

Cambria: Resource Abstraction for Parametrized Algebraic Effects and Handlers

JACK LIELL-COCK, University of Oxford, United Kingdom

SAM STATON, University of Oxford, United Kingdom

The algebraic effects and handlers paradigm separates the concerns of the interface and implementation of computational effects in programming languages. We present CAMBRIA, a language that extends this framework to the parametrized setting. Effect signatures may use abstract parameter types that are instantiated by the handler along with the operation implementations. Parameters abstract over resources, such as memory locations or thread IDs, permitting algebraic effects to encode dynamic allocation. They are first-class in the type system but erased at runtime, requiring no coercions or type-directed reduction.

We prove parametricity via a step-indexed logical relation, formalizing the abstraction guarantee provided by parametrized handlers. We also establish type safety and classify the annotations needed for completeness of the type inference algorithm. We demonstrate CAMBRIA’s practicality with a working implementation and provide examples including local state, Pólya’s urn, and concurrent thread management. The last is a parametrized effect whose abstract thread IDs are shared between concurrent computations, going beyond standard instances. CAMBRIA is the first calculus with user-defined resource-allocating effects that guarantees, via parametricity, that client code cannot depend on how a handler represents its resources.

CCS Concepts: • **Theory of computation** → **Control primitives**; **Type theory**; *Abstraction*; *Operational semantics*; • **Software and its engineering** → Functional languages.

Additional Key Words and Phrases: algebraic effects, handlers, parametricity, logical relations, resource abstraction, type inference

1 Introduction

Algebraic effects and handlers [31] provide a principled approach to programming with side effects. A computation performs abstract *operations* such as reading state or spawning a thread, and a *handler* gives these operations meaning by supplying an implementation. This separation of interface from implementation has made algebraic effects an increasingly popular foundation for effect systems in both research and practice [4, 17, 22, 25, 36].

An effect’s interface often mentions types that the client *uses* but should never *inspect*. The underlying representation is the handler’s choice alone. Local state is the classic example. Client code may allocate memory locations, and read and write through them, but it should never need to know how a location is represented. For example, a location could be an index in a list, a unique name in a map, or an opaque token from the runtime. Committing the client to a concrete representation up front buys it nothing and forces unnecessary bookkeeping. Moreover, by keeping the locations abstract, the client cannot forge new ones, inspect them for equality, or perform arithmetic on them, except through the operations the interface provides. Thus, abstraction also provides a safety guarantee.

We present CAMBRIA, a language that extends algebraic effects and handlers to the parametrized setting. Formally, parametrized handlers are derived from the theory of parametrized algebraic theories (PATs) [37, 38], which extend classic algebraic theories with the ability of operators to bind and use abstract parameters in their arguments, denoting resource allocation and usage. Then a model of the theory, or a handler, instantiates the parameter as a type, as well as interpreting the operations. This generalization was introduced to handle the problem of *instances of effects*, where a computational effect may dynamically allocate resources at runtime, such as new memory

locations in local state. Instances of effects are a leading case for parametrized handlers, but a parameter is more generally any abstract type that an operation introduces and a handler instantiates. We demonstrate that parametrized handlers, as the operationalization of PATs, enable idiomatic implementations of diverse effects and parameters (§2.1–2.3).

One approach to instances is to represent each as a concrete object in the program. Eff originally provided first-class instances in this way [4], but later removed them, and subsequent calculi recovered instances as dynamically allocated labels [7, 12], first-class names [45], or capabilities [8]. The drawback of these approaches is that either the resources are not first-class, or the representation is fixed and imposed on the client. CAMBRIA’s alternative approach to this problem separates the concerns of *dispatch* and *instances*. That is, a single handler handles all instances of the same effect, with the operators encoding an interface to multiple instances. For example, with local state, rather than having multiple copies of the `!get` and `!set` operators, we have an additional `!ref` operator that creates a new memory location that becomes an argument to `!get` and `!set`.

Effects such as dynamically threaded concurrency [18] and code jumps [15] go beyond independent instances. Parameters are shared and used to coordinate different sub-computations. This motivates a single handler interpreting all instances, rather than a witness distinguishing each one, and CAMBRIA provides this while keeping the parameters abstract. In the concurrency example, the effect operations encode POSIX-like fork and wait. Forking dynamically spawns a thread, the reference to which is an abstract thread ID that other processes can wait on. The ability to instantiate the abstract thread ID in the parametrized handler means that each scheduler implementation remains self-contained and simple. In §2.3, we demonstrate this with three different handlers implementing child-biased scheduling, parent-biased scheduling, and dependency diagnostics.

The key method underlying CAMBRIA is that parametrized handlers decompose into classic handlers and existential types. This lets the surface language stay ergonomic while the metatheory reduces to standard, modular pieces. The main result, parametricity, then gives representation independence for effects with dynamic resources without keeping track of allocation. The decomposition also eases adoption since a language that already provides effect handlers and existential types, such as OCaml 5 [36], can support parametrized handlers with little new core machinery. The addition is moreover backwards compatible since a handler with no parameters is exactly a classic handler.

In particular, we make the following contributions:

- (1) **Language design.** We present CAMBRIA, a core calculus combining algebraic effect handlers, parameter variables, let-polymorphism, and general recursion, and establish type safety (§2–3).
- (2) **Parametricity.** We prove a parametricity theorem via a step-indexed logical relation (§4). This rests on an elaboration result (§4.3) that shows parametrized handlers decompose into classic handlers and existential types.
- (3) **Type inference.** We establish an annotation completeness result (§5) that states that it suffices to annotate only operations with arities mentioning abstract parameter types. All other types, including polymorphic ones, are inferred.
- (4) **Expressiveness.** We demonstrate that CAMBRIA can express parametrized effects with abstraction guarantees that prior work cannot, including concurrent thread management from recent work on dynamic threads [18]. We provide an implementation with runnable examples (§2).

All results novel to CAMBRIA are mechanized in the Rocq prover, relying only on functional extensionality.

2 Overview of the CAMBRIA Language

We introduce CAMBRIA through a series of examples that exercise its key features: parameter variables, let-polymorphism, general recursion, and their interactions. Each example is executable in our implementation.

2.1 Local State and Fresh Name Generation

The canonical example of a dynamic effect is local state, where we have an additional operation `!ref` that allocates a reference given an initial value, along with the standard operations `!get` and `!set` that read from and write to memory locations of type `$loc`, where the leading dollar sign marks an abstract parameter type. The following program uses this effect to allocate two reference cells and manipulate data between them before returning the value at one of the locations.

```
effect !ref : Int ~> $loc.
effect !get : $loc ~> Int.
effect !set : $loc * Int ~> Unit.
do a ← !ref 2 in
do b ← !ref 3 in
do _ ← !set (a, !get b) in
!get a
```

Note that here the `effect` declarations serve merely to specify the operation arities, and in particular which types are abstract parameter types (`$p`) rather than concrete types left to be inferred (§5). The `effect` declaration allocates nothing in itself; all generative behaviour is up to the handler of an operation such as `!ref`, as we see next. (See §6 for further comparison with other systems.)

The parametrized handle construct has an additional parameter substitution clause `[$p → A]`, allowing the abstract parameter types in a computation `c` to be instantiated before being handled by a (classic) handler `h`.

```
with [$p → A] h handle c
```

Multiple parameter instantiations are separated by commas, and the instantiation clause can be dropped when there are none, recovering the classic handle construct. In the examples that follow, we show each handler with the instantiation it is applied under, writing `[$p → A] handler {...}` for the handler and substitution pair used in `with [$p → A] h handle c`.

We may handle the local state effect using the classic parameter passing trick for stateful effects. Rather than a global value, we choose the state to be a function from the names of our memory locations to the values that they store. The function encoding is for self-containment; any associative data structure would work equally well. The convenience of parametrized effect handlers is that we need not worry about the implementation of the memory location `$name` type yet; we just need an interface for fresh name generation `!fresh : Unit ~> $name`, and equality checking `!eq : $name * $name ~> Bool`. Then, we can use another handler to implement this interface at a later stage.

```
[$loc → $name] handler {
  return x    → return (fun _ → return x),
  get a k     → return (fun s → k (s a) s),
  set (a,x) k → return (fun s → k () (fun b →
    if !eq (a, b) then return x else s b)),
  ref x k     → do a ← !fresh () in return (fun s →
    k a (fun b → if !eq (a, b) then return x else s b)),
  finally s   → s (fun _ → return 0)
```

```
}

```

This function encoding necessitates a default value for initialization (in this case 0), but parametricity (§4) guarantees that only locations initialized with `!ref` can be accessed, making this default value observationally meaningless. A genuine drawback of the encoding is that it has no means of deallocating a reference that has gone out of scope. We argue in §4.5.2 that client-transparent liveness analysis for reachability-based garbage collection would violate parametricity, and discuss alternatives.

The fresh name effect could be handled by parameter passing and incrementing an integer, concretizing the `$loc` and `$name` types to integers.

```
[ $name → Int ] handler {
  return x → return (fun _ → return x),
  fresh _ k → return (fun n → k n (n+1)),
  eq (a,b) k → k (a == b),
  finally f → f 0
}
```

Despite the concretization, the following code handled in the same way would not type-check, because `a` is parametric from the perspective of the client code.

```
do a ← !ref 2 in !set (a+1, 3)
```

The following sections demonstrate that the programmer's freedom to choose the interpretation of parameters on top of the operations leads to modular and expressive language constructs.

2.1.1 Polymorphism Meets Abstraction. Parametrized handlers support statically unbounded creation of effect instances. The number of memory locations can vary based on runtime data. Consider the following CAMBRIA program that uses a polymorphic `map` to allocate a memory location for each integer in a list, then uses the same function to access each location. The primitives `[]` and `(::)` are the constructors for the empty list and element prepending, while `uncons : List a → Unit + a * List a` unrolls the list constructor once.

```
effect !ref : Int ~> $loc.
effect !get : $loc ~> Int.
effect !set : $loc * Int ~> Unit.
do map ← return (rec map f xs →
  case uncons xs of {
    inl _ → return [],
    inr (x, xs') → f x :: map f xs'
  }) in
do vals ← return (10 :: 20 :: 30 :: []) in
do refs ← map (fun n → !ref n) vals in
map (fun r → !get r == 20) refs
```

Unbounded creation is the property that separates CAMBRIA from the lexically scoped handler calculi of [7]. This program also uses parameters under let-polymorphism, applying the polymorphic `map` to parameter-typed references. CAMBRIA is the first calculus in which this dynamic creation idiom enjoys a parametricity guarantee. The body cannot distinguish handler implementations that differ in how `$loc` is instantiated.

2.2 Two Implementations of Probabilistic Urns

Parametrized handlers allow for different implementations of the same effect interface, while preventing client code from inspecting the representation. Pólya’s urn is a probabilistic process where an urn starts by containing a red ball and a blue ball. After each sample from the urn, an additional ball of the sampled colour is placed back in the urn. That is, in the following code, the value of `b2` is more likely to match `b1`.

```
do urn ← !newurn () in
do b1 ← !sample urn in
do b2 ← !sample urn in
return (b1, b2)
```

In the probabilistic setting, CAMBRIA has built-in effects `!bernoulli` : `Double` $\rightsquigarrow$ `Bool`, which returns `true` with probability given by the input clamped to $[0, 1]$, and `!uniform` : `Unit` $\rightsquigarrow$ `Double`, which samples from the uniform distribution over $[0, 1]$. The former command can be used to handle the urn effect via the local state effect, storing pairs of integers which track the number of red and blue balls.

```
[$urn → $loc] handler {
  newurn _ k → k (!ref (1, 1)),
  sample urn k →
    do (red, blue) ← !get urn in
    do isRed ← !bernoulli (red / (red + blue)) in
    do _ ← if isRed then !set (urn, (red + 1, blue))
              else !set (urn, (red, blue + 1)) in
    k isRed
}
```

The parameter is instantiated to another parameter (`$urn → $loc`) rather than a concrete type, composing the urn abstraction with the local state abstraction.

Pólya showed that sampling from this urn admits a simple equivalent process [40]: draw a single bias p uniformly from $[0, 1]$ and then draw each sample independently from a Bernoulli distribution with that bias. Hence, we can bypass local state to produce an implementation of Pólya’s urn by instantiating `$urn` as a `Double`. This direct version is more efficient because it keeps no per-sample state and draws the bias once.

```
[$urn → Double] handler {
  newurn _ k → k (!uniform ()),
  sample p k → k (!bernoulli p)
}
```

Both handlers induce the same joint distribution on the returned pair. The state-based handler gives $P(\text{both red}) = 1/2 \cdot 2/3 = 1/3$, while the direct handler gives $\mathbb{E}_{p \sim \text{Uniform}[0,1]}[p^2] = 1/3$. This is larger than the $1/4$ probability of independent, fair draws. The client only sees `$urn` and the operations `!newurn` and `!sample`. By parametricity, it cannot inspect their implementation. So, modulo a probabilistic lifting of the logical relation (see §4.5.1), the implementations are indistinguishable by Pólya’s identity.

