v/x, (\mathsf{fun}\ y \mapsto \mathsf{with}\, h \,\mathsf{handle}\, c)/k]}$$

$$\frac{}{\mathsf{with}\, h \,\mathsf{handle\, op}'(v; y.c) \rightsquigarrow \mathsf{op}'(v; y.\mathsf{with}\, h \,\mathsf{handle}\, c)}\,(\mathsf{op}' \notin h)$$

When a handled computation returns a value, the handler's return clause is executed. When it performs an operation op that is matched by the handler, the corresponding clause c_{op} is executed. Here, the original continuation c is not discarded; it is wrapped in a function and passed to the clause, bound to the variable k. The handler continues to handle this captured continuation, which is why the body of the function is $\mathsf{with}\, h \,\mathsf{handle}\, c$. This is called *deep handling.* If an operation is not matched, it propagates outwards, but the handler remains attached to its continuation.

2.3 Type and Effect System

To ensure that programs are well-behaved, we have a type and effect system. Types are split into value types and computation types.

$$\begin{array}{ll} \text{Value Type} & A, B := \mathsf{Bool} \mid A \to \underline{C} \mid \underline{C} \Rightarrow \underline{D} \\ \text{Computation Type} & \underline{C}, \underline{D} := A\,!\,\Sigma \\ \text{Effect Set} & \Sigma := \{\mathsf{op}_1 : A_1 \to B_1, \ldots, \mathsf{op}_n : A_n \to B_n\} \end{array}$$

A computation type $A\,!\,\Sigma$ describes a computation that, returns a value of type A and may perform any of the operations in the effect set Σ. The effect set is an over-approximation. The typing judgements $\Gamma \vdash v : A$ and $\Gamma \vdash c : \underline{C}$ are defined by the following rules.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathsf{fun}\ x \mapsto c : A \to \underline{C}} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathsf{return}\ v : A\,!\,\Sigma} \qquad \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\ v_2 : \underline{C}}$$

$$\frac{\Gamma \vdash c_1 : A\,!\,\Sigma \qquad \Gamma, x : A \vdash c_2 : B\,!\,\Sigma}{\Gamma \vdash \mathsf{do}\, x \leftarrow c_1 \,\mathsf{in}\, c_2 : B\,!\,\Sigma} \qquad \frac{\Gamma \vdash v : A_{\mathsf{op}}}{\Gamma \vdash !\mathsf{op}\ v : B_{\mathsf{op}}\,!\,\Sigma}\,(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}} \in \Sigma)$$

$$\frac{\Gamma \vdash h : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathsf{with}\, h \,\mathsf{handle}\, c : \underline{D}}$$

$$\frac{\Gamma, x : A \vdash c_r : B\,!\,\Sigma' \qquad [\Gamma, x : A_i, k : B_i \to B\,!\,\Sigma' \vdash c_i : B\,!\,\Sigma']_{i=1}^{n}}{\Gamma \vdash \mathsf{handler}\ \{\ \mathsf{return}\ x \mapsto c_r, \mathsf{op}_1(x; k) \mapsto c_1, \ldots, \mathsf{op}_n(x; k) \mapsto c_n\ \} : A\,!\,\Sigma \Rightarrow B\,!\,\Sigma'}\,(\Sigma \setminus \{\mathsf{op}_i\}_{i=1}^{n} \subseteq \Sigma')$$

The typing rule for a handler is the most intricate. To have type $A \,!\, \Sigma \Rightarrow B \,!\, \Sigma'$, a handler must transform a computation of type $A \,!\, \Sigma$ into one of type $B \,!\, \Sigma'$. This requires that its return clause can handle a value of type A, and each operation clause for $\mathsf{op}_i : A_i \to B_i$ can handle a parameter of type A_i and a continuation that accepts a result of type B_i. Any operations in Σ that are not explicitly handled must be included in the output effect set Σ', as they will be propagated outwards. This system guarantees that well-typed programs do not get stuck due to type errors [Pre15].

3 An Introduction to Parametrized Algebraic Theories

To address the problem of instances of effects, we turn to the framework of parametrized algebraic theories (PATs) [Sta13b, Sta13a]. The core motivation for PATs is to provide a formal account of computational effects that involve multiple, dynamically created instances of a resource. In the standard algebraic approach for global binary state, we can define operations $\mathsf{get} : 1 \to \mathsf{Bool}$ and $\mathsf{set} : \mathsf{Bool} \to 1$. A handler can implement these by passing a single boolean value around [Pre15].

