Parallel Algebraic Effect Handlers

NINGNING XIE*, University of Toronto, Canada and Google DeepMind, Canada
DANIEL D. JOHNSON*, University of Toronto, Canada and Google DeepMind, Canada
DOUGAL MACLAURIN, Google DeepMind, USA
ADAM PASZKE, Google DeepMind, Germany

Algebraic effect handlers support composable and structured control-flow abstraction. However, existing designs of algebraic effects often require effects to be executed sequentially. This paper studies parallel algebraic effect handlers. In particular, we formalize λ^p, a lambda calculus which models two key features, effect handlers and parallelizable computations, the latter of which takes the form of a **for** expression, inspired by the Dex programming language. We present various interesting examples expressible in our calculus. To show that our design can be implemented in a type-safe way, we present a higher-order polymorphic lambda calculus F^p that extends λ^p with a lightweight value dependent type system, and prove that F^p preserves the semantics of λ^p and enjoys syntactic type soundness. Lastly, we provide an implementation of the language design as a Haskell library, which mirrors both λ^p and F^p and reveals new connections to free applicative functors. All examples presented can be encoded in the Haskell implementation. We believe this paper is the first to study the combination of user-defined effect handlers and parallel computations, and it is our hope that it provides a basis for future designs and implementations of parallel algebraic effect handlers.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Polymorphism**; **Functional languages**; **Semantics**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Effect handlers, Parallelism, Type systems

ACM Reference Format:
Ningning Xie, Daniel D. Johnson, Dougal Maclaurin, and Adam Paszke. 2024. Parallel Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 8, ICFP, Article 262 (August 2024), 33 pages. https://doi.org/10.1145/3674651

1 Introduction

Algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2009] allow programmers to define structured control-flow abstraction in a flexible and composable way. Since introduced, they have been studied extensively in the community, supported in languages including Koka [Leijen 2014], Eff [Pretnar 2015], Frank [Lindley et al. 2017], Links [Lindley and Cheney 2012], and Effekt [Brachthäuser et al. 2020]. Recent work has implemented effect handlers in Multicore OCaml [Sivaramakrishnan et al. 2021] to support asynchronous I/O for concurrent programming.

As an example of effect handlers, consider the monadic encoding of the `state` effect [Kammar and Pretnar 2017] using the syntax of an untyped algebraic effect lambda calculus[1]:

*Both authors contributed equally to this research.
[1]For clarity, we use `x ← e1; e2` as a shorthand for `(λx. e2) e1`, and `e1; e2` for `(λ_. e2) e1`.

Authors' Contact Information: Ningning Xie, University of Toronto, Toronto, Canada and Google DeepMind, Toronto, Canada, ningningxie@cs.toronto.edu; Daniel D. Johnson, University of Toronto, Toronto, Canada and Google DeepMind, Toronto, Canada, ddjohnson@google.com; Dougal Maclaurin, Google DeepMind, Boston, USA, dougalm@google.com; Adam Paszke, Google DeepMind, Berlin, Germany, apaszke@google.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/8-ART262
https://doi.org/10.1145/3674651

```
handle { get ↦ λx.λk. (λs. k s s), set ↦ λx.λk. (λs. k () x),
         return ↦ λx. (λs. x) }
       (perform set 21; x ← perform get ();  (x + x) ) 0 // 42
```

Here a **handler** takes a list of operation clauses, a `return` clause that wraps the final result, and a computation to be handled. Inside each operation clause, `x` is the argument to the operation, `k` is the *resumption* captured by the handler, and each operation returns a function that dictates the evolution of state `s`. The handled computation sets the state to `21`, retrieves it using `get`, and doubles it. The initial state `s` is set to `0`. Evaluating the program produces the result `42`.

This example clearly shows that the use of algebraic effects generally introduces sequential dependencies between evaluation of different expressions. Indeed, the effect of **perform** `set 21` must take place before **perform** `get ()`, or else the evaluation result would change.

In this work, we are interested in exploring *parallel effect handlers* that relax this sequential dependency. We allow the user to scope subexpressions in ways that make them independent of each other, and show that there are a number of practically useful effect handlers that can be made to preserve this independence even in the presence of effects. This, in turn, opens up the possibility of parallel evaluation strategies of effectful programs.

Our inspiration is the recent work on Dex [Paszke et al. 2021], a strict functional programming language for array programming, which has shown that it is *possible* and *useful* to define parallel effect handlers. Specifically, Dex supports a built-in effect `Accum` which is similar to the regular state effect, but more limited. The state can only be updated through an (infix) associative increment operation `(+=)` and the state is implicitly initialized with an identity element of the increment. Using the syntax of Dex, we can write the following program that sums an input array (we explain the syntax in more detail in §2.2):

```
sum = λx:(Fin n⇒Int). snd (runAccum λy. for i:n. y += x.i)
```

In Dex, the **for** expression builds an array. Interestingly, the Dex compiler is able to evaluate the different steps of the **for** in parallel, even though they all have effects. This is exactly thanks to the restrictions induced by `Accum`: (1) associativity enables reassociation of different increments, enabling splitting work into separate subunits and (2) there is no "read" operation in `Accum`, so state cannot be retrieved until `runAccum` is complete. In particular, the effects of an increment in one iteration cannot be observed in other iterations. A careful reader might already notice the connections between this effect and the `Accy` applicative functor of McBride and Paterson [2008], a connection which we explore further in §2.3.

But, so far `Accum` is the only effect in Dex that inhibits parallelization, and the current design of Dex does not provide a clear path to extending this feature to user-defined effects and handlers. Therefore, the key questions we ask in this paper are: *is it possible to support user-defined algebraic effect handlers that preserve independence between independent subexpressions, enabling parallel execution? If so, what are their semantics*? We offer the following contributions:

- We formalize λ^p, an untyped lambda calculus that models two key features (§4): effect handlers, and the parallelizable computations that take the form of the **for** construct. The untyped semantics demonstrates the essence of the interaction between the two features.
- We present a variety of illustrative examples of parallel effect handlers that are enabled by our design (§5), demonstrating how our design can be naturally applied to parallelize effectful programs that we believe are practical and useful.
- We present F^p, a System F_ω-style calculus that extends λ^p with a type system (§6), formalizing parallel effect handlers in a type-safe way. We prove that F^p preserves the semantics of λ^p (Thm. 6.3), and that F^p enjoys syntactic type soundness (Thm. 6.2 & 6.5).

- We implement our parallel effect handler system as a library in Haskell (§7), which closely mirrors the semantics and types of F^p while also supporting an interface to parallelism based on Haskell's applicative functors. All examples in §5 have corresponding Haskell implementations.

Lastly, we discuss alternative designs in §8, survey related work in §9, and conclude in §10. The complete set of rules and the proofs for stated lemmas and theorems are provided in the appendix.

2 Background

In this section, we present a brief overview of algebraic effect handlers (§2.1), and of two mechanisms for expressing parallel computations by enforcing independence (§2.2 and 2.3).

2.1 Algebraic Effect Handlers

Algebraic effect handlers provide a flexible and modular way to incorporate effects in programming languages. We review them through an example. Consider the following non-deterministic effect. It has a single operation `amb` that takes a unit and returns a boolean:

```
ndet { amb : () → Bool }
```

We can perform an operation by calling **perform** and providing an operation along with its argument, e.g. **perform** `amb ()`. The semantics of effects are provided separately as a handler. For example, here is a handler for `amb` that collects all possible results in a list.

```
hamb = { return ↦ λx. [x], amb ↦ λx. λk. (k True) ++ (k False) }
```

The `return` clause applies to the value returned from the computation being handled[2]. In this case, `return` wraps the result into a singleton list. The `amb` operation clause takes the operation argument `x` (in this case unit), and a resumption `k` that resumes the original computation with an operation result. The handler resumes `k` twice, and concatenates the results. We can use the handler to handle a computation that contains the `amb` operation:

```
handle hamb (x ← perform amb ();  y ← perform amb (); x && y)
```

The first operation `amb` will get handled by the handler, and the program evaluates to

```
(k True) ++ (k False)
```

where `k` is `λz.` **handle** `hamb (x ← z; y ←` **perform** `amb (); x && y)`. At this point, the program resumes `k` with `x` being `True` and `False`, respectively. Note that the handler `hamb` is reinstalled inside the continuation, and thus can handle further `amb` operations. Continuing evaluating this program, we will get the result

```
[True, False, False, False]
```

As a convenience feature, parameterized effect handlers [Plotkin and Power 2003] allow passing a local parameter to handlers, which can be updated when the resumption is resumed. Below shows the implementation of a state handler as a parameterized handler:

```
hstate = { return ↦ λs.λx. x, get ↦ λs.λx.λk. k s s, set ↦ λs.λx.λk. k x () }
```

Here, both the return clause and the operation clauses receive as an additional argument the current handler parameter `s`. The return clause simply returns the computation result. The `get` clause resumes the continuation with the handler parameter `s` and the operation result `s`, while `set` resumes with the new handler parameter `x` and the operation result unit. Now we can implement the program in the introduction in a more concise and efficient way. Note that with parameterized handlers, the handler also takes an initial parameter, in this case `0`.

[2] In practice, it is common to omit the return clause when it has the default implementation `return ↦ λx. x`.

```
handle hstate 0 (perform set 21; x ← perform get (); x + x) // 42
```

2.2 Parallelizing Effects With "*for*" Expressions

In the Dex array programming language [Paszke et al. 2021], parallel computations are expressed by means of a parallelizable **for** construct. For instance, the following program increments an array:

```
incr = λx:(Fin n⇒Int). for i:n. x.i + 1  // incr <1,2,3> = <2,3,4>
```

Here, `x` of type `Fin n⇒Int` is an array indexed by indices of type `Fin n` and containing elements of type `Int`. Retrieval of individual elements is possible using the `x.i` expression, which looks up an element of array corresponding to the index `i`. Denoting arrays using angle brackets ⟨·⟩, the program `incr ⟨1, 2, 3⟩` evaluates to the result `⟨2, 3, 4⟩`.

A key property of the **for** construct in Dex is that each element of the result array for a (non-effectful) **for** expression can be evaluated independently, because there is by construction no data dependence between the different values. This allows Dex to efficiently compile such expressions to execute on hardware accelerators. Extending this, Dex also supports a parallelism-friendly `Accum` effect (as we discuss in §1), which allows each loop iteration to make an additive contribution to special "reference" objects. This `Accum` effect is carefully designed so that programs such as `runAccum λy. for i:n. y += x.i` can still execute in parallel even in the presence of these effects. Unfortunately, Dex lacks an extensible effect system, and so its users are limited to a set of built-in implementations that the compiler can understand and compile.

Abstracting away the details of Dex's compilation strategy and syntax, the essence of Dex's approach to parallel programming with effects can be summarized as combining (1) a *parallelizable* **for** *construct* where the body `expr` in each expression `for i:n. expr` is by construction independent across iterations, (2) *arrays* and *array types* such as `⟨1, 2, 3⟩ :: Fin 3 ⇒ Int`, which are produced by **for** expressions and can be constructed and indexed into in parallel, and (3) a *built-in transformation* of **for** expressions for the `Accum` effect that preserves the independence between the **for** loop's iterations. In this work, we adopt the first two of these (along with their syntax), and generalize the third to allow parallelization across a wide set of user-definable effects.

2.3 Previous Approaches to Parallelization Through Independence

Interestingly, the idea of taking advantage of independence between effectful subexpressions to enable parallel execution has been explored before from a different lens, that of *applicative functors* [McBride and Paterson 2008]. Applicative functors are relaxations of monads defined in terms of a lifting function `pure :: a -> f a` and an application function `<*> :: f (a -> b) -> f a -> f b`. If we interpret the type `f a` as meaning "an effectful computation that produces a value of type `a`", the key feature of `<*>` is that, in the expression `x <*> y`, the effects performed by `y` cannot be affected by the results of `x` and vice versa, just like different iterations of Dex's **for** are independent by construction. This property of `<*>` has been previously used to automatically parallelize effectful computations in specific Haskell monads when those computations are written using applicative primitives [Marlow et al. 2014, 2016].