2.3 Dynamically Threaded Concurrency

All of the previous examples treat each parameter as an independent effect instance. Our final example of concurrent thread management [18] does not. The `!fork` operation uses multi-shot

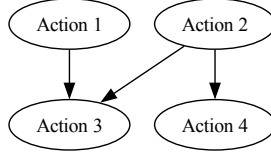


Fig. 1. An N-shaped dependency between four actions, encoded by the dynamic thread management effect.

continuations to run both child and parent threads, and the thread IDs pass between these processes while remaining abstract. The parameter therefore coordinates distinct sub-computations rather than indexing an independent instance, warranting a single handler to interpret all of them. Another such effect is code jumps [15], which appears as an example in the implementation.

A concurrent program has four operations:

- `!fork` : $\text{Unit} \leadsto \$tid + \text{Unit}$ duplicates the current process. The parent thread receives the thread ID of the child process. The child receives the unit value.
- `!wait` : $\$tid \leadsto \text{Unit}$ stalls the current thread until the process of the provided thread ID finishes.
- `!stop` : $\text{Unit} \leadsto \text{Void}$ terminates the current thread.
- `!act` : $\text{Str} \leadsto \text{Unit}$ performs a labelled observable action.

The thread IDs `$tid` are abstract, so cannot be forged, compared, or inspected. The following program constructs an N-shaped dependency between four actions, as visualized in Figure 1.

```

effect !fork : Unit  $\leadsto$  $tid + Unit.
effect !wait : $tid  $\leadsto$  Unit.
effect !stop : Unit  $\leadsto$  Void.
effect !act : Str  $\leadsto$  Unit.
case !fork () of {
  inl a1  $\rightarrow$  case !fork () of {
    inl a2  $\rightarrow$  case !fork () of {
      inl a3  $\rightarrow$  !wait a2 ; !act "Action 4" ; !stop (),
      inr _  $\rightarrow$  !wait a1 ; !wait a2 ; !act "Action 3" ; !stop () },
      inr _  $\rightarrow$  !act "Action 2" ; !stop () },
      inr _  $\rightarrow$  !act "Action 1" ; !stop () }
  }
}

```

We illustrate the expressiveness of parametrized handlers by handling this program in three different ways. The first biases the child threads to run first, running the parent process once they are completed. The second executes the parent process first, deferring each child process until it is waited on or the parent terminates. The third handler is for diagnostics. It dispatches the concurrent operations to a secondary algebraic effect that encodes the action dependencies as a DAG.

In the child-biased implementation, the parent process only receives the child's thread ID after the child has terminated, so any subsequent `!wait` on that process is trivially satisfied. Therefore, the `!wait` operation becomes a no-op, and the thread IDs can be instantiated to `Unit`. The result of a returned thread is discarded, so the handler treats returning and stopping identically. When handling the program above, the actions are executed in the order 1, 2, 3, 4.

```
[$tid  $\rightarrow$  Unit] handler {
```

```

    return _ → return (),
    act s k → !print s ; k (),
    wait _ k → k (),
    stop _ k → return (),
    fork _ k → k (inr ()) ; k (inl ())
}

```

The parent-biased handler does not have the liberty of making `!wait` a no-op. It must run the child process if it hasn't already been executed. An effective solution instantiates the thread ID as a thunk of the deferred child, $\text{Unit} \rightarrow \text{Unit} ! \{\dots\}$, where $\{\dots\}$ is shorthand for an effect signature left to be inferred. The thunk is guarded by a local boolean reference ensuring the child runs at most once. This instantiation may appear circular, because the captured continuation k , which may invoke the concurrency operations, is called within the thunk. But the continuation is resumed under the same handler, so those operations are caught, and the inferred effect signature only contains `!print` and the local-state effects `!ref`, `!set`, and `!get`. The latter can be handled similarly to those previously in §2.1, illustrating how parametrized handlers compose.

```

[$tid → Unit → Unit ! {...}] handler {
  return _ → return (),
  act s k → !print s ; k (),
  wait t k → t () ; k (),
  stop _ k → return (),
  fork _ k →
    do hasRun ← !ref false in
    do thunk ← return (fun _ → if !get hasRun then return ()
      else ( !set (hasRun, true) ; k (inr ()) )) in
    ( k (inl thunk) ; thunk () )
}

```

The wait clause now runs its thunk before resuming, forcing the child if it hasn't already run. At the end of the fork clause, the child thunk is forced one last time to ensure it runs even if nothing has waited on it. Returning and stopping are treated identically as before.

Applied to the same concurrent program above, the handler executes the actions in the order 2, 4, 1, 3. This is different to the child-biased handler, but still consistent with the dependency diagram in Figure 1. This second implementation relies on instantiating the thread ID type differently per handler.

If instead of running the program, we want a visual display of the action dependencies, we can implement a handler that converts the program into a DAG in Graphviz format. We handle the concurrency effect into an intermediate pomset effect [18] that is parametrized by a type of nodes. The only operation in this effect is `!node : List $node * Str → $node`, which adds an extra node to the DAG. The first input is the list of dependencies of the node and the second is the node's label. The result type is the newly created node's identifier. The handler is implemented by parameter passing the list of current dependencies through the computation. Informally, each operation is implemented as follows: `act` calls the node command with the current dependencies and provided label and passes on the newly created node, `wait` extends the current dependencies with the provided dependencies, `return` and `stop` forward the dependencies, and `fork` runs both the parent and child thread with the current dependencies, passing the thread ID of the child to the parent.

```

[$tid → List $node] handler {

```

```

return _ → return (fun ds → return ds),
act s k → return (fun ds → do a ← !node (ds, s) in k () (a :: [])),
wait ds' k → return (fun ds → k () (concat ds ds')),
stop _ k → return (fun ds → return ds),
fork _ k → return (fun ds → do a ← k (inr ()) ds in k (inl a) ds),
finally f → f []
}

```

The node effect can be handled by printing the DAG represented by the node operations in Graphviz format. Rather than using a user-defined implementation of the fresh effect, we use CAMBRIA's built-in one, which generates fresh instances of the Name type. The built-in primitive hash converts each unique name to a string. The result is exactly the dependency visualization in Figure 1.

```

[$node → Name] handler {
  return _ → return (fun _ → return ()),
  node (ds, s) k → return (fun _ → do a ← !fresh () in (
    !print (hash a ++ " [label=\"" ++ s ++ "\"]" );
    map (fun d → !print (hash d ++ " → " ++ hash a)) ds ;
    k a () )),
  finally f → !print "digraph {" ; f () ; !print "}"
}

```

Each of these three handlers instantiates the thread ID type to track completely different information.

3 The Calculus

The calculus we present for CAMBRIA includes only the constructs strictly necessary to expose the metatheory of parametrized handlers (parameters, parameter substitutions, and the parametrized handle form) along with general recursion and let-polymorphism. The implementation of CAMBRIA extends this calculus with standard data types and several surface conveniences, as discussed in §3.2. The examples of §2 make free use of these extensions.

3.1 Types and Syntax

The types of the language are inductively generated by the following grammar.

Value types	$A, B ::= \alpha \mid \psi \mid \text{Unit} \mid A \rightarrow \underline{C} \mid \underline{C} \Rightarrow \underline{D}$
Computation types	$\underline{C}, \underline{D} ::= \forall \vec{\alpha}. A \mid \Sigma$
Effect signatures	$\Sigma ::= \emptyset \mid \Sigma, \text{op} : A \rightarrow B$
Type variable context	$\Theta ::= \bullet \mid \Theta, \alpha$
Parameter context	$\Psi ::= \bullet \mid \Psi, \psi$
Term context	$\Gamma ::= \bullet \mid \Gamma, x : \forall \vec{\alpha}. A$

There are three contexts: Θ for type variables, Ψ for parameter variables, and Γ for term variables. The only extension to the type system beyond a standard handler calculus is the parameter variables ψ . Parameter variables can be thought of as existential type variables, because they represent an implementation detail of the effect interface. We do not allow universal quantification in operation arities, which prevents effects like heterogeneous state. This scope permits the annotation completeness result for type inference (§5), which would not hold in the polymorphic arity setting. We foresee no issues extending the type-safety and parametricity results to polymorphic arities.

The syntax of values and computations is the following.

Values	$v, w ::= x \mid () \mid \text{rec } f \ x \mapsto c \mid h$
Handlers	$h ::= \text{handler } \{\text{return } x \mapsto c_r, \text{op}_1 \ x \ k \mapsto c_1, \dots, \text{op}_m \ x \ k \mapsto c_m\}$
Computations	$c, d ::= \text{return } v \mid \text{op}(v; x.c) \mid \text{do } x \leftarrow c \text{ in } d \mid v \ w \mid \text{with } \sigma, v \text{ handle } c$
Parameter subst.	$\sigma ::= \emptyset \mid \sigma[\psi \mapsto A]$

We write $\text{fun } x \mapsto c$ as shorthand for $\text{rec } f \ x \mapsto c$ when $f \notin \text{fv}(c)$, where $\text{fv}(c)$ are the free variables in c . Similarly, we write $\text{handler } \{\text{return } x \mapsto c_r, \text{op } x \ k \mapsto c_{\text{op}}\}_{\text{op} \in \Sigma}$ as shorthand for the handler syntax where $\Sigma = \{\text{op}_1 : A_1 \rightarrow B_1, \dots, \text{op}_m : A_m \rightarrow B_m\}$. The key novelty is the *parameter substitution* σ in the handle term, which bridges the abstract types in the body to their concrete representations in the handler. We write $\text{dom}(\sigma)$ for the parameter variables bound by σ , $\text{cod}(\sigma)$ for the types they map to, and $\text{fpv}(X)$ for the free parameter variables in X .

3.2 Surface Syntax

The CAMBRIA surface language, used for the examples of §2, departs from the above calculus in a few places. These are primarily conveniences for the programmer, being more readable in practice. Each maps directly onto a calculus term, so the metatheory is unaffected.

Finally clauses. The surface admits an optional finally $x \mapsto c_f$ clause that runs after the handler returns. A handler h with a finally clause desugars at handle-time: with σ, h handle c becomes $\text{do } x \leftarrow (\text{with } \sigma, \bar{h} \text{ handle } c) \text{ in } c_f$, where $\bar{h}$ is h without the finally clause. The result is a handle within a bind, so the metatheory carries through.

Operation syntax and effect declarations. The calculus writes an operation invocation as $\text{op}(v; x.c)$, packaging the continuation $x.c$ syntactically. The surface uses the generic effect [30] $! \text{op } v$, which desugars to $\text{op}(v; x.\text{return } x)$. Hence, the calculus form $\text{op}(v; x.c)$ is equivalent to

$$\text{do } x \leftarrow ! \text{op } v \text{ in } c.$$

The surface additionally supports a local arity declaration **effect** $! \text{op} : A \rightsquigarrow B$. c used only by type inference and discussed in §5.

Surface conveniences. The remaining surface forms are sugar over the calculus. These extensions are orthogonal to the metatheory, with parametricity (Theorem 4.2), type safety (Theorem 3.3), and inference (Theorem 5.5) transferring unchanged.

- *Pattern matching.* Irrefutable patterns $p ::= x \mid _ \mid (p_1, p_2)$ are admitted in do-bindings, function arguments, and operation clauses. Pair patterns desugar to projections.
- *Multi-argument functions.* $\text{fun } x_1 \ x_2 \dots x_n \mapsto c$ and $\text{rec } f \ x_1 \dots x_n \mapsto c$ desugar to nested unary abstractions.
- *Sequencing.* $c_1 ; c_2$ desugars to $\text{do } _ \leftarrow c_1 \text{ in } c_2$.
- *Infix operators.* Arithmetic (+, −, *, /), equality (==), string concatenation (++), and cons (::) desugar to applications of named primitives.
- *Implicit do.* A computation in a value position is hoisted into a fresh do-binding around its enclosing computation. These are hoisted in left-to-right order. For example, $c_f \ c_x$ elaborates to $\text{do } f \leftarrow c_f \text{ in } (\text{do } x \leftarrow c_x \text{ in } f \ x)$.
- *Default return clause.* A handler with no explicit return clause uses the identity $\text{return } x \mapsto \text{return } x$.
- *Built-in types and operations.* The implementation provides Bool, Int, Double, Str, Name, Void, products, sums, lists, and maps, along with a small library of primitive operations including **!fresh**, **!print**, **!bernoulli**, and **!uniform**.