But what about two or more independent cells? We could define operations $\mathsf{set} : N \to \mathsf{Bool}$ and $\mathsf{get} : N \times \mathsf{Bool} \to 1$, but this is static and inflexible. We cannot dynamically create a new cell at runtime. The key insight of PATs is to make the resource instance itself a *parameter* to the operation.

3.1 Formalism of Parametrized Theories

Following the definitions in [Sta13b, Sta13a], a parametrized algebraic theory is built upon a more structured notion of an operator signature.

3.1.1 Parametrized Signatures

In PATs, an operator op is assigned a signature of the form $\mathsf{op} : (n \mid m_1, \ldots, m_p)$. This signature has two components: $n \in \mathbb{N}$ is the number of parameters the operation op *uses*, while each $m_1, \ldots, m_p \in \mathbb{N}$ indicates a different continuation, with the i-th continuation binding m_i *fresh* parameters local to that branch. This distinction between using existing parameters and binding new ones allows the theory to formally capture the manipulation of existing resources and the creation of fresh ones.

3.1.2 Parametrized Contexts and Terms

Terms in a PAT are judged within a parametrized context $\Gamma \mid \Delta$.

- $\Gamma = x_1 : m_1, \ldots, x_n : m_n$ is a context of *computation variables*. Each variable x_i is a placeholder for a computation that itself binds m_i parameters.
- $\Delta = a_1, \ldots, a_n$ is a context of *parameter variables*, representing the currently available resources or instances.

The rules for forming terms then naturally follow this structure. For an operator $\mathsf{op} : (n \mid m_1, \ldots, m_p)$, a term is formed as

$$F(\vec{a}, \vec{b}_1.t_1, \ldots, \vec{b}_l.t_l)$$

where $\vec{a}$ is a vector of n parameters from Δ, and each t_i is a term in a context $\Gamma \mid \Delta$ with Δ extended by m_i new parameters $\vec{b}_i$.

As a generic effect, an operator $\mathsf{op} : (n \mid m_1, \ldots, m_p)$ has signature

$$!\mathsf{op} : \Psi^n \to \Psi^{m_1} + \cdots + \Psi^{m_p}$$

where we denote the type of parameter as a new base type Ψ. The intuition is that we must pass n parameters to the operation call, and in each of the p continuations, which can be distinguished via case analysis, m_i fresh parameters are produced. We can integrate these generic effects with standard generic effects to express for instance an operator $\mathsf{op} : (1 \mid 0, 0)$ as $!\mathsf{op} : \Psi \to \mathsf{Bool}$, where case analysis on the boolean distinguishes the resulting continuation.

3.2 Examples

To make this formalism concrete, we consider two examples of PATs.

3.2.1 Local Binary State

The theory of local binary state [Sta13b] models single-bit memory cells that can be dynamically created. The parameters of the theory represent the memory locations. The operators are defined in Table 1.

Operator	Signature	Description
get	$\Psi \to \mathsf{Bool}$	Takes one parameter (a location) and has returns the result of the stored value.
set	$\Psi \times \mathsf{Bool} \to 1$	Takes one parameter (a location) and a boolean. Writes the boolean to the location and resumes the continuation.
ref	$\mathsf{Bool} \to \Psi$	Creates a new location initialized to the input boolean. It returns the fresh parameter (the new location).
eq	$\Psi \times \Psi \to 2$	Takes two parameters (locations). Returns true if they are equal and false if they are not.

Table 1: Operators for the Theory of Local Store

The signature for !ref : $\mathsf{Bool} \to \Psi$ is particularly interesting. It captures the act of *creation*. The operation itself does not depend on any existing parameter, but it introduces a new one into the context of the computation that follows. A language to implement this theory must have a mechanism to create a fresh parameter and make it available to the continuation.

3.2.2 Substitution and Jumps

This theory models a form of control flow with jumps, where the parameters represent code labels. The operators are defined in Table 2.