These two ways of denoting independent subexpressions (the Dex-inspired **for** construct and the applicative `<*>`) are, to some degree, equivalent. For instance, we can define versions of `pure` and `<*>` in terms of the **for** construct as

```
pure x ↦ (λ_. x)
x <*> y ↦ (λ_. outs ← (for i:2. if i == 0 then (x ()) else (y ())); outs.0 outs.1)
```

This means that, although in this work we express our semantics in terms of **for**, our work can also be interpreted as a way to incorporate applicative-style parallelism into effect systems, similar

to the rich connections between effect systems and monads [e.g. Forster et al. 2017; Kammar et al. 2013; Kiselyov and Ishii 2015; Kiselyov et al. 2013]. We will explore this connection further in §3.4.

3 Key Ideas

Before we describe our approach, we take a brief stop to rearticulate our goals in detail. We seek an effect system that has:

- *Extensiblility*. Users should be able to extend the system with new effects, rather than being limited to a set of built-in effects.
- *User-provided parallel semantics*. Users should be able to specify how these new effects should interact with parallelism (in the sense of independence of effects between select subprograms), instead of requiring all effects to use built-in parallel semantics.
- *Compositionality*. It should be possible to use multiple effects in the same program, and the semantics of their combination could be derived from their semantics in isolation.

Our language design satisfies all three goals. It features a novel formulation of *effect handlers*, as well as a syntax for *parallel subcomputations* in the form of **for** expressions. In §3.1 and 3.2 we focus on the untyped semantics, using a running example of the accumulation effect. Then, in §3.3, we give a brief overview of a type system suitable for our extension, and in §3.4 we discuss how our design can be integrated with the existing monads and applicative functors in Haskell.

3.1 The Challenge of Parallelizing Effect Handlers

Let us inspect what causes the traditional effect systems to be incompatible with parallel evaluation strategies. To do so, let us focus on a writer effect with a single `accum` operation:

```
writer { accum : Int → () }
hAccum = { return ↦ λs.λx. (x, s), accum ↦ λs.λx.λk. k (s + x) () }
```

The associated parameterized handler keeps track of the sum in the handler's local state. The `return` clause simply returns the computation result together with the parameter. When handling an `accum` operation, we update the state to `(s + x)`, and resume the computation with `()`. With this definition, we expect the following program to evaluate to `((), 3)`:

```
handle hAccum 0 (perform accum 1; perform accum 2)   // ((), 3)
```

The state is initialized to value 0, then gets incremented by 1 and then 2, and finally the `return` function is used to wrap the result of the second **perform** with accumulation result.

Note that the description above describes all the steps, as they happen *in sequence*, even though there are no data dependencies between the two operations. In particular, when evaluation reaches **perform** `accum 1`, the `hAccum` clause `accum` is called with the continuation `k = λs. λr.` **handle** `hAccum s (`**perform** `accum 2)`. The second **perform** is fully controlled by the operation invoked by the first one! And it can be invoked once, twice or even never. This leads to a problem if we wish to be able to run parts of the program in parallel: since the handler has complete control of the continuation, we are forced to evaluate the operations in sequence. There is no way for the user program to declare that these operations could be performed in parallel, and no way for the handler to make use of that parallelism.

Our goal in this work is to extend the existing calculi to allow for expressing the intent to sever the dependence between two effectful (but data-independent) subexpressions, avoiding unnecessary sequential dependencies due to the resumption continuation. This, in turn, makes it possible to evaluate them independently (and possibly in parallel), and then finally collapse the observable effects they cause back into the computation they were called from in a deterministic way.

3.2 Our Approach

Our proposal boils down to two extensions to the standard effect calculi. First, the syntax of our calculus grows to include a **for** expression, that makes the effects induced by the different instantiations of its body independent. Then, to specify how the different independent effects should reflect back into the context in which **for** is evaluated, every handler is now additionally responsible for implementing a `traverse` clause in addition to the `return` clause, which is responsible for combining the effects across these instantiations. In a sense, we treat **for** as an effectful operation itself, which must be handled by `traverse`. The traverse clause takes four parameters: (1) the number of subcomputations, (2) the current value of handler's state, (3) an array of *body continuations*, each of which encapsulate a single independent subexpression and reinstall the current handler in it, and (4) a resumption that returns to the program after the **for**. (Note that *every* handler must implement `traverse`, because executing a computation in parallel requires defining parallel semantics for *all* effectful operations that occur in the computation, not just a subset. We discuss a default implementation of `traverse` for handlers without special parallelism behavior in §4.2.)

To better illustrate our changes, let us return to the accumulation effect. Assuming the existence of helpers `reduce` and `unzip` for arrays (which we will describe shortly), our modified handler is

```
hAccum = { return ↦ ..., accum ↦ ...,  // the same as before
           traverse ↦ (λn.λs.λl.λk. pairs ← for i:n. l.i 0;
                                    (results, outs) ← unzip pairs;
                                    out ← reduce (+) outs;
                                    k (s + out) results) }
```

We illustrate our system in action by stepping through the evaluation steps of the following program, which sums up an array of values and then returns a constant string:

```
handle hAccum 0 (for i:3. perform accum (⟨1,2,3⟩.i); "done")
```

The first step is that the **for** expression is *handled* by the handler `hAccum`, just as an effect would be. In our design, all handlers must handle **for** expressions, with the innermost handler going first. Denoting the traverse clause from `hAccum` as `f_traverse`, we obtain:

```
f_traverse 3 0 ⟨ λs. handle hAccum s (perform accum ⟨1,2,3⟩.0),
                 λs. handle hAccum s (perform accum ⟨1,2,3⟩.1),
                 λs. handle hAccum s (perform accum ⟨1,2,3⟩.2)  ⟩ -- l
               (λs. λxs. handle hAccum s "done") -- k
```

Here `f_traverse` takes four arguments: (1) the array length `n`, in this case `3`, (2) the current handler parameter `s`, in this case `0`, (3) an array of *body continuations* `l` with one entry per value of `i` in the original program, and (4) the final resumption `k`. The resumption `k` is much like the original resumption when handling operations: it captures from where the **for** "effect" is performed, to where the handler is applied, with the handler reinstalled inside. With parameterized handlers, here `k` takes the updated parameter `s` and an array `xs` as the result of the **for** "effect", and resumes the program (which in this case produces a constant value `"done"`). The body continuation array `l` is unique to our design, and is the key to enable parallel effect handlers. It captures the **for** body, but additionally pushes the handler to be *inside* the body expression, and allows the traverse handler to pass different parameters to each body subexpression.

In this case, `f_traverse` begins by re-emitting a **for** expression to evaluate the loop bodies in parallel, but passes the identity element `(0)` as the handler parameter:

```
pairs ← for i:n. (⟨ λs. handle hAccum s (perform accum ⟨1,2,3⟩.0),
                    λs. handle hAccum s (perform accum ⟨1,2,3⟩.1),
```

```
λs. handle hAccum s (perform accum ⟨1,2,3⟩.2)  ⟩.i 0);
```

Since this **for** expression is no longer contained in any handler, we are free to execute each iteration in parallel, with the `accum` operations inside the **for** expression handled normally by the handler. As the return clause wraps the result into a pair, we thus obtain an array of pairs:

```
pairs ← ⟨((),1),((),2),((),3)⟩
```

Then, we use the standard `unzip` function that unzips `pairs` into `results` and `outs`, where `results` is an array of the computation results, while `outs` is an array of the accumulated handler parameters:

```
(results, outs) ← (⟨(),(),()⟩, ⟨1,2,3⟩)
```

We next sum up the array of handler outputs to a single value by applying `out ← reduce (+) outs`, obtaining the value `out ← 6`. Lastly, `(k (s + out) results)` resumes the computation with the new parameter and the results from the **for** expression:

```
(λs. λxs. handle hAccum s "done") 6 ⟨(),(),()⟩
```

which produces the final result `("done", 6)`.

3.3 Type-Checking Parallel Effect Handlers

So far we have seen untyped parallel effect handlers. It would be, of course, even better if we can implement such semantics in a type-safe manner, especially since it is known that effect handlers enjoy type safety when equipped with an effect system (e.g. Leijen [2014]; Pretnar [2015]). It turns out that giving static semantics to parallel effect handlers is trickier than one might expect.

Answer types. We first introduce the notion of *answer types* of a handler. In systems with *delimited continuations*, answer types refer to the types of values returned by contexts up to the delimiter [Danvy and Filinski 1990]. As handlers also provide a form of delimited control [Forster et al. 2017], we use answer types to mean the types of values returned by handlers, after the return clause has been applied. As an example, assume a reader effect with an operation `ask` that takes an unit and returns an integer. Then consider the following program:

```
handle { ask ↦ (λx. λk. k 42), return ↦ (λx. x + 1) } ((perform ask ()) + 100)
```

Here, the handler handles `ask` by resuming the continuation with `42`. The continuation adds `100` to the result, and finally the return clause adds `1`. Thus, the result of the program is `(42 + 100 + 1)`, namely `143`. In this case, we say that the answer type of the handler is *Int*. Moreover, we remark that this handler can *only* handle computations that produce an *Int*. Specifically, because of the use of `(+)` in the return clause, the computation result (as denoted by `x`) must be an integer.

*Handling **for** expressions with answer type constructors.* We can now discuss the static semantics of parallel effect handlers. Recall the reduction of the example program

```
handle hAccum 0 (for i:3. perform accum (⟨1,2,3⟩.i); "done")
```

from the previous section. How can we assign types to the various stages of the reduction? An interesting observation is that, at the top level, the program being handled has type `String`, so the answer type of the full program is `(String, Int)`. However, while applying the traverse clause, we obtained intermediate handler expressions such as **handle** `hAccum s (`**perform** `accum ⟨1,2,3⟩.0)`, where the expression now being handled has type `()` and the answer type is `((), Int)`, which is different than the original program. Moreover, a program may have multiple **for** expressions that construct arrays of different types, and we could expect each to produce a different type and thus a different answer types when the handler is pushed inside.

How should we represent this in the type system? We could require all **for** expressions to have the same type as the overall program being handled to prevent this problem, but that would be too restrictive to support interesting examples. Instead, our solution is to require parallel effect handlers to be *polymorphic*. Specifically, we associate each handler with an *answer type constructor* `f`, and require the return clause to map values of any type `a` into values of type `(f a)`. Importantly, this answer type constructor gives handlers partial knowledge about the answer type of the computation.

For instance, in the case of `hAccum`, if the computation returns a type `a`, the return clause (`return ↦ λs. λx. (x, s)`) gives answer type `(a, Int)`, so the answer type constructor is `f a = (a, Int)`. The `accum` clause (`accum ↦ λs. λx. λk. k (s + x) ()`)) is also polymorphic, preserving the answer type of the resumption `k`. Finally, the `traverse` clause can safely accumulate the accumulator results, all of which are of type `Int`, while remaining polymorphic over the type `a`. Thus, `hAccum` can handle computations in a well-typed way even when **for** expressions have different types.

On may wonder how expressive this design is. In particular, we cannot support the handler for `ask` above as a parallel effect handler. Nevertheless, it turns out that such design is sufficient for us to type-check a wide variety of useful parallel effect handlers, including all examples in §5.

Array types and value-dependent types. With answer type constructors, we are almost ready to present the typing rule for handlers. The only task left now is to decide how to type-check arrays. We can equip the type system with array types such as `(Fin n ⇒ Int)` for an integer array with length `n`, where `(Fin n)` is an index type denoting natural numbers less than `n`. This requires forms of *dependent types*, where expressions (`n`) can appear in types (`Fin n ⇒ Int`).