Type judgements.				
$\frac{\alpha \in \Theta}{\Theta; \Psi \vdash \alpha}$	$\frac{\psi \in \Psi}{\Theta; \Psi \vdash \psi}$	$\frac{}{\Theta; \Psi \vdash \text{Unit}}$	$\frac{\Theta; \Psi \vdash A \quad \Theta; \Psi \vdash \underline{C}}{\Theta; \Psi \vdash A \rightarrow \underline{C}}$	$\frac{\Theta; \Psi \vdash \underline{C} \quad \Theta; \Psi \vdash \underline{D}}{\Theta; \Psi \vdash \underline{C} \Rightarrow \underline{D}}$
$\frac{[\Theta; \Psi \vdash A_{\text{op}} \quad \Theta; \Psi \vdash B_{\text{op}}]_{(\text{op}:A_{\text{op}} \rightarrow B_{\text{op}}) \in \Sigma}}{\Theta; \Psi \vdash \Sigma}$			$\frac{\Theta; \Psi \vdash A \quad \Theta; \Psi \vdash \Sigma}{\Theta; \Psi \vdash A ! \Sigma}$	$\frac{\Theta, \vec{\alpha}; \Psi \vdash A}{\Theta; \Psi \vdash \forall \vec{\alpha}. A}$
$\frac{[\Theta; \Psi \vdash \forall \vec{\alpha}. A]_{(x:\forall \vec{\alpha}. A) \in \Gamma}}{\Theta; \Psi \vdash \Gamma}$				
Value judgements.				
$\frac{(x : \forall \vec{\alpha}. B) \in \Gamma \quad [\Theta; \Psi \vdash A_{\alpha}]_{\alpha \in \vec{\alpha}}}{\Theta; \Psi; \Gamma \vdash x : B[A_{\alpha}/\alpha]_{\alpha \in \vec{\alpha}}} \text{VAR}$			$\frac{}{\Theta; \Psi; \Gamma \vdash () : \text{Unit}} \text{UNIT}$	
$\frac{\Theta; \Psi; \Gamma, f : A \rightarrow B ! \Sigma, x : A \vdash c : B ! \Sigma}{\Theta; \Psi; \Gamma \vdash \text{rec } f \ x \mapsto c : A \rightarrow B ! \Sigma} \text{REC}$				
$\frac{\Theta; \Psi; \Gamma, x : A \vdash c_r : B ! \Sigma' \quad \Sigma \setminus \Sigma'' \subseteq \Sigma' \quad [\Theta; \Psi; \Gamma, x : A_{\text{op}}, k : B_{\text{op}} \rightarrow B ! \Sigma' \vdash c_{\text{op}} : B ! \Sigma']_{(\text{op}:A_{\text{op}} \rightarrow B_{\text{op}}) \in \Sigma''}}{\Theta; \Psi; \Gamma \vdash \text{handler } \{\text{return } x \mapsto c_r, \text{ op } x \ k \mapsto c_{\text{op}}\}_{\text{op} \in \Sigma''} : A ! \Sigma \Rightarrow B ! \Sigma'} \text{HANDLER}$				
Computation judgements.				
$\frac{\Theta; \Psi; \Gamma \vdash v : A}{\Theta; \Psi; \Gamma \vdash \text{return } v : A ! \Sigma} \text{RETURN}$			$\frac{\Theta, \vec{\alpha}; \Psi; \Gamma \vdash c : A ! \Sigma}{\Theta; \Psi; \Gamma \vdash c : \forall \vec{\alpha}. A ! \Sigma} \text{GEN}$	
$\frac{\Theta; \Psi; \Gamma \vdash v_1 : A \rightarrow \underline{C} \quad \Theta; \Psi; \Gamma \vdash v_2 : A}{\Theta; \Psi; \Gamma \vdash v_1 \ v_2 : \underline{C}} \text{APP}$				
$\frac{\Theta; \Psi; \Gamma \vdash c_1 : \forall \vec{\alpha}. A ! \Sigma \quad \Theta; \Psi; \Gamma, x : \forall \vec{\alpha}. A \vdash c_2 : B ! \Sigma}{\Theta; \Psi; \Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 : B ! \Sigma} \text{Do}$				
$\frac{(\text{op} : A_{\text{op}} \rightarrow B_{\text{op}}) \in \Sigma \quad \Theta; \Psi; \Gamma \vdash v : A_{\text{op}} \quad \Theta; \Psi; \Gamma, x : B_{\text{op}} \vdash c : A ! \Sigma}{\Theta; \Psi; \Gamma \vdash \text{op}(v; \ x.c) : A ! \Sigma} \text{OP}$				
$\frac{\Theta; \Psi; \Gamma \vdash v : \underline{C}[\sigma] \Rightarrow \underline{D} \quad \Theta; \Psi, \text{dom}(\sigma); \Gamma \vdash c : \underline{C} \quad [\Theta; \Psi \vdash A]_{A \in \text{cod}(\sigma)}}{\Theta; \Psi; \Gamma \vdash \text{with } \sigma, v \text{ handle } c : \underline{D}} \text{PARHANDLE}$				

Fig. 2. Typing rules for CAMBRIA.

3.3 Typing Rules

The typing rules for CAMBRIA are given in Figure 2. A type judgement $\Theta; \Psi \vdash A$ states that the type A is well-formed in type variable context Θ and parameter context Ψ . Value judgements are of the form $\Theta; \Psi; \Gamma \vdash v : A$ and computation judgements are of the form $\Theta; \Psi; \Gamma \vdash c : A ! \Sigma$. Both implicitly enforce that $\Theta; \Psi \vdash \Gamma$ and $\Theta; \Psi \vdash A$, and the computation judgement additionally enforces that $\Theta; \Psi \vdash \Sigma$. In particular, no $\alpha \in \vec{\alpha}$ appears free in Γ or Σ in the GEN rule. This permits let-polymorphism without any value restriction [20]. Moreover, parameters are never generalized

and remain rigid within the body of a handler so that the handler can instantiate them consistently. Otherwise the abstraction boundary would break. In the `HANDLER` rule, handled operations do not need to cover the full set of operations in the computation effect signature. The remaining operations are *forwarded*, which is indicated by $\Sigma \setminus \Sigma'' \subseteq \Sigma'$ in the premise. The continuation captured by an operation is bound as an ordinary variable, so an operation clause may resume it any number of times, permitting *multi-shot continuations*.

The only non-standard type judgement is the `PARHANDLE` rule. The rule mirrors a classic handle rule, but the handler's input type matches the body's type only after the parameter substitution is applied to it. When the parameter substitution is empty ($\sigma = \emptyset$), the rule becomes classic handler application. When non-empty, the parameters introduced by σ remain confined to the handled body. That is, for $\psi \in \text{dom}(\sigma)$, $\psi \notin \text{fpv}(\underline{D}, \Gamma)$, since $\underline{D}$ and Γ are implicitly well-formed in $\Theta; \Psi$. An operation whose arity mentions ψ may still be forwarded, but it is then exposed at its instantiated arity, so ψ does not escape. Our examples always handle such operations, but the metatheory does not require it.

3.4 Operational Semantics and Type Safety

Evaluation contexts identify where the next reduction step occurs.

$$E ::= \square \mid \text{do } x \leftarrow E \text{ in } c \mid \text{with } \sigma, v \text{ handle } E$$

Single-step reduction $c \rightsquigarrow c'$ is defined below, closed under the congruence rule: if $c \rightsquigarrow c'$ then $E[c] \rightsquigarrow E[c']$.

$$\begin{aligned} (\text{rec } f \ x \mapsto c) \ v &\rightsquigarrow c[v/x, (\text{rec } f \ x \mapsto c)/f] \\ \text{do } x \leftarrow \text{return } v \text{ in } c &\rightsquigarrow c[v/x] \\ \text{do } x \leftarrow \text{op}(v; y.c) \text{ in } c' &\rightsquigarrow \text{op}(v; y.\text{do } x \leftarrow c \text{ in } c') \end{aligned}$$

For handler reduction, let $h = \text{handler } \{\text{return } x \mapsto c_r, \text{ op } x \ k \mapsto c_{\text{op}}\}_{\text{op} \in \Sigma}$. Then handle terms reduce as follows.

$$\begin{aligned} \text{with } \sigma, h \text{ handle return } v &\rightsquigarrow c_r[v/x] \\ \text{with } \sigma, h \text{ handle op}(v; y.c) &\rightsquigarrow c_{\text{op}}[v/x, (\text{fun } y \mapsto \text{with } \emptyset, h \text{ handle } c)/k] & (\text{op} \in \Sigma) \\ \text{with } \sigma, h \text{ handle op}(v; y.c) &\rightsquigarrow \text{op}(v; y.\text{with } \emptyset, h \text{ handle } c) & (\text{op} \notin \Sigma) \end{aligned}$$

Reduction is deterministic because reducible computations decompose uniquely into an evaluation context and a redex. The continuation supplied to the operation clause reinstates the handler, giving *deep handling* [19]. In reinstated continuations, we discard the parameter substitution because it plays no role in reduction. This is the formal sense in which parameter variables are erased at runtime. There is no dynamic dispatch, no coercions, and no type-directed reduction. The parameter annotations are only used for type checking.

PROPOSITION 3.1 (PARAMETER ERASURE). *Define a parameter erasure map $\llbracket - \rrbracket$ as*

$$\llbracket \text{with } \sigma, h \text{ handle } c \rrbracket = \text{with } \emptyset, \llbracket h \rrbracket \text{ handle } \llbracket c \rrbracket$$

and structural identity otherwise. Then c is reducible if and only if $\llbracket c \rrbracket$ is reducible. Moreover, if $c \rightsquigarrow c'$ then $\llbracket c \rrbracket \rightsquigarrow \llbracket c' \rrbracket$.

PROOF. No rule inspects σ , so erasing the parameter substitutions gives a valid reduction by the same rule, establishing both results. $\square$

Type safety of a language states that well-typed terms do not get stuck. In the context of effect systems, this means a well-typed computation has reduced to a value, is *control-stuck* on an unhandled operation in its effect signature, or can take a reduction step. This result rests on a

preservation lemma that usually says reduction preserves a program's type. In CAMBRIA, to keep reduction type-agnostic, PARHANDLE reductions discard the parameter substitution. This gives a weaker notion of preservation, but still enforces type safety.

THEOREM 3.2 (PROGRESS AND PRESERVATION). *If $\bullet; \bullet; \bullet \vdash c : A ! \Sigma$, then either:*

- (1) $c = \text{return } v \text{ for some } \bullet; \bullet; \bullet \vdash v : A$
- (2) $c = \text{op}(v; y.c')$ for some $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma$, $\bullet; \bullet; \bullet \vdash v : A_{\text{op}}$, and $\bullet; \bullet; y : B_{\text{op}} \vdash c' : A ! \Sigma$.
- (3) $c \rightsquigarrow d$ and there is some $\bullet; \bullet; \bullet \vdash d' : A ! \Sigma$ such that $\lfloor d' \rfloor = \lfloor d \rfloor$.

PROOF. We first show progress by induction over the computation derivations, with canonical forms forcing the reduction steps if not a value or control-stuck operation. For preservation, we prove type substitution, parameter substitution, and variable substitution lemmas by structural induction:

- If $\Theta, \alpha; \Psi; \Gamma \vdash c : A ! \Sigma$ and $\Theta; \Psi \vdash B$, then $\Theta; \Psi; \Gamma[B/\alpha] \vdash c[B/\alpha] : (A ! \Sigma)[B/\alpha]$.
- If $\Theta; \Psi, \psi; \Gamma \vdash c : A ! \Sigma$ and $\Theta; \Psi \vdash B$, then $\Theta; \Psi; \Gamma[B/\psi] \vdash c[B/\psi] : (A ! \Sigma)[B/\psi]$.
- If $\Theta; \Psi; \Gamma, x : \forall \vec{\alpha}. B \vdash c : A ! \Sigma$ and $\Theta, \vec{\alpha}; \Psi; \Gamma \vdash v : B$, then $\Theta; \Psi; \Gamma \vdash c[v/x] : A ! \Sigma$.

Then, for non-handler reduction, preservation follows from d' instantiated to d . For handler reductions, the witness d' reinstates the parameter substitution the reduction discarded.

- For $d = c_r[v/x]$, we take $d' = c_r[v[\sigma]/x]$.
- For $d = c_{\text{op}}[v/x, (\text{fun } y \mapsto \text{with } \emptyset, h \text{ handle } c)/k]$, we take $d' = c_{\text{op}}[v[\sigma]/x, (\text{fun } y \mapsto \text{with } \emptyset, h \text{ handle } c[\sigma])/k]$.
- For $d = \text{op}(v; y.\text{with } \emptyset, h \text{ handle } c)$, we take $d' = \text{op}(v[\sigma]; y.\text{with } \emptyset, h \text{ handle } c[\sigma])$.

Then, since $\lfloor v[\sigma] \rfloor = \lfloor v \rfloor$ and $\lfloor c[\sigma] \rfloor = \lfloor c \rfloor$, and erasure commutes with substitution, $\lfloor d' \rfloor = \lfloor d \rfloor$ in all cases. By the substitution lemmas and PARHANDLE rule, these witnesses are well-typed at $\bullet; \bullet; \bullet \vdash d' : A ! \Sigma$. $\square$

In particular, if $\bullet; \bullet; \bullet \vdash c : A ! \emptyset$, then case (2) is vacuous and the result reduces to the computation being a value or taking a step.