Operator	Signature	Description
sub	$1 \to \Psi + 1$	Takes no parameters. Binds one new parameter (a label) in its first continuation, representing the attachment of the label to that code point. The second continuation is for the code that follows the sub block.
var	$\Psi \to 0$	Takes one parameter (a label) and performs a jump to that label. It has no continuations because we jump to a new section of code.

Table 2: Operators for the Theory of Substitution and Jumps [1]

Here again the signature for the theory of substitution shows how a new resource is bound and made available to a part of the subsequent computation.

4 Extending Handlers with Parameters

We now present an extension of the effects and handlers language, designed to support handlers for parametrized algebraic theories.

4.1 Extended Syntax

First, we introduce a new primitive value type, Ψ, for the parameters. Second, we add an inbuilt operation $!\mathsf{new} : 1 \to \Psi$, which evaluates to a fresh parameter.

$$\text{Value Type} \quad v := \cdots \mid \Psi$$

The intention is that a programmer will use this inbuilt !new operation to generate fresh names to represent their resource, which can be used to form other handlers. However, this is only an option. The programmer may also implement their own fresh name generation in a handler. For instance, using the thread ID from a process that spawns a new thread.

4.2 Extended Operational Semantics

To formally define the behaviour of new, we must ensure that it generates a parameter that is unique, as shown in the following addition to the small step semantics.

$$\frac{}{\mathsf{new}(v; y.c) \rightsquigarrow c[a/y]}\,(a \text{ fresh})$$

This rule states that evaluating new reduces to returning a fresh parameter a, which we can assume is drawn from an countable set. Note that we ignore the value v because it will be of unit type.

4.3 An Extended Type and Effect System

Given we are simply adding a base type to the type and effect system, provided the semantics of new generates a fresh parameter, there are no additional updates we need to the effect system. Implementing this feature requires only a gensym library, such as the `SymbolGen` monad or `Unique` data type in Haskell. As we do not track the scope of newly generated parameters in the handler, we cannot guarantee that the newly generated parameters are not leaked into continuations. However, as practise has shown, this bug is usually implicitly captured by the type system, and the mitigation of complexity for tracking the parameters provides even greater expressiveness.

5 Programming with Parametrized Handlers

We now demonstrate implementations for handlers for the parametrized theories from Section 3.2.

5.1 Example: Local Binary State

The theory of local binary state models dynamically allocated single-bit memory cells. A handler can implement this by adopting a parameter-passing style, a technique also used for simple global state [Pre15]. However, in our parametrized setting, the state that is passed is not a single value but a *function* from parameters to their stored values, e.g. a function $s : \Psi \to \underline{\mathsf{Bool}}$. The new construct permits dynamically extending the domain of the state map. Another implementation could use a map type from parameters to booleans.

A handler implementation is as follows:

```
$state\_handler$ = handler {
  return $x$ $\to$ return (fun $s$ $\to$ return $(x,\ s)$),
  $get(a;\ k)$ $\to$ return (fun $s$ $\to$ $k\,(s\ a)\ s$),
  $set(x;\ k)$ $\to$ return (fun $s$ $\to$ $k$ () (fun $a$ $\to$ if $a \equiv$ fst $x$ then snd $x$ else $s\ a$)),
  $ref(x;\ k)$ $\to$ return (fun $s$ $\to$ do $a \leftarrow$ $!new$ () in $k\ a$ (fun $b$ $\to$ if $b \equiv a$ then $x$ else $s\ b$))
}
```

The handler can be broken down into the following parts:

- $\mathsf{ref}(x;k)$: This clause implements the ref operator. It first generates a fresh parameter a, which represents the address of the new memory cell. It then constructs a new state map, which behaves like the old state map s for all existing parameters, but returns the initial value x for the new parameter a. Finally, it resumes the continuation k, passing it the new address a as its result, and provides the *new* state map as the state parameter.

- $\mathsf{get}(a;k)$ and $\mathsf{set}(x;k)$: These clauses use existing parameters. The get clause looks up the parameter a in the current state map and passes the result to its continuation k. The state map itself remains unchanged. The set clause constructs a new state map that is updated at parameter fst x (the parameter) to snd x (the update value) and passes this new map to its continuation.

- return x: The return clause, when the computation is finished, returns the final result x wrapped in the state monad unit.