COROLLARY 3.3 (TYPE SAFETY). *If $\bullet; \bullet; \bullet \vdash c : A ! \Sigma$ and $c \rightsquigarrow^* d$, then d is a value, control-stuck on an operation in Σ , or reducible.*

PROOF. By induction on the reduction length, maintaining the invariant that every reduct carries a well-typed witness at $A ! \Sigma$ erasing to it. The base case is c , and Theorem 3.2 supplies the witness for each successor. Theorem 3.1 and determinism force the actual reduct and the witness to share a parameter-erased successor, and Theorem 3.2 applied to the witness of d provides the trichotomy. $\square$

4 Parametricity

This section formalizes the abstraction guarantee that parametrized handlers provide to client code. We prove a parametricity result using a step-indexed binary logical relation (Theorem 4.2). Parametrized handlers decompose into classic handlers and existential types. Thus, the logical relation reduces to a standard combination of handlers, universal types, existential types, and recursion, with each type constructor contributing an independent clause. Since parameters are erased and their abstraction is captured by a single relation, the construction is direct, needing neither a biorthogonal definition of the computation relation [5], nor Kripke worlds which are required for a full mutable heap [2].

We formalize the decomposition as an elaboration into a core calculus (§4.3) and prove the parametricity theorem on the core. Adequacy transports the result back to CAMBRIA.

4.1 The Core Language

Parametrized handlers can intuitively be thought of as first packing the concrete parameter instantiation into an existential type, and then handling the abstract computation. That is, a handle with a single parameter substitution elaborates along the following lines.

with $\{\psi \mapsto A\}, v \text{ handle } c \mapsto \text{unpack } \text{pack}(\psi, A, v) \text{ as } (\psi, x) \text{ in } (\text{with } x \text{ handle } c)$

This decomposition is formalized in §4.3. Before proceeding, we introduce the syntax of the core language, the target language of the elaboration. We update the handle syntax and extend the value types, values and computations with the following constructs in place of the parametrized handle term (with $\sigma, v \text{ handle } c$) in §3.1.

Value types $A, B ::= \dots \mid \exists \psi. A$
 Values $v, w ::= \dots \mid \text{pack}(\psi, A, v)$
 Computations $c, d ::= \dots \mid \text{with } v \text{ handle } c \mid \text{unpack } v \text{ as } (\psi, x) \text{ in } c$

Parameter substitutions σ are no longer part of the syntax in the core language. Core judgements, denoted with $\vdash_c$, have a new typing derivation for existential types and, in place of the PARHANDLE rule, we have a plain handle rule, as well as existential introduction and elimination rules.

$$\frac{\Theta; \Psi, \psi \vdash_c A}{\Theta; \Psi \vdash_c \exists \psi. A} \quad \frac{\Theta; \Psi; \Gamma \vdash_c v : \exists \psi. A \quad \Theta; \Psi, \psi; \Gamma, x : A \vdash_c c : \underline{C}}{\Theta; \Psi; \Gamma \vdash_c \text{unpack } v \text{ as } (\psi, x) \text{ in } c : \underline{C}} \text{ UNPACK}$$

$$\frac{\Theta; \Psi; \Gamma \vdash_c v : \underline{C} \Rightarrow \underline{D} \quad \Theta; \Psi; \Gamma \vdash_c c : \underline{C}}{\Theta; \Psi; \Gamma \vdash_c \text{with } v \text{ handle } c : \underline{D}} \text{ HANDLE} \quad \frac{\Theta; \Psi \vdash_c A \quad \Theta; \Psi; \Gamma \vdash_c v : B[A/\psi]}{\Theta; \Psi; \Gamma \vdash_c \text{pack}(\psi, A, v) : \exists \psi. B} \text{ PACK}$$

In the UNPACK rule, $\psi \notin \text{fpv}(\underline{C}, \Gamma)$ because $\Theta; \Psi \vdash_c \Gamma$ and $\Theta; \Psi \vdash_c \underline{C}$ implicitly hold in the conclusion.

The operational semantics replaces parametrized handler reduction with classic handler reduction and adds a new reduction rule for existential terms, inheriting the remaining rules. Let $h = \text{handler } \{\text{return } x \mapsto c_r, \text{ op } x \ k \mapsto c_{\text{op}}\}_{\text{op} \in \Sigma}$, then the core language reductions on the new constructs are as follows.

$$\begin{array}{ll} \text{with } h \text{ handle return } v & \rightsquigarrow_p c_r[v/x] \\ \text{with } h \text{ handle op}(v; y.c) & \rightsquigarrow_p c_{\text{op}}[v/x, (\text{fun } y \mapsto \text{with } h \text{ handle } c)/k] \quad (\text{op} \in \Sigma) \\ \text{with } h \text{ handle op}(v; y.c) & \rightsquigarrow_p \text{op}(v; y. \text{with } h \text{ handle } c) \quad (\text{op} \notin \Sigma) \\ \text{unpack } \text{pack}(\psi, A, v) \text{ as } (\psi, x) \text{ in } c & \rightsquigarrow_a c[v/x] \end{array}$$

A classic handle evaluation frame replaces the parametrized one.

$$E ::= \dots \mid \text{with } v \text{ handle } E$$

Core language reduction is written $\rightsquigarrow$, distinguishing it from CAMBRIA reduction $\rightsquigarrow$. We further annotate core reductions with $\rightsquigarrow_p$ for *principal* steps and $\rightsquigarrow_a$ for *administrative* steps. The only administrative step is the existential elimination, which has no runtime content. In CAMBRIA, this distinction collapses into the single parametrized handle reduction, as formalized in Theorem 4.6.

4.2 Step-Indexed Logical Relation and Fundamental Theorem

A logical relation builds a model of a calculus to capture an explicit property. Each type is interpreted as a relation between pairs of closed terms, capturing when two terms are indistinguishable at that type. We build a step-indexed logical relation [1, 3] over the core language to show that contextual equivalence is parametric over the type variables and parameters. The formal result is the fundamental theorem (Theorem 4.2), which states that every well-typed term is related to itself

uniformly across all admissible interpretations of its free variables. Thus, the parameters cannot be inspected or used outside of the interfaces to the effects.

As parametrized handlers decompose into well-known constructs, the clauses of the step-indexed logical relation are standard. While we only introduce the relation on the core syntax, it extends to empty, product, sum, and list types, and all of the features in the current version of CAMBRIA. This includes row polymorphism [21, 23], which is already utilized for effect inference in the current type system. Instantiating row variables to closed signatures gives the present relation, and row generalization is handled exactly like type generalization. Since parameters and row variables are disjoint with independent substitution, the two features compose. For features not yet supported in the implementation, such as recursive types [1] and universal quantification in operation arities [34], we expect the parametricity result to generalize given the separation of concerns between parameters, type variables, and step-indexing. We leave this to future work.

The step-indexing is necessary for general recursion, and integrates well with handlers. Each operation observation strictly decreases the index, making the self-reference well-founded without resorting to induction over the size of the computation trees. Since captured continuations are bound as ordinary variables and the relation is downward-closed in the step index, each resumption is related, and multi-shot continuations need no special treatment. The step index tracks whether two terms can be discerned within that number of computation steps, so the relation is refined as the step index grows.

Definition 4.1. An *admissible relation* is a step-indexed relation on closed value terms $R \subseteq \text{Val}_0 \times \text{Val}_0 \times \mathbb{N}$, an element of which we write $(v_1, v_2)_n$, that is downward-closed: if $(v_1, v_2)_n \in R$ and $n' \leq n$, then $(v_1, v_2)_{n'} \in R$. We write $\mathbf{Rel}$ for the set of all admissible relations.

Given a type context Θ and parameter context Ψ , a *relational interpretation* $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ assigns an admissible relation to each type variable and parameter. The logical relation assigns to each typing judgement $\Theta; \Psi \vdash_c X$ a mapping from relational interpretations to an admissible relation, written as $\llbracket X \rrbracket_\rho$. When the type variable and parameter contexts are closed ($\Theta \cup \Psi = \emptyset$), the interpretation is the unique map on the empty domain, so we drop the subscript. The logical relation is defined inductively over the typing judgements and step index.

$$\begin{aligned}
\llbracket \alpha \rrbracket_\rho &= \rho(\alpha) \\
\llbracket \psi \rrbracket_\rho &= \rho(\psi) \\
\llbracket \text{Unit} \rrbracket_\rho &= \{((), ())_n \mid n \in \mathbb{N}\} \\
\llbracket A \rightarrow C \rrbracket_\rho &= \{(v_1, v_2)_n \mid \forall i \leq n. \forall (w_1, w_2)_i \in \llbracket A \rrbracket_\rho. (v_1 \ w_1, v_2 \ w_2)_i \in \llbracket C \rrbracket_\rho\} \\
\llbracket C \rightrightarrows D \rrbracket_\rho &= \{(v_1, v_2)_n \mid \forall i \leq n. \forall (c_1, c_2)_i \in \llbracket C \rrbracket_\rho. \\
&\quad (\text{with } v_1 \text{ handle } c_1, \text{ with } v_2 \text{ handle } c_2)_i \in \llbracket D \rrbracket_\rho\} \\
\llbracket \exists \psi. A \rrbracket_\rho &= \{(\text{pack}(\psi, B_1, v_1), \text{pack}(\psi, B_2, v_2))_n \mid \exists R \in \mathbf{Rel}. (v_1, v_2)_n \in \llbracket A \rrbracket_{\rho[\psi \mapsto R]}\} \\
\llbracket \forall \vec{\alpha}. A \rrbracket_\rho &= \{(v_1, v_2)_n \mid \forall \rho' \in \mathbf{Rel}^{\vec{\alpha}}. (v_1, v_2)_n \in \llbracket A \rrbracket_{\rho[\rho']}\} \\
\llbracket \Gamma \rrbracket_\rho &= \{(\gamma_1, \gamma_2)_n \mid \forall (x : \forall \vec{\alpha}. A) \in \Gamma. (\gamma_1(x), \gamma_2(x))_n \in \llbracket \forall \vec{\alpha}. A \rrbracket_\rho\}
\end{aligned}$$

Two closed computations are related at step index n if they remain observationally equivalent after n reduction steps. That is, $(c_1, c_2)_n \in \llbracket A \mid \Sigma \rrbracket_\rho$ if for all $i < n$:

- If $c_1 \rightsquigarrow^i$ return v_1 , then there is v_2 such that $c_2 \rightsquigarrow^* \text{return } v_2$ and $(v_1, v_2)_{n-i} \in \llbracket A \rrbracket_\rho$
- If $c_1 \rightsquigarrow^i \text{op}(v_1; y.d_1)$ for $(\text{op} : A_{\text{op}} \rightarrow B_{\text{op}}) \in \Sigma$, then there are v_2 and d_2 such that $c_2 \rightsquigarrow^* \text{op}(v_2; y.d_2)$ and:
 - $(v_1, v_2)_{n-i} \in \llbracket A_{\text{op}} \rrbracket_\rho$

$$- \forall j < n-i. \forall (w_1, w_2)_j \in \llbracket B_{\text{op}} \rrbracket_{\rho}. (d_1[w_1/y], d_2[w_2/y])_j \in \llbracket A ! \Sigma \rrbracket_{\rho}$$

The step index tracks reductions on the left, giving an asymmetric approximating relation. The self-reference of $\llbracket A ! \Sigma \rrbracket_{\rho}$ is well-founded because the index strictly reduces. As in the value case, computations are related at a universally quantified return type if they are related for every relational interpretation of the bound variables.

$$\llbracket \forall \vec{\alpha}. A ! \Sigma \rrbracket_{\rho} = \{(c_1, c_2)_n \mid \forall \rho' \in \mathbf{Rel}^{\vec{\alpha}}. (c_1, c_2)_n \in \llbracket A ! \Sigma \rrbracket_{\rho[\rho']}\}$$

The fundamental theorem states that this relation holds for every well-typed term.

THEOREM 4.2 (FUNDAMENTAL THEOREM). *If $\Theta; \Psi; \Gamma \vdash_c t : X$, then for all $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$, $n \in \mathbb{N}$, and $(\gamma_1, \gamma_2)_n \in \llbracket \Gamma \rrbracket_{\rho}$,*

$$(t[\gamma_1], t[\gamma_2])_n \in \llbracket X \rrbracket_{\rho}.$$

PROOF. The proof rests on three lemmas. For each, let $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ be a relational interpretation for type contexts $\Theta; \Psi$.

- **Monotonicity:** For all $\Theta; \Psi \vdash_c X$, if $(x_1, x_2)_n \in \llbracket X \rrbracket_{\rho}$ and $n' \leq n$, then $(x_1, x_2)_{n'} \in \llbracket X \rrbracket_{\rho}$. This is proved by induction over the definition of the logical relation $\llbracket - \rrbracket_{\rho}$.
- **Anti-reduction:** For all $\Theta; \Psi \vdash_c \underline{C}$, if $c_1 \rightsquigarrow^i c'_1$, $c_2 \rightsquigarrow^* c'_2$, and $(c'_1, c'_2)_n \in \llbracket \underline{C} \rrbracket_{\rho}$, then $(c_1, c_2)_{n+i} \in \llbracket \underline{C} \rrbracket_{\rho}$. This holds because reduction is deterministic.
- **Substitution:** For $\Theta, \alpha; \Psi \vdash_c X$ and $\Theta; \Psi \vdash_c A$, $\llbracket X[A/\alpha] \rrbracket_{\rho} = \llbracket X \rrbracket_{\rho[\alpha \mapsto \llbracket A \rrbracket_{\rho}]}$. Equivalently for substitution of parameter variables. This is also proved by induction over the definition of the logical relation.

Then the fundamental theorem is proved by induction over the value and computation typing judgements. The insight that parametrized handlers decompose into classic handlers and existential types makes all of these cases standard. Nevertheless, we highlight the key ones.

Rec and Handler: By inner induction on n with the same pattern. The inner IH gives the recursive occurrences related at $n-1$, the structural IH relates the bodies after a step, and anti-reduction lifts this back to n . For the handler, the return case is straightforward. For an operation $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma$, the handle term reduces either to its operation clause applied to the deep continuation ($\text{fun } y \mapsto \text{with } h \text{ handle } c$) if handled, or to $\text{op}(v; y.\text{with } h \text{ handle } c)$ if forwarded. In both cases the inner IH relates the continuations one step index lower.

Do: Via the bind lemma (Theorem 4.3).

Gen: For any $\rho' \in \mathbf{Rel}^{\vec{\alpha}}$, using $\vec{\alpha} \not\prec \text{fv}(\Gamma, \Sigma)$, we can weaken to $(\gamma_1, \gamma_2)_n \in \llbracket \Gamma \rrbracket_{\rho[\rho']}$. Then we use the IH to conclude the computations are in $\llbracket A ! \Sigma \rrbracket_{\rho[\rho']}$.

Pack: The IH gives $(v[\gamma_1], v[\gamma_2])_n \in \llbracket B[A/\psi] \rrbracket_{\rho}$. Taking $R = \llbracket A \rrbracket_{\rho}$ as the existential witness, admissible by monotonicity, the packed values are related at $\llbracket \exists \psi. B \rrbracket_{\rho}$ by the substitution lemma.

Unpack: The IH on v yields packed witnesses $\text{pack}(\psi, B_k, w_k)$ (for $k \in \{1, 2\}$), together with some $R \in \mathbf{Rel}$ relating w_1, w_2 at $\llbracket A \rrbracket_{\rho[\psi \mapsto R]}$. The IH on c at the extended contexts puts the bodies in $\llbracket \underline{C} \rrbracket_{\rho[\psi \mapsto R]} = \llbracket \underline{C} \rrbracket_{\rho}$ (using $\psi \notin \text{fpv}(\underline{C})$), and anti-reduction lifts this to the unpack terms. $\square$

LEMMA 4.3 (BIND LEMMA). *If $(c_1, c_2)_n \in \llbracket \forall \vec{\alpha}. A ! \Sigma \rrbracket_{\rho}$ and for all $i \leq n$ and $(v_1, v_2)_i \in \llbracket \forall \vec{\alpha}. A \rrbracket_{\rho}$ we have $(d_1[v_1/x], d_2[v_2/x])_i \in \llbracket B ! \Sigma \rrbracket_{\rho}$, then*

$$(\text{do } x \leftarrow c_1 \text{ in } d_1, \text{do } x \leftarrow c_2 \text{ in } d_2)_n \in \llbracket B ! \Sigma \rrbracket_{\rho}.$$

PROOF. We prove the lemma via strong induction on n . If $c_1 \rightsquigarrow^i$ return v_1 for some $i < n$, then the bind reduces to $d_1[v_1/x]$ after $i+1$ steps. Then $c_2 \rightsquigarrow^*$ return v_2 with $(v_1, v_2)_{n-i} \in \llbracket \forall \vec{\alpha}. A \rrbracket_{\rho}$, the lemma premise gives $(d_1[v_1/x], d_2[v_2/x])_{n-i-1} \in \llbracket B ! \Sigma \rrbracket_{\rho}$, and anti-reduction and monotonicity lift this back to n . If instead $c_1 \rightsquigarrow^i \text{op}(v_1; y.c'_1)$ for some $i < n$, the bind reduces to $\text{op}(v_1; y.\text{do } x \leftarrow c'_1 \text{ in } d_1)$ after $i+1$ steps. Then $c_2 \rightsquigarrow^* \text{op}(v_2; y.c'_2)$ with $(v_1, v_2)_{n-i} \in \llbracket A_{\text{op}} \rrbracket_{\rho}$ and continuations

related for $j < n-i$. The inner IH applies to $(\text{do } x \leftarrow c'_1[w_1/y] \text{ in } d_1, \text{do } x \leftarrow c'_2[w_2/y] \text{ in } d_2)$ at this strictly lower index, and anti-reduction again lifts this back to n . $\square$

4.3 Elaboration from CAMBRIA and Operational Correspondence

The fundamental theorem applies to the core language. To transfer parametricity to CAMBRIA, we need to connect the two calculi via an elaboration. We define such an elaboration and show that it is faithful. Hence, the fundamental theorem is reflected by CAMBRIA, and the abstract interfaces of the parametrized handlers are formally guaranteed.

Definition 4.4. The *elaboration translation* $(-)^{\dagger}$ from CAMBRIA's syntax to the core syntax is given by structural identity on all terms, apart from the parametrized handle clause, where it is defined inductively over the size of the parameter substitution.

$$(\text{with } \emptyset, v \text{ handle } c)^{\dagger} = \text{with } v^{\dagger} \text{ handle } c^{\dagger}$$

$$(\text{with } \sigma[\psi \mapsto A], v \text{ handle } c)^{\dagger} = \text{unpack pack}(\psi, A, v^{\dagger}) \text{ as } (\psi, x) \text{ in } (\text{with } \sigma, x \text{ handle } c)^{\dagger}$$

This elaboration preserves typing judgements, and is thus well-defined.

PROPOSITION 4.5 (TYPE PRESERVATION). *If $\Theta; \Psi; \Gamma \vdash c : \underline{C}$, then $\Theta; \Psi; \Gamma \vdash c^{\dagger} : \underline{C}$.*

PROOF. The only non-trivial case is the PARHANDLE elaboration. This is proved by inner induction on the parameter substitution. When $\sigma = \emptyset$, the rule is exactly the classic handler rule. When $\sigma = \sigma'[\psi \mapsto A]$, unwinding the typing derivation gives:

$$\frac{\Theta; \Psi; \Gamma \vdash v : \underline{C}[\sigma'[\psi \mapsto A]] \Rightarrow \underline{D} \quad \Theta; \Psi, \text{dom}(\sigma'), \psi; \Gamma \vdash c : \underline{C} \quad \Theta; \Psi \vdash A \quad \dots}{\Theta; \Psi; \Gamma \vdash \text{with } \sigma'[\psi \mapsto A], v \text{ handle } c : \underline{D}}$$

After invoking the structural IH, as $\psi \notin \text{fpv}(\underline{D})$, we can derive

$$\frac{\Theta; \Psi; \Gamma \vdash_c v : (\underline{C}[\sigma'] \Rightarrow \underline{D})[A/\psi]}{\Theta; \Psi; \Gamma \vdash_c \text{pack}(\psi, A, v) : \exists \psi. (\underline{C}[\sigma'] \Rightarrow \underline{D})}, \quad (1)$$

and weakening $\Theta; \Psi, \text{dom}(\sigma'), \psi; \Gamma \vdash_c c : \underline{C}$ gives,

$$\Theta; \Psi, \psi; \Gamma, x : \underline{C}[\sigma'] \Rightarrow \underline{D} \vdash_c \text{with } \sigma', x \text{ handle } c : \underline{D}. \quad (2)$$

Then, combining (1) and (2) we can derive

$$\Theta; \Psi; \Gamma \vdash_c \text{unpack pack}(\psi, A, v) \text{ as } (\psi, x) \text{ in } (\text{with } \sigma', x \text{ handle } c)^{\dagger} : \underline{D}$$

where we invoked the IH on $(\text{with } \sigma', x \text{ handle } c)^{\dagger}$. $\square$

This elaboration is sound up to administrative reduction. A step in CAMBRIA forces a step by its elaboration, modulo the existential unpackings which carry no runtime content. That is, each step is matched by some administrative steps from each elaboration, with a final principal step to close the gap.

PROPOSITION 4.6 (OPERATIONAL CORRESPONDENCE). *Define a simulation relation $c \triangleleft c'$ when the elaborations $c^{\dagger}$ and $c'^{\dagger}$ share a common reduct d , with $c^{\dagger} \rightsquigarrow_a^* \rightsquigarrow_p d$ and $c'^{\dagger} \rightsquigarrow_a^* d$. Then $c \rightsquigarrow c'$ implies $c \triangleleft c'$.*

PROOF. We first show by induction that $(\text{with } \sigma, h \text{ handle } b)^{\dagger} \rightsquigarrow_a^* (\text{with } h^{\dagger} \text{ handle } b^{\dagger})$. For $\sigma = \emptyset$, this is immediate after zero reductions. For $\sigma = \sigma'[\psi \mapsto A]$,

$$\begin{aligned} (\text{with } \sigma'[\psi \mapsto A], h \text{ handle } b)^{\dagger} &= \text{unpack pack}(\psi, A, h^{\dagger}) \text{ as } (\psi, x) \text{ in } (\text{with } \sigma', x \text{ handle } b)^{\dagger} \\ &\rightsquigarrow_a (\text{with } \sigma', h \text{ handle } b)^{\dagger} \\ &\rightsquigarrow_a^* \text{with } h^{\dagger} \text{ handle } b^{\dagger} \end{aligned} \quad (\text{IH})$$

For the reduction rules, the only non-trivial cases are handler reductions. Elaboration is structural identity on all other terms, so principal steps correspond exactly to CAMBRIA steps, and $d = c'^{\dagger}$ witnesses $c \triangleleft c'$. For $c = \text{with } \sigma, h \text{ handle } c_t$ where c_t is a terminal form,

$$c^{\dagger} \rightsquigarrow_a^* \text{with } h^{\dagger} \text{ handle } c_t^{\dagger} \rightsquigarrow_p c'^{\dagger}$$

because parametrized handling reinstates $\sigma = \emptyset$ and elaboration commutes with substitution. So $d = c'^{\dagger}$ witnesses $c \triangleleft c'$.

Finally, we show that $c \triangleleft c'$ implies $E[c] \triangleleft E[c']$ by induction on the evaluation context. The base case is immediate, and the `do` context follows directly from the IH because elaboration is again structural identity. For $E = \text{with } \sigma, h \text{ handle } E'$, assume $E'[c] \triangleleft E'[c']$ witnessed by d . Then,

$$\begin{aligned} E[c]^{\dagger} &\rightsquigarrow_a^* \text{with } h^{\dagger} \text{ handle } E'[c]^{\dagger} \rightsquigarrow_a^* \rightsquigarrow_p \text{with } h^{\dagger} \text{ handle } d, \\ E[c']^{\dagger} &\rightsquigarrow_a^* \text{with } h^{\dagger} \text{ handle } E'[c']^{\dagger} \rightsquigarrow_a^* \text{with } h^{\dagger} \text{ handle } d. \end{aligned}$$

So (with $h^{\dagger} \text{ handle } d$) witnesses $E[c] \triangleleft E[c']$. $\square$

Since the elaboration produces only classic handlers and existential types, and reduces to a classic handler when $\sigma = \emptyset$, a language offering these two features needs no new core constructs to extend support to parametrized handlers.

4.4 Contextual Equivalence and Adequacy of the Logical Relation

The operational correspondence allows us to state the logical relation over CAMBRIA terms via their elaborations. To reason with it, we connect the logical relation to the standard notion of program equivalence in CAMBRIA, contextual equivalence. We define two programs to be contextually equivalent when no context can distinguish their behaviour. In the presence of general recursion, it is enough to observe termination, since any behavioural difference can be converted to divergence on one side. Hence, two programs are contextually equivalent when, in any context, one terminates if and only if the other does.

Definition 4.7. In CAMBRIA, a *computation context* is a computation with a hole in it.

$$K ::= \square \mid \text{op}(v; x.K) \mid \text{do } x \leftarrow K \text{ in } c \mid \text{do } x \leftarrow c \text{ in } K \mid \text{with } \sigma, v \text{ handle } K$$

Given a well-typed computation $\Theta; \Psi; \Gamma \vdash c : A \mid \Sigma$, a *closing context* for c is a computation context K such that $\bullet; \bullet; \bullet \vdash K[c] : \text{Unit} \mid \emptyset$. Given two well-typed computations $\Theta; \Psi; \Gamma \vdash c_1, c_2 : A \mid \Sigma$, they are *contextually equivalent*, written $c_1 \approx c_2$, if for all closing contexts K ,

$$K[c_1] \rightsquigarrow^* \text{return } () \iff K[c_2] \rightsquigarrow^* \text{return } ().$$

Adequacy states that the logical relation captures this notion of contextual equivalence in CAMBRIA. In particular, if any two elaborated, well-typed CAMBRIA terms are in the logical relation for any relational interpretation, step index, and environment substitutions, then they are contextually equivalent. This is achieved via a plugging lemma which propagates the logical relation through any computation context, bridging the fundamental theorem to contextual equivalence.