- **Equality testing:** Note that rather than use an operation $!\mathsf{eq} : \Psi \times \Psi \to \mathsf{Bool}$, we have implemented equality testing on the parameters in the language. One could replace $a \equiv b$ with $!\mathsf{eq}(a, b)$ to allow for equality testing to be handled arbitrarily, but we have chosen the former option for simplicity.

5.2 Example: Substitution and Jumps

The theory of substitution and jumps models a form of non-local control flow where parameters act as code labels. A handler for this theory must be able to create new labels and interpret jumps to them. This is instantiated by the handler returning a sum type, $X + \Psi$, where X is the original return type, to indicate the code either terminates with a value or jumps to another piece of code.

A handler is implemented as follows:

```
subst_handler = handler {
  return x  → return ( inl  x),
  var(a;  k) → return ( inr  a),
  sub(v;  k) → do a ← !new () in case k ( inl  a) of {
    inl  x → return ( inl  x),
    inr  b → if b ≡ a then k ( inl  ()) else return ( inr  b)
  }
}
```

The handler can be broken down into the following parts:

- return x: The return clause wraps a terminated value in an inl .

- $\mathsf{var}(a;k)$: The var clause, which represents an explicit jump, terminates the current computation and returns the code label a wrapped in inr . This signals to any enclosing handlers that a jump has occurred. The continuation k cannot be used because it has domain 0.

- $\mathsf{sub}(v;k)$: This clause first creates a fresh label a with new. It resumes the continuation k, with the new label. The core logic resides in the 'case' statement, which inspects the result of the continuation:
  - If the continuation returns a value inl x, the block has completed without a jump, and the value is propagated.
  - If the continuation returns a label inr b, it means a jump call occurred. If the label matches the newly created label, the jump is caught and the latter continuation is run. If the jump was to a different label, it is propagated upwards.

6 Future Work

6.1 Equational Reasoning

The most critical next step is to formally connect the operational semantics of our handlers with the equational specifications of the theories they are meant to model. For example, using the language's semantics, one should prove that the substitution handler satisfies the equational laws of the theory of substitution [FS14], such as:

$$x : 1 \mid - \vdash \mathsf{sub}(a.\mathsf{var}(a), x) = x$$

6.2 Further Theories and Applications

The expressiveness of Cambria should be further tested by implementing handlers for more advanced parametrized theories. Interesting targets include:

- A fragment of the π-calculus, where parameters represent channel names [Sta13b].
- The algebraic formulation of quantum computation, where parameters represent qubits [Sta15].
- The Beta-Bernoulli process [SSY+18].
- Dynamic threads (draft coming soon)

The current barrier is the lack of language features implemented in Cambria, such as recursion and ADTs.

6.3 Connection to Scoped Effects

Recent research has established a connection between PATs and *scoped effects* [LMM+24]. Scoped effects model effects that are not necessarily algebraic, such as transaction rollback, catching exceptions, or backtracking with *once*. These effects can be encoded as PATs where parameters are used linearly and non-commutatively. It would be interesting to see if Cambria could be augmented to track parameter usage more precisely to enforce safe implementations of scoped effects.

References

[FS14] Marcelo Fiore and Sam Staton. Substitution, jumps, and algebraic effects. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 1–10, New York, NY, USA, July 2014. Association for Computing Machinery.

[LMM+24] Sam Lindley, Cristina Matache, Sean Moss, Sam Staton, Nicolas Wu, and Zhixuan Yang. Scoped Effects as Parameterized Algebraic Theories, February 2024. arXiv:2402.03103 [cs, math].

[Pre15] Matija Pretnar. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, December 2015.

[SSY+18] Sam Staton, Dario Stein, Hongseok Yang, Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. The Beta-Bernoulli process and algebraic effects. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 141:1–141:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Sta13a] Sam Staton. An Algebraic Presentation of Predicate Logic: Extended Abstract. In *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 401–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISSN: 0302-9743.

[Sta13b] Sam Staton. Instances of Computational Effects: An Algebraic Perspective. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, page 519, USA, June 2013. IEEE Computer Society.

[Sta15] Sam Staton. Algebraic Effects, Linearity, and Quantum Programming Languages. *ACM SIGPLAN Notices*, 50(1):395–406, January 2015.