LEMMA 4.8 (PLUGGING LEMMA). *If for all $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$, $n \in \mathbb{N}$, and $(\gamma_1, \gamma_2)_n \in \llbracket \Gamma \rrbracket_{\rho}$, we have*

$$(c_1^{\dagger}[\gamma_1], c_2^{\dagger}[\gamma_2])_n \in \llbracket \underline{C} \rrbracket_{\rho},$$

then for any computation context K , where $\Theta'; \Psi'; \Gamma' \vdash K[c_1], K[c_2] : \underline{C}'$, we have for all $\rho' : \Theta' \cup \Psi' \rightarrow \mathbf{Rel}$, $n \in \mathbb{N}$, and $(\gamma'_1, \gamma'_2)_n \in \llbracket \Gamma' \rrbracket_{\rho'}$,

$$(K[c_1]^{\dagger}[\gamma'_1], K[c_2]^{\dagger}[\gamma'_2])_n \in \llbracket \underline{C}' \rrbracket_{\rho'}.$$

PROOF. The proof is by structural induction on the computation context. In each case, the fundamental theorem (Theorem 4.2) relates non-hole subterms to themselves, and the IH relates the subcontext. The only non-standard case is (with σ, v handle K'), which uses inner induction on σ . The base case proceeds with the same pattern (relating non-hole terms via Theorem 4.2 and subcontexts via the IH). The inductive case elaborates as

$$(\text{with } \sigma' [\psi \mapsto A], v \text{ handle } K')^\dagger = \text{unpack pack}(\psi, A, v^\dagger) \text{ as } (\psi, x) \text{ in } (\text{with } \sigma', x \text{ handle } K')^\dagger,$$

which reduces back to $(\text{with } \sigma', v \text{ handle } K')^\dagger$, and we use the inner IH to relate the two terms after environment substitutions. $\square$

The logical relation is asymmetric because the step index tracks reductions on one side, capturing step-bounded approximate behaviour. The approximation becomes a full bidirectional equivalence for a relational interpretation ρ when for all $n \in \mathbb{N}$, $(x_1, x_2)_n \in \llbracket X \rrbracket_\rho$ and $(x_2, x_1)_n \in \llbracket X \rrbracket_{\rho^{-1}}$, where ρ^{-1} is the relational interpretation inverted pointwise. In this case we write $(x_1, x_2) \in \llbracket X \rrbracket_\rho^\approx$.

THEOREM 4.9 (ADEQUACY). *Let $\Theta; \Psi; \Gamma \vdash c_1, c_2 : \underline{C}$. Suppose for all relational interpretations $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ and related environments $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket_\rho^\approx$, we have $(c_1^\dagger[\gamma_1], c_2^\dagger[\gamma_2]) \in \llbracket \underline{C} \rrbracket_\rho^\approx$. Then $c_1 \approx c_2$.*

PROOF. Let K be any context where $\bullet; \bullet; \bullet \vdash K[c_1], K[c_2] : \text{Unit} ! \emptyset$. By Theorem 4.8 applied in both directions,

$$(K[c_1]^\dagger, K[c_2]^\dagger) \in \llbracket \text{Unit} ! \emptyset \rrbracket^\approx. \quad (3)$$

Hence, by definition of the logical relation, $K[c_1]^\dagger$ terminates at return $()$ if and only if $K[c_2]^\dagger$ does, because the operation signature is empty and the only elements of $\llbracket \text{Unit} \rrbracket$ are $(((), ()))_n$. Next, we show for $\bullet; \bullet; \bullet \vdash c : \text{Unit} ! \emptyset$,

$$c \rightsquigarrow^* \text{return } () \iff c^\dagger \rightsquigarrow^* \text{return } (). \quad (4)$$

The forward direction is given by repeated application of Theorem 4.6 using that core reductions are deterministic. In the reverse direction, each CAMBRIA step induces at least one core step (Theorem 4.6), so the finite deterministic core trace implies a finite CAMBRIA trace. By Theorem 3.3, the CAMBRIA reduction terminates at return $()$, the only terminal form of type $\text{Unit} ! \emptyset$. Thus from (3) and (4), $c_1 \approx c_2$. $\square$

4.5 Applications of Parametricity

Beyond the standard *free theorems* [42] provided by parametricity results, two consequences are key in the parametrized handler setting: representation independence between parametrized handler implementations and the ability to reason equationally within the parametrized effect theories. We present each as an implication of Theorem 4.2, illustrating them with the examples from §2.1.

4.5.1 Representation Independence for Parametrized Handlers. Two implementations of the same parametrized effect should be indistinguishable to client code that only interacts through its abstract interface. Parametricity guarantees this. If any two handlers agree up to an admissible relation on the parameter, the handled computations are observationally equivalent.

COROLLARY 4.10. *Let $\sigma_1 = \{\psi \mapsto A_1\}$ and $\sigma_2 = \{\psi \mapsto A_2\}$ be two well-typed parameter substitutions, and let*

$$\Theta; \Psi, \psi; \Gamma \vdash c : \underline{C}, \quad \Theta; \Psi; \Gamma \vdash h_1 : \underline{C}[\sigma_1] \Rightarrow \underline{D}, \quad \Theta; \Psi; \Gamma \vdash h_2 : \underline{C}[\sigma_2] \Rightarrow \underline{D}.$$

Suppose that for every relational interpretation $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ and related environment $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket_\rho^\approx$, there exists an admissible relation $R \in \mathbf{Rel}$ such that

$$(h_1^\dagger[\gamma_1], h_2^\dagger[\gamma_2]) \in \llbracket \underline{C} \Rightarrow \underline{D} \rrbracket_{\rho[\psi \mapsto R]}^\approx. \quad (5)$$

Then (with σ_1, h_1 handle c) $\approx$ (with σ_2, h_2 handle c).

PROOF. Let $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ and $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket_\rho^\approx$. By Theorem 4.9, we require

$$((\text{with } \sigma_1, h_1 \text{ handle } c)^\dagger[\gamma_1], (\text{with } \sigma_2, h_2 \text{ handle } c)^\dagger[\gamma_2]) \in \llbracket \underline{D} \rrbracket_\rho^\approx. \quad (6)$$

Let R be the witness for (5). Since $\psi \notin \text{fpv}(\Gamma)$, $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket_{\rho[\psi \mapsto R]}^\approx$. By Theorem 4.2 applied in both directions, with the inverse relational interpretation for the reverse,

$$(c^\dagger[\gamma_1], c^\dagger[\gamma_2]) \in \llbracket \underline{C} \rrbracket_{\rho[\psi \mapsto R]}^\approx.$$

The handler clause of the logical relation gives

$$(\text{with } h_1^\dagger[\gamma_1] \text{ handle } c^\dagger[\gamma_1], \text{with } h_2^\dagger[\gamma_2] \text{ handle } c^\dagger[\gamma_2]) \in \llbracket \underline{D} \rrbracket_{\rho[\psi \mapsto R]}^\approx = \llbracket \underline{D} \rrbracket_\rho^\approx$$

as $\psi \notin \text{fpv}(\underline{D})$. Finally,

$$\begin{aligned} (\text{with } \sigma_i, h_i \text{ handle } c)^\dagger[\gamma_i] &= \text{unpack pack}(\psi, A_i, h_i^\dagger[\gamma_i]) \text{ as } (\psi, x) \text{ in } (\text{with } x \text{ handle } c^\dagger[\gamma_i]) \\ &\rightsquigarrow_a (\text{with } h_i^\dagger[\gamma_i] \text{ handle } c^\dagger[\gamma_i]). \end{aligned}$$

So by anti-reduction, (6) holds. $\square$

This result extends to substitutions of multiple parameters $\sigma = \{\psi_1 \mapsto A_1, \dots, \psi_k \mapsto A_k\}$ by induction on k . The elaboration unfolds into k nested unpack-pack layers, each contributing one $\rightsquigarrow_a$ step. The proof is otherwise unchanged.

Example 4.11. We sketch a use case of Theorem 4.10 for two implementations of the fresh effect. Take h_1 to be the fresh effect handler from §2.1 with $\sigma_1 = \{\psi \mapsto \text{Int}\}$ and take h_2 to be a string-counter variant with $\sigma_2 = \{\psi \mapsto \text{Str}\}$:

```
handler {
  return x → return (fun _ → return x),
  fresh _ k → return (fun s → k s ("x" ++ s)),
  eq (a,b) k → k (a == b),
  finally f → f ""
}
```

producing the sequence of strings "", "x", "xx", ... for the names.

We choose the admissible relation $R = \{(i, "x"^i)_n \mid i, n \in \mathbb{N}\}$, where $"x"^i$ is the string of i copies of x . The clauses of the handlers satisfy this choice of R and its inverse because:

- **return** each produces a function ignoring the parameter, so the results are related.
- **fresh** applied to related i and $"x"^i$ invokes the continuation on this pair, and the resulting function is applied to related $i+1$ and $"x"^{i+1}$.
- **eq** applied to related arguments (i, j) and $(\text{"x"}^i, \text{"x"}^j)$ gives related booleans because $i \leftrightarrow \text{"x"}^i$ is injective in both directions.

So the two implementations agree up to R , satisfying the premise of Theorem 4.10. Hence, any body that uses the fresh effect is observationally equivalent under h_1 and h_2 .

The relation may do work beyond bijective encodings. The two implementations of Pólya’s urn from §2.2 are such an example. The first stores the urn’s state as a pair of integers in a memory location while the other uses a single double, with the witnessing R a *coupling* between a cell holding state (r, b) and a draw p from the Beta distribution $\text{Beta}(r, b)$. The deterministic logical relation already accounts for half of the equivalence, where the abstraction over $\$urn$ ensures that no client can observe which representation a handler uses. But it does not establish that the two handlers satisfy R because `!uniform` and `!bernoulli` are not part of the core calculus (§3). The handlers sample differently, only agreeing in distribution by conjugacy of the Beta prior to Bernoulli observations [40]. That is, $\mathbb{E}[\text{Beta}(r, b)] = r/(r + b)$ matches the predictive probability of red, and conditioning on a draw updates it to $\text{Beta}(r + 1, b)$ or $\text{Beta}(r, b + 1)$, matching the urn update. So, parametricity reduces the proof obligation to the conjugacy coupling and the implementation of the probabilistic primitives. Making this formal requires lifting the logical relation along a probability monad, replacing membership in an admissible relation with a coupling lifting [16] and re-establishing anti-reduction and compositionality for the lifted relation (e.g. via entropy-passing semantics [9, 43]), which we leave to future work.

In contrast, the three dynamic thread handlers from §2.3 are not representation independent of one another because they differ in their scheduling strategies. The child-biased and parent-biased handlers *choose* different orderings for the threads, so while each is a sound interpretation of the operations, they are not observationally equivalent. Theorem 4.10 can still be used to relate handlers with the same scheduling strategy differing in the representation of $\$tid$. For example, a child-biased handler that instead passes the parent thread an arbitrary integer with instantiation $[\$tid \rightarrow \text{Int}]$ is observationally equivalent to the child-biased handler of §2.3 that uses `Unit`, since the client cannot inspect the thread ID.

```
[$tid → Int] handler {
  ...
  fork _ k → k (inr ()) ; k (inl 57)
}
```

4.5.2 Equational Properties of Parametrized Handlers. A standard use of effect handlers is to verify implementations against the equational laws of the effect theory. With local state, initializing a reference then immediately overwriting it should be equivalent to initializing it to the latter value.

$$\text{ref}(v; a.\text{set}((a, w); ().c)) \approx \text{ref}(w; a.c)$$

Under a function-based handler, both sides reduce to a function with a pointing to w , and the equivalence follows.

Equations where a different number of resources are allocated on each side are subtler. The *garbage collection law* states that allocating a reference and never using it is equivalent to having never allocated it.

$$\text{ref}(v; a.c) \approx c \quad (a \notin \text{fv}(c))$$

Under the function-based handler, the two sides produce functions that differ at a . A proof would have to show that this entry is unobservable, and parametricity provides exactly this argument. We demonstrate the law for the fresh effect.

Example 4.12. Let $\Theta; \Psi, \psi; \Gamma \vdash c : A ! \Sigma$, where $\Sigma \supseteq \{\text{fresh} : \text{Unit} \rightarrow \psi, \text{eq} : \psi \times \psi \rightarrow \text{Bool}\}$, $\psi \notin \text{fpv}(A, \Gamma)$, and $a \notin \text{fv}(c)$. Then

$$\text{with } \{\psi \mapsto \text{Int}\}, h_\nu \text{ handle } c \approx \text{with } \{\psi \mapsto \text{Int}\}, h_\nu \text{ handle } \text{fresh}((); a.c),$$

where h_ν is the integer parameter passing fresh effect handler from §2.1.

PROOF. Elaborating each side, the existential layer reduces to a plain handle, and the **finally** clause of h_v feeds the handler's output f to the application $f\ 0$. On the right, the fresh command is handled, incrementing the counter by one. As $a \notin \text{fv}(c)$, each side respectively reduces to

$$c_1 = \text{do } f \leftarrow (\text{with } \bar{h}_v^\dagger \text{ handle } c^\dagger) \text{ in } f\ 0, \quad c_2 = \text{do } f \leftarrow (\text{with } \bar{h}_v^\dagger \text{ handle } c^\dagger) \text{ in } f\ 1,$$

where $\bar{h}_v$ is the handler without the **finally** clause, typed at $A ! \Sigma \Rightarrow (\psi \rightarrow A ! \Sigma') ! \Sigma'$ with $\Sigma' = \Sigma \setminus \{\text{fresh}, \text{eq}\}$. So the problem reduces to showing $c_1 \approx c_2$.

We let $\rho : \Theta \cup \Psi \rightarrow \mathbf{Rel}$ and choose the admissible relation $R = \{(i, i+1)_n \mid i, n \in \mathbb{N}\}$. This choice of R maintains the relatedness of the handlers much like in Theorem 4.11. The **return** clause produces a function constant in the parameter argument, fresh maps the related pair $(i, i+1)$ to the related pair $(i+1, i+2)$, and eq is invariant because $+1$ is injective in both directions. So $(\bar{h}_v^\dagger, \bar{h}_v^\dagger) \in \llbracket A ! \Sigma \Rightarrow (\psi \rightarrow A ! \Sigma') ! \Sigma' \rrbracket_{\rho[\psi \mapsto R]}^\approx$.

Let $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket_\rho^\approx = \llbracket \Gamma \rrbracket_{\rho[\psi \mapsto R]}^\approx$, since $\psi \notin \text{fpv}(\Gamma)$. By the fundamental theorem (Theorem 4.2) applied in both directions (inverting $\rho[\psi \mapsto R]$ for the reverse),

$$((\text{with } \bar{h}_v^\dagger \text{ handle } c^\dagger)[\gamma_1], (\text{with } \bar{h}_v^\dagger \text{ handle } c^\dagger)[\gamma_2]) \in \llbracket (\psi \rightarrow A ! \Sigma') ! \Sigma' \rrbracket_{\rho[\psi \mapsto R]}^\approx.$$

Since $(0, 1) \in R$ at every step index, the function relation and bind lemma (Theorem 4.3) give

$$(c_1[\gamma_1], c_2[\gamma_2]) \in \llbracket A ! \Sigma' \rrbracket_{\rho[\psi \mapsto R]}^\approx = \llbracket A ! \Sigma' \rrbracket_\rho^\approx,$$

since $\psi \notin \text{fpv}(A, \Sigma')$. Then anti-reduction lifts this to the original computations, and adequacy to contextual equivalence in CAMBRIA. $\square$

Parametricity and Reachability Analysis. Theorem 4.12 states that allocating an unused reference is observationally undetectable. However, the function-based handler of §2.1 retains the entry for that reference forever, since nothing signals when a reference has gone out of scope. While observationally harmless, it is a form of memory leak because the value persists throughout the entire body of the handle. This behaviour is unavoidable. A standard free theorem implies that naive liveness detection and parametricity are incompatible.

COROLLARY 4.13 (VACUOUSNESS [42]). *Any closed CAMBRIA function of type $\forall \alpha. (\alpha \rightarrow A ! \emptyset)$ where α is not free in A is extensionally constant. It returns the same value (or diverges) for every input.*

Any garbage-collection scheme that determines liveness from references occurring in the captured continuation requires a primitive that accepts the continuation as an arbitrary value (to work uniformly across handler instantiations), and produces an output with type independent of the polymorphic input. For example, the following are the type signatures for the primitives `isReachable`, which computes whether a reference `Ref` is still reachable in a continuation, `liveRefs`, which finds the live references in a continuation, and `onUnreachable`, which registers a finalizer to fire when a reference becomes unreachable.

```
isReachable   :  $\forall a. a \rightarrow (\text{Ref} \rightarrow \text{Bool} ! \{\}) ! \{\}$ 
liveRefs      :  $\forall a. a \rightarrow \text{List Ref} ! \{\}$ 
onUnreachable :  $\forall a. a \rightarrow (\text{Ref} * (\text{Unit} \rightarrow \text{Unit} ! \{\}) \rightarrow \text{Unit} ! \{\}) ! \{\}$ 
```

By Theorem 4.13, such primitives cannot perform non-trivial liveness analysis.

Reachability-based garbage collection for parametric local state is thus beyond the scope of this paper. Substructural typing, regions, and built-in reference effects remain possible alternatives, which we discuss in §7.

5 Type Inference

So far we have established CAMBRIA's type safety and parametricity. We now show that parametrized handlers are also practical, introducing only a minimal annotation burden. Since parameters abstract concrete types within the body of a handle, annotations are necessary to indicate which types are intended to be abstracted. The main result of this section, Theorem 5.5, is that it suffices to annotate only the operation arities that introduce new parameters. The type checker can infer everything else.

Inference for CAMBRIA follows Algorithm W for Hindley–Milner type systems [10, 27], extended to effects [32]. The inference algorithm uses *row variables* [21, 23], which may appear at the end of effect signatures and are required to encode operations not yet observed at that program point. The algorithm traverses the syntax tree, unifying value types and effect rows eagerly, and generalizing at each `Do` binding. There is no value restriction at generalization, which is sound in this setting [20], because only variables not free in the environment or effect row are generalized, and recursive functions are monomorphic. An operation invocation `!op v` adds `op` to the effect row with fresh arity, unifying the input with the type of `v` and the output with the type of `!op v`. Using an effect declaration (§3.2) unifies the declared arity with the operation in the effect row, if it exists. Handler clauses are checked as in the declarative `HANDLER` rule, with the shared answer type and output row found by unification. The forwarding side condition $\Sigma \setminus \Sigma'' \subseteq \Sigma'$ is discharged by unifying the output row with the input row variable and unhandled operations, rather than a separate subset test.

The only construct that departs from established inference is `PARHANDLE`. Parameters only enter the body through type or effect declarations. Unification, denoted $\doteq$, treats each parameter as an additional rigid constant, distinguishing them from unification variables. That is, a parameter only unifies with itself and fails against any other type, including a distinct parameter. At a handle term, the algorithm applies σ to the inferred body type and unifies the result against the handler's input type. The no-escape condition $\text{dom}(\sigma) \cap \text{fpv}(\underline{D}, \Gamma) = \emptyset$ of `PARHANDLE` is an additional side condition ensuring no parameter has leaked into the context or result type via unification. We write $\Theta; \Psi; \Gamma \vdash_w t : X$ to indicate the algorithm infers X for t .

$$\frac{\begin{array}{c} \Theta; \Psi; \Gamma \vdash_w v : \underline{C}_h \Rightarrow \underline{D} \quad \Theta; \Psi, \text{dom}(\sigma); \Gamma \vdash_w c : \underline{C}_c \quad [\Theta; \Psi \vdash A]_{A \in \text{cod}(\sigma)} \\ \underline{C}_c[\sigma] \doteq \underline{C}_h \quad \text{dom}(\sigma) \cap \text{fpv}(\underline{D}, \Gamma) = \emptyset \end{array}}{\Theta; \Psi; \Gamma \vdash_w \text{ with } \sigma, v \text{ handle } c : \underline{D}} \text{ PARHANDLE}$$

This is the new machinery in total. The remaining inference is ordinary Algorithm W with the parameters treated as constants. The algorithm is syntax-directed and terminating, hence decidable, and unifying value types and effect rows has principal solutions, being first-order unification [10] and row unification [21, 23] with parameters as constants.

We establish *principality* of CAMBRIA by showing soundness and completeness of the inference algorithm. Soundness states that every type inferred by the algorithm is derivable in the declarative system. Hence, the algorithm accepts only well-typed programs that have the type-safety and parametricity guarantees. Completeness, treated next, characterizes the annotations for which the algorithm infers the most general derivable type for a computation.

THEOREM 5.1 (SOUNDNESS). *If $\Theta; \Psi; \Gamma \vdash_w c : \underline{C}$, then $\Theta; \Psi; \Gamma \vdash c : \underline{C}$.*

PROOF. CAMBRIA type inference differs from Hindley–Milner with effect handlers only in the handle rule generalized to the `PARHANDLE` rule. Since parameters are rigid, unification remains sound, and all remaining cases follow from established proofs [10, 20, 32].

For `PARHANDLE` (with σ, v handle c), the IH on v gives a derivable handler type $\underline{C}[\sigma] \Rightarrow \underline{D}$. The IH on c gives a derivable body type $\underline{C}$, with $\text{dom}(\sigma)$ rigid in Ψ . Unifying $\underline{C}[\sigma]$ against the handler's input

restricts them to be equal, where the types in $\text{cod}(\sigma)$ are well-formed. For the no-escape condition, the algorithm explicitly checks that $\text{dom}(\sigma) \cap \text{fpv}(\underline{D}, \Gamma) = \emptyset$, so $\Theta; \Psi; \Gamma \vdash$ with σ, v handle $c : \underline{D}$ is derivable. $\square$

We write $\underline{C} \sqsubseteq \underline{C}'$ when $\underline{C}$ arises from $\underline{C}'$ by substituting value types for its type variables and *closed* signatures for its row variables. Closing the row variables recovers the closed-row typing of §3. We develop completeness in three stages. The first states that any term that does not introduce a parameter is inferable without any annotations because inference is exactly Algorithm W with effect handlers.

LEMMA 5.2 (GROUND COMPLETENESS). *Let every handle expression in c have $\sigma = \emptyset$. If $\Theta; \Psi; \Gamma \vdash c : \underline{C}$, then $\Theta; \Psi; \Gamma \vdash_w c : \underline{C}'$ with $\underline{C} \sqsubseteq \underline{C}'$.*

PROOF. Inference over this fragment is identical to Algorithm W with effects, for which completeness has been established [10, 32]. $\square$

In the full parametrized system, an *annotated term* is one in which every handle expression carries an explicit type annotation on the body.

with σ, v handle $(c : \underline{C})$

LEMMA 5.3 (PARAMETRIZED COMPLETENESS). *Let c be an annotated term. If $\Theta; \Psi; \Gamma \vdash c : \underline{C}$, then $\Theta; \Psi; \Gamma \vdash_w c : \underline{C}'$ with $\underline{C} \sqsubseteq \underline{C}'$.*

PROOF. By structural induction on the typing derivation. All cases apart from PARHANDLE follow the Hindley–Milner argument for algebraic effects [10, 32]. Since parameters are rigid and never generalized, unification and generalization arguments are unaffected. In particular, gen-admissibility [10] still holds, so Do is unchanged.

The only interesting case is PARHANDLE: with σ, v handle $(c : \underline{C})$. By the IH, the algorithm infers a handler type $\underline{C}' \Rightarrow \underline{D}'$ such that $\underline{C}[\sigma] \Rightarrow \underline{D} \sqsubseteq \underline{C}' \Rightarrow \underline{D}'$. The body is then checked against the annotation $\underline{C}$ with $\text{dom}(\sigma)$ rigid in Ψ , and the algorithm unifies $\underline{C}[\sigma] \doteq \underline{C}'$. Since unification returns the most general solution, the final inferred output type $\underline{D}''$ satisfies $\underline{D} \sqsubseteq \underline{D}''$. $\square$

The annotation in Theorem 5.3 uses the full type $\underline{C}$ of the body. We strengthen this to a *relevant* annotation consisting only of the operations that mention the introduced parameters. This result relies on operation arities being monomorphic (§3). Additional annotations would be required for universally quantified arities, and we leave the interplay of polymorphic and parameter annotations to future work.

Definition 5.4. Given a parameter substitution σ , the σ -relevant operations of Σ are

$$\text{rel}_\sigma(\Sigma) = \{ \text{op} : A_{\text{op}} \rightarrow B_{\text{op}} \in \Sigma \mid \text{dom}(\sigma) \cap (\text{fpv}(A_{\text{op}}) \cup \text{fpv}(B_{\text{op}})) \neq \emptyset \}.$$

A *relevantly annotated term* is one in which each handle expression carries an annotation listing only the σ -relevant operations, with σ, v handle $(c : \text{rel}_\sigma(\Sigma))$. The return type and all other operations are omitted. Note that this partial annotation is possible via effect declarations (§3.2).

THEOREM 5.5 (RELEVANT COMPLETENESS). *Let c be a relevantly annotated term. If $\Theta; \Psi; \Gamma \vdash c : \underline{C}$, then $\Theta; \Psi; \Gamma \vdash_w c : \underline{C}'$ with $\underline{C} \sqsubseteq \underline{C}'$.*

PROOF. We extend the proof of Theorem 5.3, strengthening the PARHANDLE case to show that from the relevant annotations, the algorithm infers a type at least as general as the body's declarative type. The handle introduces $\text{dom}(\sigma)$ into Ψ as rigid constants, and the operations split into two kinds:

- (1) σ -relevant. These arities are supplied by the annotation, so they are typed exactly as in Theorem 5.3.
- (2) σ -free. For any such $\text{op} : A_{\text{op}} \rightarrow B_{\text{op}}$, we have $\text{dom}(\sigma) \cap (\text{fpv}(A_{\text{op}}) \cup \text{fpv}(B_{\text{op}})) = \emptyset$, so σ is the identity on the arity, and the op is inferred as in the unparametrized system with no annotation.

Return type. A value or type mentioning $\psi \in \text{dom}(\sigma)$ can enter the body only through a σ -relevant operation. Hence, once the σ -relevant operations are annotated, every ψ is fixed wherever it flows. The return type of the body is then inferred with the parameters acting as fixed base types. $\square$

Thus, the annotation burden is minimal. The local-state example of §2.1 is annotated only with the three σ -relevant arities `!get` : `$loc` $\leadsto$ Int, `!set` : `$loc` * Int $\leadsto$ Unit, and `!ref` : Int $\leadsto$ `$loc`, which is enough to infer the type of the whole program. Moreover, the relevant annotation is a term-independent sufficient condition. Operations in $\text{rel}_\sigma(\Sigma)$ may be omitted from annotations when ψ propagates to their arity via term-level flow. For example, annotating only `!set` in

```
effect !set : $loc * Int  $\leadsto$  Unit.
do x  $\leftarrow$  !ref 3 in !set (x, 4)
```

permits the algorithm to infer the arity of `!ref` is Int $\leadsto$ `$loc`. For this reason, CAMBRIA enforces no constraints on which operations need to be annotated, letting the programmer manage the interface with individual effect declarations.

6 Related Work

Algebraic effects and handlers. Algebraic effects were introduced by Plotkin and Power [30] and equipped with handlers by Plotkin and Pretnar [31]. They are now incorporated in numerous languages, such as Eff [4], Koka [22], Frank [25], Links [17], and OCaml 5 [36]. Handlers in OCaml 5 are widely deployed, but the language does not track effect use in its type system; the focus is rather on handlers as a control mechanism. CAMBRIA is closest in spirit to the fine-grained call-by-value calculus of Kammar et al. [20], whose type-safety results we extend to the parametrized setting.

Parametrized algebraic theories. CAMBRIA is inspired by *parametrized algebraic theories* [37, 38] (PATs), where operations may bind and use abstract parameters, giving an algebraic account of effects that allocate resources. PATs have been used to axiomatize a range of effects, including local state [38], code jumps [15], the Beta-Bernoulli process [40], quantum measurement [39], and scoped effects [24]. CAMBRIA complements this line of work by realizing it in the algebraic effects and handlers framework, with parametricity enforcing the abstraction layer. Concretely, an operation returning a parameter, such as `!fresh` : Unit $\leadsto$ `$name`, is the *generic effect* [30] of a PAT operation that binds that parameter in its continuation. Another recent realization, Paella [35], is an Idris 2 library that models dynamically allocated resources through a Kripke possible-world semantics, where types are indexed by the resources currently allocated. CAMBRIA addresses the same setting as a stand-alone calculus where parameters are abstract types erased at runtime, and it establishes parametricity rather than providing a semantic framework.

Effect instances and named handlers. The *problem of instances*, where effects dynamically allocate resources, has motivated a substantial line of work. Eff originally had first-class effect instances [4], but removed them because the dynamic correspondence between operations and handlers complicated reasoning. Subsequent calculi recovered instances by a range of methods, including dynamically allocated labels [7, 12] and capability passing [8]. As a practical language, Koka [21] uses *named handlers* [44, 45], which expose each instance as a first-class name passed as evidence at runtime. This can model dynamically allocated heaps with first-class references, and

rank-2 polymorphism scopes the instances to prevent leakage. In each of the approaches above, the instance is a concrete object in the program. CAMBRIA takes a different approach, separating dispatch from instances. A handler interprets all instances of an effect, and the resource is exposed only as an abstract *parameter* that the handler instantiates. Thus, CAMBRIA guarantees *representation independence* over the user-defined resource. Moreover, there is no type-directed reduction, and since a parameter is a genuine type rather than a witness bound to an instance, a single handler can coordinate distinct instances, as in the dynamic threads (§2.3) and jump [15] effects.

Parametricity and logical relations. Parametricity [33] is the source of CAMBRIA’s abstraction guarantees, stemming from reading parameters as existential types [29] and data abstraction as representation independence [28]. We establish this through a step-indexed logical relation [3], which was extended to recursive and quantified types by Ahmed [1], and which we adapt to handlers. This has been done biorthogonally [5] by coupling the computation relation to the evaluation contexts, and abstraction over effect signatures has separately been obtained through a type discipline [6]. We move the abstraction to resource identifiers and take the direct approach, where each type contributes an independent clause and new features are local.

Reasoning about dynamic allocation. Relational models with dynamic allocation often require Kripke logical relations indexed by possible worlds, such as in generative, state-dependent ADTs [2], or fresh-name generation in the ν -calculus [47]. CAMBRIA has similar guarantees for local state and fresh-name effects (§2.1) *without* appealing to possible worlds. The relational interpretation of the parameter type plays the role that worlds play for a primitive full mutable heap. Complementary to our type-based guarantees are program logics for handlers, such as the Hazel separation logic [11] and its relational counterpart Blaze [13], which, on a per-program basis, verify properties of handler programs over real mutable state and, for Blaze, concurrency.

Concurrent effects. Kammar et al. [18] develop an equational theory for dynamic threads as an algebraic effect, using concrete thread IDs without type abstraction. CAMBRIA’s parameter variables make thread IDs abstract, giving parametricity guarantees and idiomatic handlers (§2.3) for concurrent programs.

7 Conclusion

We have presented CAMBRIA, a calculus that extends algebraic effects and handlers with parameters to provide type abstraction for resource-allocating effects. Parameter variables are first-class in the type system but play no role at runtime. A step-indexed logical relation establishes parametricity, where client code cannot inspect the concrete representation of the parameter outside of the effect interfaces. The decomposition of parametrized handlers into ordinary handlers and existential types means that the surface language stays ergonomic while the metatheory reduces to standard, modular pieces.

We have demonstrated CAMBRIA’s expressiveness with examples including local state, Pólya’s urn, and concurrent thread management. The calculus is type safe and backwards compatible, and type inference stays practical, as demonstrated in the implementation, with only effect arities mentioning parameters needing annotations. These results are backed by a mechanization in the Rocq prover. In the dynamic threads example, the thread IDs go beyond standard instances by coordinating concurrent processes. Yet they remain abstract, providing a representation-independence guarantee.

Future work. Natural extensions include substructural parameters to incorporate quantum effects [39] and scoped effects [24]. Such parameters could also address the reachability limitation of §4.5.2, alongside built-in references and region-based memory management [26, 41]. A probabilistic semantics would also allow internalization of de Finetti-style representation equivalences like

the one between the two Pólya urn handlers (§2.2). We are also interested in whether parameter abstraction offers an alternative to tunnelling [46], preventing effect-polymorphic code from accidentally handling effects it is unaware of. Other directions include refining the type system to be bidirectional [14, 25], and a denotational semantics to establish the categorical structure of parametrized handlers.

References

- [1] Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proc. ESOP '06*. https://doi.org/10.1007/11693024_6
- [2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *Proc. POPL '09*. <https://doi.org/10.1145/1480881.1480925>
- [3] Andrew Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* (2001). <https://doi.org/10.1145/504709.504712>
- [4] Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *J. Log. Algebr. Methods Program.* (2015). <https://doi.org/10.1016/j.jlamp.2014.02.001>
- [5] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. In *Proc. POPL '18*. <https://doi.org/10.1145/3158096>
- [6] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. In *Proc. POPL '19*. <https://doi.org/10.1145/3290319>
- [7] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In *Proc. POPL '20*. <https://doi.org/10.1145/3371116>
- [8] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. In *Proc. OOPSLA '20*. <https://doi.org/10.1145/3428194>
- [9] Ryan Culpepper and Andrew Cobb. 2017. Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In *Proc. ESOP '17*. https://doi.org/10.1007/978-3-662-54434-1_14
- [10] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proc. POPL '82*. <https://doi.org/10.1145/582153.582176>
- [11] Paulo Emilio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. In *Proc. POPL '21*. <https://doi.org/10.1145/3434314>
- [12] Paulo Emilio de Vilhena and François Pottier. 2023. A Type System for Effect Handlers and Dynamic Labels. In *Proc. ESOP '23*. https://doi.org/10.1007/978-3-031-30044-8_9
- [13] Paulo Emilio de Vilhena, Simcha van Collem, Ines Wright, and Robbert Krebbers. 2026. A Relational Separation Logic for Effect Handlers. In *Proc. POPL '26*. <https://doi.org/10.1145/3776676>
- [14] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *CSUR* (2021). <https://doi.org/10.1145/3450952>
- [15] Marcelo Fiore and Sam Staton. 2014. Substitution, jumps, and algebraic effects. In *Proc. LICS '14*. <https://doi.org/10.1145/2603088.2603163>
- [16] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in Higher-Order Separation Logic. In *Proc. POPL '24*. <https://doi.org/10.1145/3632868>
- [17] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proc. TyDe '16*. <https://doi.org/10.1145/2976022.2976033>
- [18] Ohad Kammar, Jack Liell-Cock, Sam Lindley, Cristina Matache, and Sam Staton. 2026. An equational axiomatization of dynamic threads via algebraic effects. In *Proc. POPL '26*. <https://doi.org/10.1145/3776706>
- [19] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proc. ICFP '13*. <https://doi.org/10.1145/2500365.2500590>
- [20] Ohad Kammar and Matija Pretnar. 2017. No Value Restriction is Needed for Algebraic Effects and Handlers. *J. Funct. Program.* (2017). <https://doi.org/10.1017/S0956796816000320>
- [21] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proc. MSFP '14*. <https://doi.org/10.4204/EPTCS.153.8>
- [22] Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proc. POPL '17*. <https://doi.org/10.1145/3009837.3009872>
- [23] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proc. TLDI '12*. <https://doi.org/10.1145/2103786.2103798>
- [24] Sam Lindley, Cristina Matache, Sean Moss, Sam Staton, Nicolas Wu, and Zhixuan Yang. 2024. Scoped Effects as Parameterized Algebraic Theories. In *Proc. ESOP '24*. https://doi.org/10.1007/978-3-031-57262-3_1

- [25] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proc. POPL '17*. <https://doi.org/10.1145/3009837.3009897>
- [26] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *Proc. ICFP '24*. <https://doi.org/10.1145/3674642>
- [27] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978). [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- [28] John C. Mitchell. 1986. Representation Independence and Data Abstraction. In *Proc. POPL '86*. <https://doi.org/10.1145/512644.512669>
- [29] John C. Mitchell and Gordon Plotkin. 1988. Abstract Types Have Existential Type. *TOPLAS* (1988). <https://doi.org/10.1145/44501.45065>
- [30] Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categ. Struct.* (2003). <https://doi.org/10.1023/A:1023064908962>
- [31] Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proc. ESOP '09*. https://doi.org/10.1007/978-3-642-00590-9_7
- [32] Matija Pretnar. 2014. Inferring Algebraic Effects. *Log. Methods Comput. Sci.* (2014). [https://doi.org/10.2168/LMCS-10\(3:21\)2014](https://doi.org/10.2168/LMCS-10(3:21)2014)
- [33] John Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Proc. IFIP Congress '83*.
- [34] Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Proc. ESOP '19*. https://doi.org/10.1007/978-3-030-17184-1_13
- [35] Jesse Sigal, Ohad Kammar, Cristina Matache, and Conor McBride. 2024. Paella: Algebraic Effects with Parameters and their Handlers. In *Proc. HOPE '24 Workshop at ICFP*. Slides available on author's webpage..
- [36] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proc. PLDI '21*. <https://doi.org/10.1145/3453483.3454039>
- [37] Sam Staton. 2013. An Algebraic Presentation of Predicate Logic (Extended Abstract). In *Proc. FoSSaCS '13*. https://doi.org/10.1007/978-3-642-37075-5_26
- [38] Sam Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *Proc. LICS '13*. <https://doi.org/10.1109/LICS.2013.58>
- [39] Sam Staton. 2015. Algebraic Effects, Linearity, and Quantum Programming Languages. In *Proc. POPL '15*. <https://doi.org/10.1145/2676726.2676999>
- [40] Sam Staton, Dario Stein, Hongseok Yang, Nathanael Ackerman, Cameron Freer, and Daniel Roy. 2018. The Beta-Bernoulli process and algebraic effects. In *Proc. ICALP '18*. <https://doi.org/10.4230/LIPIcs.ICALP.2018.141>
- [41] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* (1997). <https://doi.org/10.1006/inco.1996.2613>
- [42] Philip Wadler. 1989. Theorems for free!. In *Proc. FPCA '89*. <https://doi.org/10.1145/99370.99404>
- [43] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. 2018. Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion. In *Proc. ICFP '18*. <https://doi.org/10.1145/3236782>
- [44] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. In *Proc. ICFP '20*. <https://doi.org/10.1145/3408981>
- [45] Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-Class Names for Effect Handlers. In *Proc. OOPSLA '22*. <https://doi.org/10.1145/3563289>
- [46] Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. In *Proc. POPL '19*. <https://doi.org/10.1145/3290318>
- [47] Yu Zhang and David Nowak. 2003. Logical Relations for Dynamic Name Creation. In *Proc. CSL '03*. https://doi.org/10.1007/978-3-540-45220-1_48