In this work, we feature a limited form of dependent types, *value-dependent types* [Swamy et al. 2011], that allow values to appear in types. As an example, the type `(x:Int) → (Fin x ⇒ Int)` denotes a function that takes an integer `x` and returns an integer array of length `x`. Value-dependent types integrate well with effect handlers, as we do not deal with effects on the type level. Combining effect handlers with more general forms of dependent types is possible (see, e.g. Ahman [2017]), which however is largely an orthogonal extension. With array types, we can now (partially) annotate the first expression in the traverse clause of `hAccum` as:

```
traverse ↦ λn:Int. λs. λl. λk. pairs ← l (for i:Fin n. 0); ...
```

where value-dependent types make it possible to use the integer argument `n` in the index type `Fin n` in the **for** expression.[3]

Typing `hAccum`. Putting all pieces together, we are now ready to present the types for handlers, using `hAccum` as the example. Recall that the answer type constructor for `hAccum` is `f a = (a, Int)`. We write simple `Int → t` rather than `(x: Int) → t` if `x` does not appear in `t`. The types for the clauses in `hAccum` are given below (where we omit effect annotations for simplicity):

```
return   : forall a. Int → a → (a, Int)
accum    : forall a. Int → Int → (Int → () → (a, Int)) → (a, Int)
traverse : forall a b. (n : Int) → Int → (Fin n ⇒ (Int → (b, Int))) --l
                                       → (Int → (Fin n ⇒ b) → (a, Int)) -- k
                                       → (a, Int)
```

Note that `traverse` is polymorphic over `a` and `b`, where `a` is the type of the full computation and `b` is the type of the body of the **for** expression. `traverse` takes four arguments: (1) the length `(n:Int)` forms a dependent type; (2) the second `Int` is the type of the handler parameter; (3) the `n`-length

[3]We remark the value dependent types allow a safe way to project out from an array. On the other hand, we may also use arrays with statically unknown length, where projecting out from an array returns a `Maybe`.

array `l` consists of body continuations which each map an `Int` parameter to a `(b, Int)` result; (4) lastly, the resumption `k` takes an updated handler parameter `Int` and an `n`-length array (the output of the **for**), and returns a wrapped result `(a, Int)`. The final result of `traverse` is thus also `(a, Int)`.

3.4 Embedding Parallel Effect Handlers into Haskell

We demonstrate the practicality of our approach by additionally providing a Haskell implementation of our system (§7), where effect computations are wrapped in a monad, denoted `PE effs`, building on the `Eff effs` monad and algebraic data type effect representation of Kiselyov and Ishii [2015]. Here, however, we run into an immediate challenge: how should we represent our *traverse* clause in Haskell's type system? In particular, since Haskell lacks a value-dependent type system[4] we use to formalize F^p, it is not straightforward to ensure that well-typed handlers preserve the length of their input array.

Our solution is to abstract away the exact type of the array using Haskell's typeclasses, and thus require handlers to be polymorphic over length. For this purpose, we can use Haskell's (conveniently-named) `Traversable` typeclass, which we excerpt below:

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative m => (a -> m b) -> t a -> m (t b)
```

The `traverse` typeclass method allows one to inspect each element of the structure `t` and modify it by means of an arbitrary monad or applicative functor `m`. Haskell's built-in list is an instance of `Traversable`, but instances can also be defined for e.g. fixed-length tuples `(a, a, a)`. This means that, in order for a function to work for *all* `Traversable t`, it must preserve the length of its input. We can use this to define a Haskell version of `hAccum`'s *traverse* clause from the previous section as

```
hAccumTraverse :: Traversable t => Int -> t (Int -> PE effs (b, Int))
                     -> (Int -> t b -> PE effs (a, Int)) -> PE effs (a, Int)
```

The only differences are that the effectful operations now occur under our `PE effs` monad, the `Fin n ⇒ x` types are now abstracted as `t x` for some `Traversable t`, and the argument `n` is removed since it can be inferred from the `t x` argument.

Interestingly, once we have abstracted the exact type of the array in this manner, it becomes straightforward to extend our system to support *heterogeneous* collections in addition to homogeneous arrays, by using a *rank-2* variant of Traversable originally proposed in the context of heterogeneously-typed parsing expression grammars [Blažević and Légaré 2017]:

```
class (Rank2.Functor t, Rank2.Foldable t) => Rank2.Traversable t where
    Rank2.traverse :: Applicative m => (forall b. p b -> m (q b)) -> t p -> m (t q)
```

A "wrapped list" such as `[p Int]` can be treated as a rank-2 traversable, but heterogeneous collections such as `(p Int, p Bool, p String)` also qualify. In this case, the `rank2Traverse` method allows one to convert e.g. `(Maybe Int, Maybe Bool, Maybe String)` to `(Either String Int, Either String Bool, Either String String)` using a function `Maybe a -> Either String a`, but requires one to be generic over both the length of the collection and the specific type of the object at each index. The heterogeneous version of our type for `hAccumTraverse` then becomes

```
newtype HAccumCont b = HAccumCont (Int -> PE effs (b, Int))
hAccumTraverse :: Rank2.Traversable t => Int -> t HAccumCont
                     -> (Int -> t Identity -> PE effs (a, Int)) -> PE effs (a, Int)
```

[4] Though there are well-known ways to simulate dependent types in Haskell; see, e.g. Eisenberg and Weirich [2012].

The body continuation array is now represented as an abstract structure whose elements are of type `HAccumCont b` ≈ `Int -> PE effs (b, Int)`, and the final resumption requires the handler to provide a same-shaped structure whose elements are of type `Identity b` (isomorphic to `b` itself), with the key difference being that now `b` can be *different* for different indices of the structure.

Remarkably, this extension is exactly what we need to bridge the gap between the **for**-based parallelism of our system λ^p and the applicative-style parallelism of Marlow et al. [2014], if we combine it with the *free applicative functor* as introduced by Capriotti and Kaposi [2014]:

```
data FreeAp p a where
  Pure :: a -> FreeAp p a
  Ap   :: FreeAp p (a -> b) -> p a -> FreeAp p b
```

`FreeAp` is a "minimal" applicative functor, and can be interpreted as a data structure that simply holds each "effectful" component `p a` in an unevaluated form along with a pure function that knows how to combine them. Furthermore, `FreeAp p a` has exactly the structure of a rank-2 traversable: it is a heterogeneous data structure where each non-`Pure` element is wrapped in some type constructor `p`. This means our system can be directly extended to support an applicative-style interface to parallelism without changing the core semantics, making it immediately compatible with Haskell's existing wide support for applicative functors, as we discuss in more detail in Section 7.

3.5 Summary

Before moving on to the details of our approach, we briefly recap the key aspects of our design. To prevent effect handling from introducing unnecessary sequential dependencies, we augment a standard effect calculus with **for** expressions, which identify parallelizable independent subcomputations, and *traverse* clauses, which enable user-defined handlers to handle them. The *traverse* receives arguments that capture (1) the **for** expression itself (with the handler pushed inside each independent subcomputation) and (2) the remaining computation outside of the **for** expression. To type-check this system, we introduce answer type constructors, and also equip the type system with value-dependent types for typing arrays. Finally, we embed our system into Haskell using a rank-2 Traversable typeclass constraint, which allows us to handle heterogeneously-typed collections and connects our approach to the existing literature on applicative functors.

4 A Calculus of Parallel Effect Handlers

In this section we introduce an untyped calculus λ^p that lays out a basis for user-extensible parallel effects, in order to demonstrate the essence of the design. §6 will present a fully typed semantics.

4.1 Syntax

The syntax and semantics of λ^p are summarized in Fig. 1. Expressions e include values v, applications $e_1\ e_2$, the parallelizable ($\textbf{for}\ x : n.\ e$) construct, the projection operation $e_1.e_2$, and parameterized handler ($\textbf{handle}\ h\ e_1\ e_2$) that takes a handler h, a handler parameter e_1, and a computation e_2 to be handled. The formalism supports parameterized handlers for generality, though they are not necessary for any of our examples; see §5 for further discussion.

Values v include literals i, variables x, lambdas $\lambda x.\ e$, arrays $\langle v_0, \ldots, v_n \rangle$, and ($\textbf{perform}\ op$) that performs an operation. We often use f for lambdas, n for literals, and s for handler parameters.

A handler h defines the semantics of effects, where for simplicity we assume that every effect has exactly one operation. A handler takes three clauses: (1) $return \mapsto f_r$, a return clause that gets applied when the computation returns a value; (2) $op \mapsto f_p$, an operation clause that defines the operation implementation; and (3) $traverse \mapsto f_t$, a novel traverse clause critical to our calculus

expressions	e	::=	$v \mid e_1\ e_2 \mid \textbf{for}\ x : n.\ e \mid e_1.e_2 \mid \textbf{handle}\ h\ e_1\ e_2$
values	v, f, n, s	::=	$i \mid x \mid \lambda x.\ e \mid \langle v_0, \ldots, v_n \rangle \mid \textbf{perform}\ op$
handlers	h	::=	$\{return \mapsto f_r, op \mapsto f_p, traverse \mapsto f_t\}$
evaluation contexts	E	::=	$\square \mid E\ e \mid v\ E \mid E.e \mid v.E \mid \textbf{handle}\ h\ E\ e \mid \textbf{handle}\ h\ s\ E$
	F	::=	$\square \mid F\ e \mid v\ F \mid F.e \mid v.F \mid \textbf{handle}\ h\ F\ e$

$$
\begin{array}{llcll}
(app) & (\lambda x.\ e)\ v & \longrightarrow & e[x := v] & \\
(index) & \langle v_0, \ldots, v_n \rangle.i & \longrightarrow & v_i & \\
(return) & \textbf{handle}\ h\ s\ v & \longrightarrow & f_r\ s\ v & \text{if } (return \mapsto f_r) \in h \\
(perform) & \textbf{handle}\ h\ s\ E[\textbf{perform}\ op\ v] & \longrightarrow & f_p\ s\ v\ k & \text{if } op \notin \text{bop}(E) \land (op \mapsto f_p) \in h \\
& & & \multicolumn{2}{l}{\quad \text{where } k = \lambda s.\ \lambda x.\ \textbf{handle}\ h\ s\ E[x]} \\
(traverse) & \textbf{handle}\ h\ s\ F[\textbf{for}\ x : n.\ e] & \longrightarrow & f_t\ n\ s\ \ell\ k & \text{if } (traverse \mapsto f_t) \in h
\end{array}
$$

$$
\text{where } \ell = \langle \ell_0, \ell_1, \ldots, \ell_{n-1} \rangle, \quad \ell_i = \lambda s.\ \textbf{handle}\ h\ s\ e[x := i], \quad k = \lambda s.\ \lambda xs.\ \textbf{handle}\ h\ s\ F[xs]
$$

$$
\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']}\ (\text{STEP}) \qquad \frac{\forall\ 0 \le i < n.\ e[x := i] \longmapsto v_i}{F[\textbf{for}\ x : n.\ e] \longmapsto F[\langle v_0, \ldots, v_{n-1} \rangle]}\ (\text{PARALLEL})
$$

Fig. 1. Syntax and semantics of λ^p.

that handles parallel effects. Here we assume *return* and *traverse* are built-in operations, and *op* is an effect-specific operation. We discuss each clause in detail in the next section.

Evaluation contexts, essentially an expression with a hole ($\square$) in it, explicitly indicate the evaluation order of an expression. As we will see, when handling an operation, we will search in the evaluation context the innermost corresponding handler. We distinguish between evaluation contexts E and *pure* evaluation context F that contains no **handle** frame. Notably, F still has the frame ($\textbf{handle}\ h\ F\ e$), where the hole is in the parameter. The notation $E[e]$ denotes an expression obtained by substituting e into the hole of E, e.g., $((v\ \square)\ f)[e] = (v\ e)\ f$. We write $\text{bop}(E)$ for the set of operations that can be handled by a handler frame in E; it follows that $\text{bop}(F) = \emptyset$ for any F.

4.2 Operational Semantics

The bottom of Fig. 1 defines the operational semantics of λ^p. The evaluation rules have two forms: $\longrightarrow$ defines a primitive evaluation step, and $\longmapsto$ evaluates expressions inside evaluation contexts. We write $\longmapsto^*$ for the reflexive and transitive closure of $\longmapsto$.

Primitive evaluation rules ($\longrightarrow$). We first discuss primitive evaluation rules. Rule (*app*) defines the standard call-by-value β-reduction. Rule (*index*) projects out the ith element from an array $\langle v_0, \ldots, v_n \rangle$, returning v_i. Rules (*return*) and (*perform*) define the standard operational semantics of effect handlers. In particular, when a handler handles a computation, there are two possibilities. If the computation returns a value, then rule (*return*) applies the return clause for that handler f_r to the value. This can be used to e.g. wrap the result v to Just v. If the computation performs an operation **perform** *op* v that calls the operation *op* with the argument v, then rule (*perform*) finds the innermost handler for the operation (specified as $op \notin \text{bop}(E)$), and applies the operation clause f_p to the parameter s, the operation argument v, as well as the resumption k. The resumption k takes a new handler parameter s and the operation result x, and captures the handler with the new parameter and the evaluation context between the handler and the operation call.

Traverse. Rule (*traverse*) captures the essence of parallel effect handlers in λ^p, adding a third option of how the computation to be handled can interact with the handlers. Specifically, if the evaluation reaches ($\textbf{for}\ x : n.\ e$) then we would like the expression e to be executed in parallel

for each x in n. However, naively evaluating e could get us stuck, as the expression may perform effectful operations! Instead, we allow the handler implementers to specify how a **for** expression should be handled. In particular, rule (*traverse*) first finds the innermost handler h, and applies its traverse clause f_t to (1) the array length n, (2) the new handler parameter s, (3) an array of body continuations ℓ, and (4) and resumption k that resumes the program segment following the loop.

There are several things to be noted here. First, h is the innermost handler for any operation rather than for a specific operation. The difference here from rule (*perform*) can be seen from the use of F (instead of E) when looking for handlers. One way to interpret the rule is that **for** is an effect that can be handled by any handler – this is true in the formalism as every handler defines the traverse clause. Second, the body continuation array ℓ reifies the computation in the original **for** expression, and pushes the handler inside. Thus, the corresponding operations in e can now be handled by h. Moreover, since the handlers in ℓ require parameters, each element of ℓ must be called with a handler parameter as its argument. Lastly, k is the resumption that takes a new handler parameter s and the result xs as the result of the for construct, and resumes the original computation. This resumption closely resembles the resumption when handling an ordinary effect.

Depending on the implementation of f_t, the program can have different behaviors.

- f_t may never call the body continuations in ℓ, in which case the **for** expression is discarded. In this case it can either pass something arbitrary to k, or abort execution entirely.
- f_t may call each element of ℓ exactly once inside a for expression, e.g. $\textbf{for}\ i : n.\ (\ell.i)\ s$. Then the **for** expression will keep propagating to outer handlers. When there is no outer handler, it means all handlers have properly handled the **for** expression, and thus we are able to execute the expressions in parallel (in rule (*parallel*), which we will discuss shortly).
- f_t may produce multiple **for** expressions, potentially calling the continuations in ℓ multiple times. Then each of these new **for** expressions will propagate to outer handlers.
- f_t may call the elements of ℓ individually, outside of a **for** expression. This will break the independence of the computations and force any remaining effects to be evaluated sequentially. (This should generally be avoided, however, if we wish to enable parallel execution.)

If a handler has no special behavior for parallelism, this default implementation may suffice:

$$\mathit{traverse} \mapsto \lambda n.\ \lambda s.\ \lambda \ell.\ \lambda k.\ k\ s\ (\textbf{for}\ i : n.\ (\ell.i)\ s)$$

In this case, the traverse clause distributes the same handler parameter s to all iterations of the original **for** expression, evaluates each iteration $\ell.i$ under a new **for** expression, and passes the handler parameter s as well as the result from the new **for** expression to k. More generally, handlers may need to pre-process the arguments to each iteration, or post-process their results; we will see more practical examples of this in §5.

Evaluation inside evaluation contexts ($\longmapsto$). We now turn to the rules that evaluate expressions inside evaluation contexts. Rule (*step*) says that if an expression e can take a primitive evaluation step to e', then the whole expression $E[e]$ evaluates to $E[e']$. Rule (*parallel*) is where parallelism takes place. Specifically, when we have a **for** expression not under any handlers (recall that F is a pure evaluation context), it means all handlers have been pushed inside the **for** expression, and so we are ready to evaluate the **for** body in parallel! For every i ranging from 0 up to n, we evaluate the expression e after substituting x by i. Here we assume some form of built-in parallelism support for evaluating the $\forall$ parallelism (for example, a set of operating system threads, or the built-in parallelism support for **for** in Dex).

Lastly, we remark that our design of treating **for** as an effectful operation also has implications on program reasoning. Specifically, for any algebraic operation op, we expect the following equality property: $F[op\ v] \equiv x \leftarrow op\ v; F[x]$ [Plotkin and Power 2003]. Generalizing the property in the

presence of handlers, we have $E[op\ v] \equiv x \leftarrow op\ v; E[x]$ when $op \notin \text{bop}(E)$. In our design, since **for** needs to be handled by every handler, we have the property for **for** expressions only under pure contexts, i.e., $F[\textbf{for}\ x : n.\ e] \equiv x \leftarrow \textbf{for}\ x : n.\ e; F[x]$. It is possible to extend our system with a "pure for" that only allows a pure body (checked by the type system), in which case handlers would not need to handle it, and a “pure for” can be naturally lifted outside of any evaluation context.

5 Practical Examples

Now that we have described our system, in this section we will show how we can apply our design and implement a variety of practically interesting effects. We will express these examples using a richer surface language that includes tuples, conditionals, algebraic data types, etc. While not included in the grammar, we can define a (**handler** $h\ s\ e$) construct that takes a computation e to be handled and calls it under the handler as syntactic sugar, which is useful for defining handlers taking a suspended (unit-taking) computation:

$$\textbf{handler}\ h\ e_1\ e_2 \triangleq \textbf{handle}\ h\ e_1\ (e_2\ ())$$

5.1 Accumulative Writer

We begin by showing again how to express the parallel accumulation effect in our language. We generalize the `accum` example from §3.2 to work on an associative binary operation (<>) and an identity element for that operation (essentially forming a *monoid*):

```
runAccum = λ(<>). λmempty.
  handler { return ↦ λs.λx. (x, s), accum ↦ λs.λx.λk. k (s <> x) (),
            traverse ↦ (λn.λs.λl.λk. pairs ← for i:n. (l.i) mempty;
                                       (results, outs) ← unzip pairs
                                       out ← reduce (<>) outs;
                                       k (s <> out) results)  } mempty
```

Here `runAccum` takes a binary operation (<>) and an identity element (`mempty`), and returns a handler, using `mempty` as the initial parameter. (We omit the implementation of the helper function `reduce` here, but note that it could be implemented using a parallel reduction circuit of depth $O(\log n)$ by forming a balanced binary tree over array elements, and using another parallel **for** construct to apply (<>) at each node in parallel.)

In the case of `sum`, we have `(<>)` being `(+)` and `mempty` being `0`. Using handlers to handle effects allows us to easily give different semantics to the same effect: if we define `(<>)` as `maximum` (without being a monoid anymore), we obtain a handler that accumulates only the largest result.

We remark again that parameterized handlers are included in the formalism for generality and for convenience when writing examples, but they are not necessary for encoding the example; we provide an unparameterized version of this handler in Appendix A.

5.2 Weak Exceptions

Our effect system can also express a form of exception handling, using the effect

```
exn { throw : String → () }
```

To account for exceptions, our handler wraps the result into the standard `Either String b` data type, with two constructs `Left String` and `Right b`. Since we wish to be able to execute **for** iterations in parallel, our handler for `exn` treats them as "weak" exceptions: an exception in one iteration of a **for** does not interrupt execution in any other iterations, although it will still prevent execution of the code after the **for** body. Our handler is as follows:

```
runWeakExcept =
  handler { return ↦ λ_.λx. Right x, throw ↦ λ_.λerr.λk. Left err,
            traverse ↦ (λn.λ_.λl.λk. eithers ← for i:n. (l.i) ();
                                     combined ← firstFailure eithers;
                                     case combined of  Left err → Left err
                                                       Right res → k () res  }
```

If a computation completes, then the `return` clause wraps it inside `Right`; otherwise the `throw` clause wraps the error inside `Left`. Inside `traverse`, we first evaluate the iteration continuation `l`, then use a function `firstFailure` (not implemented here for space) to extract either the first `Left`, or the table of values if all values were wrapped in `Right`. In case of some `Left` value, the handler will propagate it instead of calling `k`; otherwise, the handler resumes with the result `res`.

The "weak" nature of these exceptions can be observed if we combine the handler with the accumulative writer from the previous section:

```
runAccum (++) "" (λ_.  runWeakExcept (λ_.
    perform accum "start ";
    for i:5. (if i == 2 then (perform accum "!"; perform throw "error")
                        else perform accum (toString i) );
    perform accum " end")  // (Left "error", "start 01!34")
```

In this example, `runAccum` takes as the binary operator `(++)`, the string concatenation operator, and as the initial parameter the empty string `""`. All **for** iterations execute their effects in parallel, and then computation aborts at the end of the for expression (thus `" end"` will not be accumulated). The result returned is the computation result `Left "error"` and the accumulated value `"start 01!34"`. Notice that, even though the **perform** `throw "error"` happened for $i = 2$, the effects from $i = 3$ and $i = 4$ are still accumulated into the final result because their effects were processed independently.

Due to the modularity of our system, we are free to combine handlers in different orders. If `runWeakExcept` is put before `runAccum`, then the result will be only `Left "error"`.

5.3 (Pseudo) Random Number Generation

One effect that is particularly useful for real-world numerical computation is the generation of (pseudo) random numbers, which we represent with the following effect:

```
random { sampleUniform : () → Int }
```

Suppose we wish to parallelize a program such as the following example, which computes a binomial random variable by summing weighted coin flips, then scales it by another random variable:

```
binomial_times_uniform = λn. λp.
  (_, count) ← runAccum (+) 0 (λ_. for _:n. u ← perform sampleUniform ();
                                           if u < p then (perform accum 1) else ());
  v ← perform sampleUniform ();
  count * v
```

We want each coin flip to draw distinct random numbers, but also execute in parallel. One way to accomplish this is using a *splittable PRNG* [Claessen and Pałka 2013], whose state (called a *key*) can be split into arbitrarily many independent streams of random numbers; this technique is used to e.g., implement accelerator-friendly random numbers in the machine learning framework JAX [Google 2020]. Conveniently, this design can be directly mapped to our parallel effects system. We assume the existence of two functions: `splitKey`, which takes a key and a natural number, and

returns a table of new keys; and `sampleUniform`, which takes a key and returns a random number between 0 and 1. Given this, we can implement a simple random number effect as follows[5]:

```
runRandom = λseed. handler { return ↦ λkey.λx. x,
       sampleUniform ↦ (λkey.λ_.λk. ⟨key1, key2⟩ ← splitKey key 2;
                              u ← genUniform key1;
                              k key2 u),
       traverse ↦ (λn.λkey.λl.λk. keys ← splitKey key (n + 1);
                            results ← for i:n. (l.i) (keys.i) ;
                            k (keys.(n+1)) results)  } seed
```

Here the function takes an initial seed and returns a handler. We handle `sampleUniform` by splitting the key, then using one result to generate the uniform and the other as the new handler parameter to run the continuation. We handle **for** expressions similarly, except that we perform an $(n+1)$-way split to generate independent streams of random numbers for each iteration.

An interesting observation regarding this handler is that, with this implementation of `traverse`, the following two computations may yield different results:

```
resultWithFor = runRandom shared_seed (λ_. for i:2. perform sampleUniform ())
resultUnrolled = runRandom shared_seed (λ_. u0 ← perform sampleUniform ();
                                          u1 ← perform sampleUniform ();
                                          ⟨u0, u1⟩)
```

Specifically, assume the current key is some `key`. In the first case, the computation first splits `key` into two keys `key1` and `key2`, passing them to the two iterations respectively. The first iteration then splits `key1` into `key11` and `key12`, and generates one number using `key11`. In the second case, the computation first splits `key` into `key1` and `key2`, generating one number using `key1`, and then splits `key2` to `key3` and `key4`, and generating another number using `key3`. Note that the number generated using `key11` is not necessaily the same as the number generated using `key1`. On the other hand, if we knew in advance that every iteration generated exactly one random number, we could have implemented it so that they got the same numbers. But more generally, one body iteration may generate an arbitrary amount of random numbers, and it is difficult to predict how many random numbers will be generated by an arbitrary user program, which would be necessary to preserve equivalence between parallel and sequential programs in general.

As the `traverse` clause is user-defined, it should not be surprising that unrolling the **for** construct could yield different results. In this case, the particular keys used to generate (pseudo) random numbers generally should not matter, since the distributions remain the same.

5.4 Nondeterminism

Our next example is the nondeterminism effect (also known as the list monad). Conceptually, the `amb` operator takes as argument an array of values, and nondeterministically picks one. Unlike the PRNG effect, however, the result of a computation in the `Amb` is not a single result but instead the array of *all* possible results we might obtain:

```
runAmb (λ_. chars ← (for i:3. perform amb ⟨"H", "T"⟩)); reduce (++) chars)
//  ⟨"HHH", "HHT", "HTH", "HTT", "THH", "THT", "TTH", "TTT"⟩
```

We define the handler for `amb` as follows, which collects the results of all choices. Unlike the other effect handlers we have introduced, in the `amb` clause this handler calls the continuation inside a

[5]See Appendix A for an unparameterized version of `runRandom`.

parallelizable **for** expression, instead of calling it only once or discarding it. This means the amb handler can introduce new parallelism opportunities into code that appears sequential.

```
runAmb = handler { return ↦ λ_.λx. ⟨x⟩,
                   amb ↦ (λ_.λoptions.λk. n ← length options;
                                          concatenate (for i:n. k () (options.i))),
                   traverse ↦ (λn.λ_.λl.λk. results ← for i:n. (l.i) ();
                                            productElts ← cartesianProd results;
                                            m ← length productElts;
                                            for i:m. k () (productElts.i)) } ()
```

Here, return wraps the result into a singleton array, and the amb clause calls k with all possible options and collects the result into a final array. Inside the traverse clause, cartesianProd is a function which is assumed to take a length-n array of arbitrary-length arrays and return an arbitrary-length array of length-n arrays, such that each element of the result is formed by taking one element from each of the n original arrays.

Again, thanks to the compositionality of our system, users are free to nest multiple effects. For instance, by nesting runAmb inside runAccum, we can count samples with certain properties, e.g.

```
runAccum (+) 0 (λ_. runAmb (λ_.
  d1 ← perform amb ⟨0,1,2,3,4,5,6,7,8,9⟩;  d2 ← perform amb ⟨0,1,2,3,4,5,6,7,8,9⟩;
  if (d1 + d2 == 13) then perform accum 1 else () ))
```

Let us emphasize again that even though the code example looks entirely serial, it will be converted into a parallel loop over all valid values for d1 and d2 by the amb effect.

5.5 Parallelizable Shared State

Finally, we present a more complex example, which shows that our system is expressive enough to support a rudimentary form of communication between otherwise-parallel computation threads, similar to concurrent programming constructs. Our "shared state" effect is defined via an operation update, with type signature

```
sharedstate { update : (value → value) → value }
```

Running **perform** update f calls f with the current value of a shared state variable to produce a new value, then returns the original value to the user computation. This generalizes the standard get operation as **perform** update (λv. v) and the put operation as **perform** update (λ_. u); it can also be used to modify the stored value in place (such as **perform** update (λv. 2 * v)). Importantly, we require that these operations must occur in *some serial order* across parallel iterations of **for** expressions, similar to the "compare and swap" pattern for atomic variables in concurrent programming systems [Herlihy 1991]. If two subcomputations run **perform** update f and **perform** update g, respectively, then either f runs before g (and thus the output of f is the input to g), or vice versa.

We can handle the sharedstate effect by *interleaving* the updates across parallel "threads": run the body subexpressions of each **for** expression in parallel as much as possible, then run each of the functions passed to update in sequence, and then resume the continuations for each **for** subexpression in parallel. This produces a hybrid of parallel and serial execution behavior: we can continue running most of the computations in parallel, and only use sequential computations while running the (hopefully cheap) update functions.

As a concrete implementation of this in our calculus, we present a "round-robin" handler, where the parallel subcomputations take turns updating the state. The unique property of this handler compared to previous examples is that this handler responds to update events by packaging the

continuation itself into the returned value, wrapped in `Left`. These update events can then be interleaved and processed recursively by the `traverse` handler, and finally "unwound" by an outer loop. (For convenience, we express this program in terms of explicit recursive call syntax, which could be desugared using the Y combinator or a similar construction, and a recursive algebraic datatype `Rec`, which ensures the recursive program is well-typed.)

```
// `Rec a` contains either a final `a` or a pair of (update function, continuation).
data Rec a = Rec (Either (value → value, value → Rec a) a)
runSharedState = (λs. λexpr.
  // interleave : Rec a → Either (value → value, value → ⟨() → Rec a⟩) ⟨a⟩
  //                    → Either (value → value, value → ⟨() → Rec a⟩) ⟨a⟩
  // where `value → value` represents an "atomic" updater function,
  // `a` is the answer type of an individual for body subexpression,
  // `value → Rec a` is one body subexpression continuation,
  // `value → ⟨() → Rec a⟩` is an interleaved computation that runs all updates
  // in order and produces a list of new body subexpression "thunks"
  interleave ← (λ(Rec result). λrest. case (result, rest) of
    (Right b, Right bs) → Right (b:bs)
    (Left (f,k), Right bs) → Left (f, λv. (λ_.k v):(map (λb.λ_.b) bs))
    (Right b, Left (f,ks)) → Left (f, λv. (λ_.b):(ks v))
    (Left (f1,k1), Left (f2,k2)) → Left ((λv.f2 (f1 v)),(λv.(λ_.k1 v):(k2 (f1 v)))));
  // The traverse rule produces a `Rec b` from an array of `Rec a` and a continuation
  // `k : ⟨a⟩ → b` by either calling the final continuation directly (if all body
  // subexpressions produce `Right`), or packaging this continuation and a recursive
  // traversal into `Left`.
  traverse ← (λn.λ_.λl.λk.
              results ← for i:n (l.i) ();
              case foldr interleave (Right ⟨⟩) results of
                Right all → Rec (Right (k all))
                Left (f,go) → Rec (Left (f, (λv. traverse n () (go v) k))) );
  // root : Rec a
  root ← handler { return ↦ λ_.λx. Right x, update ↦ λ_.λf.λk. Left (f, k),
                   traverse ↦ traverse } () expr;
  // Unwrap each layer of the result (and update the shared state) until we obtain
  // a `Right` value, indicating termination.
  unwind ← (λs1.λv. case v of Rec (Right res) → (s1, res);
                               Rec (Left (f, k)) → unwind (f s1) (k s1));
  unwind s root)
```

Under this handler, a single **for** inside a stateful computation may transform into a sequence of **for** expressions after pushing the shared state handler into it; each such transformation represents a round of sequential state updates within the overall parallel program. This will not affect handlers that were embedded within the `runSharedState` call, but it may change the behavior of handlers outside it; for instance, if `runSharedState` is nested inside `runAccum`, all `accum` calls before the first call to `update` in each thread will be accumulated before any `accum` calls performed afterward, effectively interleaving them due to the synchronization effects.

We conclude this example by showing how to use this state effect to perform rudimentary communication between otherwise-independent subcomputations. For instance, we can send a value from the second subcomputation in a parallel **for** to the first:

```
waitUntil = λf. ( v ← perform update (λx. x); if f v then v else waitUntil f)
result = runSharedState (0,0) (λ_. for i:2.
  if i == 0 then ((x, y) ← waitUntil (λ(x,y). x == 3); y)
             else (perform update (λ_. (4, 7)); perform update (λ_. (3, 12));
                   perform update (λ_. (1, 3)); 42)  )
```

This will produce the value `((1,3), ⟨12, 42⟩)`; here `(1,3)` is the final state, and `⟨12, 42⟩` is the array of values returned by the loop. In particular, the first element of the array is 12, because this is the value of `y` when `x == 3`, even though this update was performed by the second subcomputation.

This handler demonstrates that, although our system is designed to *enable* independent parallel execution of the body of **for** expressions, it does not strictly *require* it. In Section 8 we discuss some ways to constrain the power of handlers if a stricter interpretation of parallelism is desired.

6 Typed Parallel Effect Handlers

So far we have seen the untyped operational semantics of λ^p. In this section, we present an explicitly typed calculus F^p that extends λ^p with types and typed constructs. To limit clutter in the typing rules, we present the typed calculus with *unparameterized handlers* in this section. As mentioned, parameterized handlers are not fundamental to our design; we give the typed calculus with parameterized handlers in the appendix.

To design a typed semantics, it is important to support the behavior of handlers when handling a **for** construct, namely the rule (*traverse*) (Fig. 1). We present its variant (*traverse-unp*) with unparameterized handlers below. With unparameterized handlers, the body continuation ℓ takes unit as an argument (and thus be a unit-taking thunk), so that the handler still has control over when the body continuations (and any effects contained within them) are executed.

$$(\textit{traverse-unp}) \quad \textbf{handle}\ h\ F[\textbf{for}\ x : n.\ e] \quad \longrightarrow \quad f_t\ n\ \ell\ k \quad \text{if } (\textit{traverse} \mapsto f_t) \in h$$
$$\text{where } \ell = \langle \ell_0, \ell_1, \ldots, \ell_{n-1} \rangle, \quad \ell_i = \lambda().\ \textbf{handle}\ h\ s\ e[x := i]\ \forall i, \quad k = \lambda xs.\ \textbf{handle}\ h\ F[xs]$$

According to the rule, we push the handler to be inside the **for** expression. However, the expression e that the handler applies to may have a different type from that of the whole program $F[\textbf{for}\ x : n.\ e]$. Furthermore, we see that f_t takes the array length n as an input, requiring a form of dependent types. Therefore, as discussed in §3.3, F^p has the following features:

- Each handler is associated with an answer type constructor (§3.2). A handler can be applied to computations with any types; formally, if the answer type constructor is ρ, the return clause of the handler can take any type τ, and turn it into type $(\rho\ \tau)$, the result of applying ρ to τ.
- The type system includes value dependent types to support dependency on array length.

In this section, we present F^p in detail, as a higher-order polymorphic lambda calculus equipped with a *row effect system* [Hillerström and Lindley 2016], with proven semantics preservation over λ^p and syntactic type soundness.

6.1 Syntax

Fig. 2 presents the syntax of F^p. Inside the expressions e, the changes compared to λ^p are highlighted in gray. Specifically, lambdas, **for** expressions, and **perform** are now annotated with type and effect information. Expressions are extended with type applications $(e\ \tau)$, and a form of coercion $e \triangleright \tau$ for converting between index types and integers, as we will see. Similarly, values are extended with type abstractions $(\Lambda a : \kappa.\ v)$ and coercion $(v \triangleright \tau)$. Type abstractions have values as bodies, which is needed for connecting to the untyped semantics (see §6.3).

Types τ include type variables a, type constants c (including $\mathsf{Int} : \star$), and dependent function types $(x : \tau_1) \rightarrow \epsilon\ \tau_2$; we write $(\tau_1 \rightarrow \epsilon\ \tau_2)$ where $x \notin \mathsf{fv}(\tau_2)$. Types further include polymorphic types $(\forall a : \kappa.\ \tau)$, type applications $(\tau_1\ \tau_2)$, index types $(\mathsf{Fin}\ n)$, and arrays $(\mathsf{Fin}\ n \Rightarrow \tau)$ with length

expressions	e	$::=$	$v \mid e_1\ e_2 \mid e\ \tau \mid \textbf{handle}\ h\ e \mid \textbf{for}\ x : \mathsf{Fin}\ n.\ e \mid e_1.e_2 \mid e \triangleright \tau$
values	v, f, n, s	$::=$	$i \mid x \mid \lambda^{\epsilon} x : \tau.\ e \mid \Lambda a : \kappa.\ v \mid \langle v_0, \ldots, v_n \rangle \mid \textbf{perform}\ op\ \epsilon\ \overline{\tau} \mid v \triangleright \tau$
handlers	h	$::=$	$\{return \mapsto f_r, op \mapsto f_p, traverse \mapsto f_t\}$
types	τ, σ, ρ	$::=$	$a \mid c \mid (x : \tau_1) \to \epsilon\ \tau_2 \mid \forall a : \kappa.\ \tau \mid \tau_1\ \tau_2 \mid \mathsf{Fin}\ n \mid \mathsf{Fin}\ n \Rightarrow \tau$
effect rows	ϵ	$::=$	$a \mid \langle\rangle \mid \langle l \mid \epsilon \rangle$
kinds	κ	$::=$	$\star \mid \kappa_1 \to \kappa_2 \mid \mathsf{eff}$
type ctxs	Γ	$::=$	$\bullet \mid \Gamma, x : \tau \mid \Gamma, a : \kappa$
effect ctxs	Σ	$::=$	$\{\ \overline{l_i : \{\ op_i : \forall \overline{a : \kappa}.\ \sigma_1 \to \sigma_2\ \}}\ \}$

$\boxed{\Gamma \vdash_{\mathsf{wf}} \tau : \kappa}$ *(Kinding)*

$$\textsc{K-VAR}\ \frac{a : \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{wf}} a : \kappa} \qquad \textsc{K-CONST}\ \frac{(c : \kappa)}{\Gamma \vdash_{\mathsf{wf}} c : \kappa} \qquad \textsc{K-APP}\ \frac{\Gamma \vdash_{\mathsf{wf}} \tau_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\mathsf{wf}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{wf}} \tau_1\ \tau_2 : \kappa_2}$$

$$\textsc{K-ARROW}\ \frac{\Gamma \vdash_{\mathsf{wf}} \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash_{\mathsf{wf}} \tau_2 : \star \qquad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash_{\mathsf{wf}} (x : \tau_1) \to \epsilon\ \tau_2 : \star} \qquad \textsc{K-FORALL}\ \frac{\Gamma, a : \kappa \vdash_{\mathsf{wf}} \tau : \star}{\Gamma \vdash_{\mathsf{wf}} \forall a : \kappa.\ \tau : \star}$$

$$\textsc{K-FIN}\ \frac{\Gamma \vdash_{\mathsf{v}} n : \mathsf{Int}}{\Gamma \vdash_{\mathsf{wf}} \mathsf{Fin}\ n : \star} \qquad \textsc{K-ARRAY}\ \frac{\Gamma \vdash_{\mathsf{wf}} \mathsf{Fin}\ n : \star \qquad \Gamma \vdash_{\mathsf{wf}} \tau : \star}{\Gamma \vdash_{\mathsf{wf}} \mathsf{Fin}\ n \Rightarrow \tau : \star} \qquad \textsc{K-EMPTY}\ \frac{}{\Gamma \vdash_{\mathsf{wf}} \langle\rangle : \mathsf{eff}} \qquad \textsc{K-ROW}\ \frac{\Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash_{\mathsf{wf}} \langle l \mid \epsilon \rangle : \mathsf{eff}}$$

Fig. 2. Syntax and well-formed types of F^p

n. Note that index types $\mathsf{Fin}\ n$ can depend on a value n. Combining index types and value dependent types, we can have a valid type such as $(n : \mathsf{Int}) \to \epsilon\ (\mathsf{Fin}\ n \Rightarrow \mathsf{Int})$.

Effect rows ϵ are either a variable a, the empty row $\langle\rangle$, or effect concatenation $\langle l \mid \epsilon \rangle$. Note how lambdas $(\lambda^{\epsilon} x : \tau.\ e)$ and function types $(x : \tau_1) \to \epsilon\ \tau_2$ are annotated with the effect information.

We employ a kind system to distinguish different types and to ensure well-formedness of types. Kinds κ include the basic kind $\star$, the arrow kind $\kappa_1 \to \kappa_2$, and the effect kind eff. The bottom of Fig. 2 defines well-formedness of types. The judgment $\Gamma \vdash_{\mathsf{wf}} \tau : \kappa$ reads that under the typing context Γ, the type τ has kind κ. Most rules are standard. In rule K-FIN, the rule uses the value typing $(\vdash_{\mathsf{v}})$, which we will discuss shortly, to check that n has type Int.

Lastly, the type context Γ maps variables to their types, and type variables to their kinds. And the effect context Σ maps an effect label l to its operation op which, with polymorphic variables $\overline{a}$, takes an operation argument σ_1 and returns a result σ_2.

6.2 Type System

Fig. 3 presents the typing rules for F^p. There are three judgments that type-check values $(\Gamma \vdash_{\mathsf{v}} v : \tau)$, expressions $(\Gamma \vdash e : \tau \mid \epsilon)$, and handlers $(\Gamma \vdash_{\mathsf{h}} h : l \mid \epsilon \mid \rho)$, respectively.

The judgment $\Gamma \vdash_{\mathsf{v}} v : \tau$ reads that under the type context Γ, the value v has type τ. Rule T-LIT type-checks integers. Since F^p is explicitly typed, an explicit coercion is needed to convert between index types and integers. Rule T-FIN types literal i with $\mathsf{Fin}\ n$ only if we know that $0 \leq i < n$. In rule T-ABS, the function type keeps track of the effect that can be raised by the body when the function is applied. Rule T-ARRAY type-checks an array of length n, returning the array type $\mathsf{Fin}\ n \Rightarrow \tau$. Lastly, rule T-PERFORM type-checks **perform**. The rule first gets from the effect context the type of op. It then type-checks that the type arguments $\overline{\tau}$ have the expected kinds $\overline{\kappa}$. The result type is a function from σ_1 to σ_2 with the type variables substituted accordingly. Moreover, **perform** carries the effect context ϵ, and in the result type the rule adds the label l to the effect.

$\boxed{\Gamma \vdash_{\mathsf{v}} v : \tau}$ *(Typing values)*

$$\text{T-VAR}\quad \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\mathsf{v}} x : \tau} \qquad \text{T-LIT}\quad \frac{}{\Gamma \vdash_{\mathsf{v}} i : \mathsf{Int}} \qquad \text{T-FIN}\quad \frac{0 \le i < n}{\Gamma \vdash_{\mathsf{v}} (i \triangleright \mathsf{Fin}\ n) : \mathsf{Fin}\ n} \qquad \text{T-ABS}\quad \frac{\Gamma \vdash_{\mathsf{wf}} \tau_1 : \star \qquad \Gamma, x : \tau_1 \vdash e : \tau_2 \mid \epsilon}{\Gamma \vdash_{\mathsf{v}} (\lambda^{\epsilon} x : \tau_1.\ e) : (x : \tau_1) \to \epsilon\ \tau_2}$$

$$\text{T-TABS}\quad \frac{\Gamma, a : \kappa \vdash_{\mathsf{v}} v : \tau}{\Gamma \vdash_{\mathsf{v}} (\Lambda a : \kappa.\ v) : \forall a : \kappa.\ \tau} \qquad \text{T-ARRAY}\quad \frac{\Gamma \vdash_{\mathsf{v}} v_i : \tau}{\Gamma \vdash_{\mathsf{v}} \langle v_0, \ldots, v_n \rangle : \mathsf{Fin}\ n \Rightarrow \tau}$$

$$\text{T-PERFORM}\quad \frac{op : \forall \overline{a : \kappa}.\ \sigma_1 \longrightarrow \sigma_2 \in \Sigma(l) \qquad \Gamma \vdash_{\mathsf{wf}} \overline{\tau} : \overline{\kappa}}{\Gamma \vdash_{\mathsf{v}} \textbf{perform}\ op\ \epsilon\ \overline{\tau} : (\sigma_1 \to \langle l \mid \epsilon \rangle\ \sigma_2)[\overline{a} := \overline{\tau}]}$$

$\boxed{\Gamma \vdash e : \tau \mid \epsilon}$ *(Typing expressions)*

$$\text{T-VAL}\quad \frac{\Gamma \vdash_{\mathsf{v}} v : \tau}{\Gamma \vdash v : \tau \mid \epsilon} \qquad \text{T-INT}\quad \frac{\Gamma \vdash e : \mathsf{Fin}\ n \mid \epsilon}{\Gamma \vdash (e \triangleright \mathsf{Int}) : \mathsf{Int} \mid \epsilon} \qquad \text{T-TAPP}\quad \frac{\Gamma \vdash e : \forall a : \kappa.\ \sigma \mid \epsilon \qquad \Gamma \vdash_{\mathsf{wf}} \tau : \kappa}{\Gamma \vdash e\ \tau : \sigma[a := \tau] \mid \epsilon}$$

$$\text{T-APP1}\quad \frac{\Gamma \vdash e_1 : \tau_1 \longrightarrow \epsilon\ \tau_2 \mid \epsilon \qquad \Gamma \vdash e_2 : \tau_1 \mid \epsilon}{\Gamma \vdash e_1\ e_2 : \tau_2 \mid \epsilon} \qquad \text{T-APP2}\quad \frac{\Gamma \vdash e : (x : \tau_1) \longrightarrow \epsilon\ \tau_2 \mid \epsilon \qquad \Gamma \vdash_{\mathsf{v}} v : \tau_1}{\Gamma \vdash e\ v : \tau_2[x := v] \mid \epsilon}$$

$$\text{T-FOR}\quad \frac{\Gamma \vdash_{\mathsf{wf}} \mathsf{Fin}\ n : \star \qquad \Gamma, x : \mathsf{Fin}\ n \vdash e : \tau \mid \epsilon}{\Gamma \vdash (\textbf{for}\ x : \mathsf{Fin}\ n.\ e) : (\mathsf{Fin}\ n \Rightarrow \tau) \mid \epsilon} \qquad \text{T-PRJ}\quad \frac{\Gamma \vdash e_1 : \mathsf{Fin}\ n \Rightarrow \tau \mid \epsilon \qquad \Gamma \vdash e_2 : \mathsf{Fin}\ n \mid \epsilon}{\Gamma \vdash e_1.e_2 : \tau \mid \epsilon}$$

$$\text{T-HANDLE}\quad \frac{\Gamma \vdash_{\mathsf{h}} h : l \mid \epsilon \mid \rho \qquad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \textbf{handle}\ h\ e : (\rho\ \sigma) \mid \epsilon}$$

$\boxed{\Gamma \vdash_{\mathsf{h}} h : l \mid \epsilon \mid \rho}$ *(Typing handlers)*

$$\text{T-HANDLER}\quad \frac{\begin{array}{c} op : \forall \overline{a : \kappa}.\ \sigma_1 \longrightarrow \sigma_2 \in \Sigma(l) \qquad \Gamma \vdash_{\mathsf{v}} f_r : \forall a : \star.\ a \to \epsilon\ (\rho\ a) \qquad \Gamma \vdash_{\mathsf{wf}} \rho : \star \to \star \\ \Gamma \vdash_{\mathsf{v}} f_p : \forall (\overline{a : \kappa}).\ \forall (b : \star).\ \sigma_1 \to \epsilon\ (\sigma_2 \to \epsilon\ (\rho\ b)) \to \epsilon\ (\rho\ b) \\ \Gamma \vdash_{\mathsf{v}} f_t : \forall (a : \star)(b : \star).\ (n : \mathsf{Int}) \to \epsilon\ (\mathsf{Fin}\ n \Rightarrow (() \to \epsilon\ (\rho\ a))) \\ \qquad\qquad \to \epsilon\ ((\mathsf{Fin}\ n \Rightarrow a) \to \epsilon\ (\rho\ b)) \to \epsilon\ (\rho\ b) \end{array}}{\Gamma \vdash_{\mathsf{h}} \{ return \mapsto f_r, op \mapsto f_p, traverse \mapsto f_t \} : l \mid \epsilon \mid \rho}$$

Fig. 3. Typing rules of F^p

We now move to typing expressions. The judgment $\Gamma \vdash e : \tau \mid \epsilon$ reads that under the type context Γ, the expression e has type τ and may produce effects in ϵ. Rule T-VAL uses $\vdash_{\mathsf{v}}$ to type-check values, which are allowed to have any effect annotations. Rule T-INT converts from an index into an integer.

There are two rules concerning applications. Rule T-APP1 types $e_1\ e_2$, where e_2 is an arbitrary expression, in which case the type of e_1 must be $\tau_1 \to \epsilon\ \tau_2$; namely, the argument cannot appear in types. On the other hand, rule T-APP2 types $e\ v$, where the argument is a value. In such case, e_1 can have type $(x : \tau_1) \to \epsilon\ \tau_2$. The rule then checks that v has type τ_1, and the result type is $\tau_2[x := v]$.

Rule T-FOR type-checks **for** expressions. Note that $(x : \mathsf{Fin}\ n)$ is added to the context where checking e. The result type is an array $\mathsf{Fin}\ n \Rightarrow \tau$. Rule T-PRJ projects from an array. The rule checks that the index e_2 has type $\mathsf{Fin}\ n$, ensuring that there is no out-of-bounds exception.

Rule T-HANDLE takes care of handling. The rule uses $\vdash_{\mathsf{h}}$ to get information about the handler h; more details will be explained together with rule T-HANDLER below. The computation e to be handled has type σ, and may produce effects in $\langle l \mid \epsilon \rangle$. The result type is then $\rho\ \sigma$, with effect ϵ.

Lastly, the judgment $\Gamma \vdash_{\mathsf{h}} h : l \mid \epsilon \mid \rho$ reads that under the type context Γ, the handler handles label l, with effect context ϵ, and, most interestingly, has answer type constructor ρ. The type constructor ρ is unique to our calculus, and is the key to type-check the *traverse* clause inside the handler. In this rule, we first get the type of the operation from the context $\Sigma(l)$. Then we check three clauses with value typing. (1) The return clause f_r takes any type a and turns it into type $\rho\ a$. (2) The operation clause f_p takes the operation argument σ_1, and a resumption that waits for the operation result σ_2, and returns the result $\rho\ b$. Here again the operation is polymorphic over the result b, but we know from the result clause that the result must have type constructor ρ. (3) Finally, the traverse clause f_t takes the length of the array n (using the dependent function type), the body continuation array $\mathsf{Fin}\ n \Rightarrow (() \to \epsilon\ (\rho\ a))$, and the resumption that waits for the result of the **for** expression $\mathsf{Fin}\ n \Rightarrow a$, and produces the result in $\rho\ b$.

6.3 Operational Semantics and Semantics Preservation

Since the expressions are now explicitly typed, we can update the operational semantics (Fig. 1) to be explicitly typed accordingly. The updates are mostly standard and, for space reasons, we put the typed operational semantics in the appendix. We use $\longrightarrow\!\!\!\!\!\rightarrow$ and $\longmapsto\!\!\!\!\!\rightarrow$ (instead of $\longrightarrow$ and $\longmapsto$) for the corresponding evaluation rules in F^p.

We can prove that the typed operational semantics preserves the semantics of λ^p. To this end, we first define an erasure function $[\![\cdot]\!]$ that erases all type and effect information in the input. Below we present a few interesting cases; the complete definition is provided in the appendix:

$$[\![\mathbf{for}\ x : \mathsf{Fin}\ n.\ e]\!] = \mathbf{for}\ x : n.\ [\![e]\!] \qquad [\![e \triangleright \tau]\!] = [\![e]\!] \qquad [\![\Lambda a : \kappa.\ v]\!] = [\![v]\!] \qquad [\![e\ \tau]\!] = [\![e]\!]$$

First, notice that erasing a **for** expression with index type $\mathsf{Fin}\ n$ only erases the Fin, but not the number n — as we have seen in rule (*parallel*), the index n plays an role in the operational semantics. Type information, including in coercion, type abstractions, and type applications, is all erased.

Now it also becomes more evident why type abstractions $\Lambda a : \kappa.\ v$ have a value body in F^p, which is needed to ensure semantics preservation over the untyped semantics [Xie et al. 2020]. Specifically, if we allow type abstractions over arbitrary expressions, then consider the expression $(\lambda^{\epsilon} f : \sigma.\ e)\ (\Lambda a : \kappa.\ \mathbf{perform}\ op\ \epsilon\ v)$, where we assume an operation op, and omit some annotations for clarity. In F^p, the argument $(\Lambda a : \kappa.\ \mathbf{perform}\ op\ \epsilon\ v)$ is a value, and thus the operation is not performed until f is applied to a type argument inside the expression e. On the other hand, with erasure, we have $[\![\Lambda a : \kappa.\ \mathbf{perform}\ op\ \epsilon\ v]\!] = \mathbf{perform}\ op\ [\![v]\!]$, which is *not* a value, and the operation will get performed immediately. By restricting type abstractions to have a value body, we rule out this example. Such a restriction is reminiscent of the *value restriction* [Wright 1995]. In practice, it is often the case that we have a function under a type abstraction, so the restriction is of less practical relevance. More formally, we can prove:

Lemma 6.1 (Type erasure of values)**.** *If e is a value in F^p, then $[\![e]\!]$ is a value in λ^p.*

We are then ready to prove that the typed semantics preserves of F^p the untyped semantics of λ^p:

Theorem 6.2 (Semantics preservation)**.** *If $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$, then either $[\![e_1]\!] \longmapsto [\![e_2]\!]$, or $[\![e_1]\!] = [\![e_2]\!]$.*

6.4 Type Soundness

We prove that F^p enjoys syntactic type soundness. Type preservation is straightforward.

Theorem 6.3 (Type Preservation)**.** *Given $\bullet \vdash e : \tau \mid \epsilon$, and $e \longrightarrow e'$, then $\bullet \vdash e' : \tau \mid \epsilon$.*

Next, we establish progress. The following lemma establishes progress with effects. When an expression has effects, we must consider the case where the expression gets stuck because of unhandled operations inside: (1) a pure context with an effectful **for** expression and thus (the typed version of) rule (*parallel*) does not apply; or (2) a context without a corresponding handler.

Lemma 6.4 (Progress with effects). *If $\bullet \vdash e : \tau \mid \epsilon$, then either e is a value, or there exists e' such that $e \longmapsto e'$, or $e = F[\textbf{for}\ i : \mathsf{Fin}\ n.\ e]$, or $e = E[\textbf{perform}\ op\ \epsilon'\ \overline{\tau}\ v]$ such that $op \notin \mathsf{bop}(E)$.*

When expressions are pure, both cases (1) and (2) are impossible, and thus we have:

Theorem 6.5 (Progress). *If $\bullet \vdash e : \tau \mid \langle\rangle$, then either e is a value, or there exists e' such that $e \longmapsto e'$.*

7 Haskell Implementation

We now show the practicality of our design by providing an implementation it as a Haskell library. Although we build our implementation on top of Haskell's own interpreter and thus inherit its execution semantics and type system, our library exhibits the same overall behavior as λ^p, and many aspects of F^p have Haskell equivalents. We focus here on the key features of our library.

7.1 Representing Effects as Algebraic Data Types

We represent effects using generalized algebraic data types (GADTs) parameterized with their return value, following the approach of Kiselyov and Ishii [2015]. Example effect definitions include:

```
data Reader v r   where  Ask :: Reader v v
data Except e r   where  Throw :: forall r. e -> Except e r
data Amb r        where  Choose :: forall r. [r] -> Amb r
```

Here `Ask :: Reader v v` means that the `Ask` operation is part of the `Reader v` effect, and returns a result of type `v` to the continuation. `Choose :: [r] -> Amb r` means that the `Choose` operation is in the `Amb` effect, takes a list `[r]` of arbitrary type, and returns a single element of that type. (This means that we identify the effect kind eff with the kind $\star \to \star$ of partially-applied GADTs.)

Effectful computations occur within a monad `PE effs a`, where `effs` is the effect row and `a` is the result type. The `PE` monad is based on the "free-er" extensible effect monad `Eff` of Kiselyov and Ishii [2015], which we adapt to distinguish between explicit effects (represented as `Impure (Effect eff) cont`) and parallel traversals (represented as `Impure (Traverse body) cont`). Individual effects can be performed in an effectful context using the function

```
perform :: Member eff effs => eff r -> PE effs r
```

Here `Member eff effs` is a typeclass constraint that ensures the Haskell compiler can identify the position of `eff` within the effect row for `effs` and thus relay it to the right handler.

7.2 Defining and Using Effect Handlers

Users implement new handlers by defining instances of a parallel handler typeclass, which includes an implementation for each handler clause as well as declarations of the types of the handled effect, existing effects, handler parameter, and answer type constructor:

```
class ParallelizableHandler h where
  type Effs h :: [Effect]
  type Op h :: Effect
  type Param h :: Type
  type Answer h :: Type -> Type
  handleReturn   :: h -> Param h -> a -> PE (Effs h) (Answer h a)
  handlePerform  :: h -> Param h -> Op h a ->
                      (Param h -> a -> PE (Effs h) (Answer h b)) ->
                      PE (Effs h) (Answer h b)
  handleTraverseRank2 :: Rank2.Traversable struct =>
                           h -> Param h -> struct (HandledCont h) ->
```

```
                                (Param h -> struct Identity -> PE (Effs h) (Answer h a)) ->
                                PE (Effs h) (Answer h a)
```

Here `Effs h` is the row of effects the handler itself can use, `Op h` is the effect the handler can handle, `Param h` is the type of the (optional) handler parameter (which can be `()` for handlers without parameters), and finally `Answer h` is the answer type constructor (§3.2), which gives the handler partial control over the handled results.

We use the `Rank2.Traversable` typeclass from the `rank2classes` package [Blažević and Légaré 2017] to ensure that handlers preserves the length and types of their input array. The `Rank2.Traversable struct` constraint implies that a value of type `struct p` is (isomorphic to) a heterogenous collection of values `(p a1, p a2, ..., p an)` for some types `a1, a2, ..., an`. The handler implementation can interact with such a structure using the following heterogenously-typed generalizations of the ordinary `Functor`, `Foldable`, and `Traversable` typeclass methods:

```
class Rank2.Functor s where
    -- If `f` maps `p a` to `q a` for all `a`, `(f Rank2.<$>)` maps a heterogenous
    -- collection of `p a`, `p b`, ... to another collection of `q a`, `q b`, ...
    Rank2.(<$>) :: (forall a. p a -> q a) -> s p -> s q

class Rank2.Foldable s where
    -- If `f` maps `p a` to a monoid `m` for all `a`, `(Rank2.foldMap f)` aggregates
    -- info across a heterogenous collection of `p a`, `p b`, ... into a single `m`.
    Rank2.foldMap :: Monoid m => (forall a. p a -> m) -> s p -> m

class (Rank2.Functor s, Rank2.Foldable s) => Rank2.Traversable s where
    -- If `f` maps `p a` to `q a` with effects in an applicative `m`,
    -- `(Rank2.traverse f)` maps a heterogenous collection of `p a`, `p b`, ... to a
    -- heterogenous collection of `q a`, `q b`, ... with effects in `m`.
    Rank2.traverse :: Applicative m => (forall b. p a -> m (q a)) -> s p -> m (s q)
```

The type `HandledCont h a` is a newtype wrapper around `Param h -> PE (Effs h) (Answer h a)`, used to embed handled iteration continuations in the rank-2 structure. A value of type `struct (HandledCont h)` is conceptually a collection of values of type `Param h -> PE (Effs h) (Answer h a)` where `a` may differ between elements of the collection.

As a convenience feature, we allow handler implementers to avoid implementing their handler directly in terms of the Rank-2 typeclass methods, and instead implement the simpler signature

```
handleTraverseList :: h -> Param h -> [Param h -> PE effs (Answer h a)] ->
                      (Param h -> [a] -> PE (Effs h) (Answer h b)) ->
                      PE (Effs h) (Answer h b)
```

in which case the length and types are checked dynamically using runtime assertions. A handler defined via either method can then be used to handle effects using the function

```
handle :: ParallelizableHandler h =>
    h -> Arg h -> PE (Op h : Effs h) r -> PE (Effs h) (Answer h r)
```

Similar to the corresponding construct in λ^p, `handle` takes a handler, an argument, and an effectful computation, and evaluates to a computation with that effect handled (represented in the type system by removing that effect from the effect row).

7.3 Expressing Parallel Computations

We adapt the parallel **for** `i:n e` expression into a Haskell function `forP [0..n-1] (\i -> e)`, with type `forP :: Traversable t => t a -> (a -> PE effs b) -> PE effs (t b)`. This performs a parallel effectful map over the structure `t a`, in contrast to the monadic version `forM` with the same signature provided by the Haskell standard library, and has the same behavior as **for** in λ^p.

As an extension, and motivated by the design considerations in §3.4 we additionally allow users to directly map over *heterogeneous* collections using the method

```
rank2TraverseP :: Rank2.Traversable t =>
    (forall a. p a -> PE effs (q a)) -> t p -> PE effs (t q)
```

which can be applied to any collection type as long as it is an instance of the `Rank2.Traversable` typeclass. A particularly interesting and relevant heterogeneous collection is the free applicative functor of Capriotti and Kaposi [2014], which at its simplest takes the form

```
data FreeAp p a where
  Pure :: a -> FreeAp p a
  Ap   :: FreeAp p (b -> a) -> p b -> FreeAp p a
```

Defining the type alias `Indep effs a = FreeAp (PE effs) a`, each value of type `Indep effs a` consists of a heterogeneously-typed list of independent effectful subcomputations (each of type `PE effs b` for some `b` and wrapped in the constructor `Ap`), coupled with a single pure function which combines their results (wrapped in `Pure`).[6] We provide a lifting function `indep :: PE effs a -> Indep effs a` that embeds a single effectful computation as a length-one list, and allow users to combine multiple such expressions using Haskell's `<*>` and add postprocessing logic using `<$>`, making it possible to build full parallel programs. For instance, a program that collects two results into a tuple could be written as `(,) <$> indep (perform ActionA) <*> indep (perform ActionB)`. Importantly, this does not actually run or even combine the computations, and instead simply holds them in a list inside `FreeAp`. The computations must be explicitly invoked using the function `runIndep :: Indep effs a -> PE effs a`, which evaluates all stored effectful subexpressions in parallel under a single call to `rank2TraverseP`.

Remarkably, our implementation of `runIndep` can be implemented using `rank2TraverseP` alone without any knowledge of the internals of our system. This means we get applicative-style parallelism "for free": we can directly combine our handlers with Haskell's existing syntax and library support for applicative functors without making any changes to the core semantics. For instance, using the `QualifiedDo` and `ApplicativeDo` extensions [Marlow et al. 2016], we can have GHC automatically parallelize code written in do-notation, so that the following all have the same behavior:

```
foo = do
  [va, vb] <- forP [0,1] $ \i ->
    if i==0 then perform ActionA
            else perform ActionB
  vc <- perform (ActionC va vb)
  pure (va, vb, vc)
```

```
foo = do
  (va, vb) <- runIndep $ do
    va <- indep $ perform ActionA
    vb <- indep $ perform ActionB
    pure (va, vb)
  vc <- perform (ActionC va vb)
  pure (va, vb, vc)
```

```
foo = AutoParallel.do
  va <- perform ActionA
  vb <- perform ActionB
  vc <- perform (ActionC va vb)
  pure (va, vb, vc)
```

We note that the operator `(<*>)` for `Indep effs a` is not the same as the operator `(<*>)` for `PE effs a`; the former runs operations in parallel (and is handled by our *traverse* clause) whereas Haskell's monad/applicative laws require the latter to run them sequentially using the monadic bind

[6] Since our system is agnostic to the actual data representation of `FreeAp` as long as it can be traversed, in practice we can use an equivalent but asymptotically more efficient representation due to Menendez [2013].

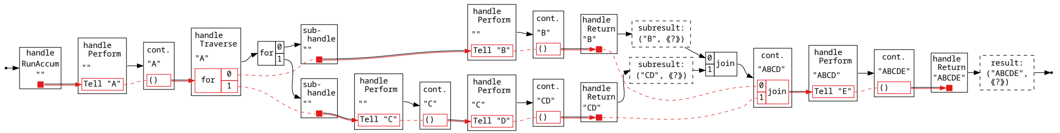

Fig. 4. Graph of dependencies generated by our tracing subsystem for a program using our `runAccum` handler: `do perform (Tell "A"); forP [0,1] (\i -> if i==0 then perform (Tell "B") else (do perform (Tell "C"); perform (Tell "D"))); perform (Tell "E")`. Observe that `"B"` is accumulated separately from `"C"` and `"D"` (visible as the parameter of each `handleReturn`), and then combined afterward.

`(>>=)`. Indeed, `Indep effs a` is not a monad at all, and thus the effectful subexpressions of `Indep effs a` must be independent by construction, which is the key feature enabling parallelization.

7.4 Pure and Concurrent Backends

We provide two execution strategies for effectful computations in our library. The first, `runPure :: PE '[] r -> r`, runs top-level paralellizable expressions using an ordinary `map`. When the Haskell program is compiled using the `-threaded` flag, this may result in parallel execution, but this is not guaranteed. The second, `runConcurrentIO :: PE '[IO] r -> IO r`, runs them by explicitly forking parallel threads using Haskell's concurrency primitives [Peyton Jones et al. 1996], yielding a computation in the IO monad. In general, these execution strategies should always produce the same result for pure programs.

Additionally, however, our `runConcurrentIO` handler explicitly embeds the `IO` monad into the effect row, making it possible for the user code to perform arbitrary IO actions, and, more importantly, for handler code to perform IO actions as part of handling the effects. This makes it possible to construct alternative implementations for some handlers that can take advantage of Haskell's concurrent IO primitives. For instance, we can provide a handler `runSharedIO` for the `SharedState` effect which uses the native `IO`-based `MVar` synchronization mechanism to mediate access to the shared state in an opportunistic (and nondeterministic) way, instead of using the deterministic round-robin strategy discussed in §5.5.

7.5 Visualizing Dependencies Using Runtime Tracing

To help illustrate the behavior of our design, we augment the Haskell implementation with a runtime tracing subsystem, which transforms an effectful computation by intercepting all effects, `forP` calls, handlers, and continuations, and adding additional metadata to allow reconstructing a graph of their sequential dependencies. This allows us to automatically construct visualizations of each of the example effects described in §5. Figure 4 shows one example for a program using our `Accum` effect.

8 Discussion

In this section, we discuss potential extensions and design variants of our design.

8.1 Pairwise Applicative-Style Parallelization

Our calculus λ^p focuses on n-ary parallelism expressed using **for** $i : n$ expressions. As we have seen in §3.4 and §7.3, it is possible to extend our system to handle heterogeneous collections, but this is currently not directly reflected in our typed semantics F^p because all elements of arrays are assumed to have the same type. An alternative design for our system would be to directly express parallelism in the style of the applicative `<*>`. Instead of **for** expressions and *traverse* clauses, we could add $e_1 \circledast e_2$ expressions and *app* clauses, with the semantics and handler type given by

$$(app) \quad \textbf{handle}\ h\ s\ F[e_1 \circledast e_2] \quad \longrightarrow \quad f_a\ s\ k_1\ k_2\ k_3 \quad \text{if } (app \mapsto f_a) \in h$$
$$\text{where } k_1 = \lambda s.\ \textbf{handle}\ h\ s\ e_1, \quad k_2 = \lambda s.\ \textbf{handle}\ h\ s\ e_2, \quad k_3 = \lambda s.\ \lambda x.\ \textbf{handle}\ h\ s\ F[x]$$

$$\frac{e_1 \longmapsto v_1,\ e_2 \longmapsto v_2}{F[e_1 \circledast e_2] \longmapsto F[v_1\ v_2]}\ (\textit{PARALLEL}')$$

$$\begin{aligned} f_a : &\forall (a : \star)(b : \star)(c : \star). \\ &\tau \to_\epsilon (\tau \to_\epsilon (\rho\ (a \to_\epsilon b))) \to_\epsilon (\tau \to_\epsilon (\rho\ a)) \\ &\to_\epsilon (\tau \to_\epsilon b \to_\epsilon (\rho\ c)) \to_\epsilon (\rho\ c) \end{aligned}$$

Such a clause would resemble a hybrid of <*> in an applicative functor and the ordinary *op* clause for an effect handler. The advantage of this formulation is that it allows one to directly express parallelism across computations of different types in the semantics. The primary disadvantage is that handlers would be restricted to combining pairs of effectful expressions in isolation, in the order they appear in the program, and could not make use of potentially-more-efficient n-way splits or reductions. More complex variants may also be possible, e.g. incorporating some version of free applicative functors into λ^p to support heterogeneous n-ary parallelism.

8.2 Immediately-Invoked Body Continuations

Our *traverse* clause currently allows handlers to call the elements of the body continuation array more than once, or to call them outside of a **for** expression. However, all of our example handlers call each body continuation exactly once inside a **for** expression. Thus, an alternative design would be to always translate the **for** expression this manner, and then pass the result to the traverse clause. For unparameterized handlers, such a design leads to the following rule for traverse:

$$(traverse) \quad \textbf{handle}\ h\ F[\textbf{for}\ x : n.\ e] \quad \longrightarrow \quad f_t\ n\ \ell\ k \quad \text{if } (traverse \mapsto f_t) \in h$$
$$\text{where } \ell = \textbf{for}\ x : n.\ \textbf{handle}\ h\ e,\ k = \lambda xs.\ \textbf{handle}\ h\ F[xs]$$

A similar rule could be constructed for parameterized handlers, by having the user specify how to split the handler parameter s across the iterations. One interpretation of such a rule is that all handlers will be pushed inside the **for** expression, since ℓ gets immediately handled by the next handler. The handler h then defines how to process the result from ℓ and what gets passed to the resumption k; for example, `runAccum` (§5.1) reduces the accumulated results, while `runWeakExcept` (§5.2) may or may not call k depending on the result from ℓ. This new rule could be useful when the system wants to ensure that each parallel computation always runs once.

8.3 Restricting the Answer Type Constructor

Under our current system, handlers are allowed to directly include a continuation as part of their result, producing an answer type containing a function type. This can allow handlers to rewrite the structure of programs in powerful ways. For instance, our shared state effect (§5.5) uses this to introduce synchronization points that enable communication between otherwise-independent "threads". This can be useful, but it can also lead to potentially surprising changes in the behavior of user programs (especially in the presence of other effects), and may impede program analysis.

It may be desirable to limit this ability in order to provide stronger guarantees about the behavior of parallel computations. One way would be to forbid the answer type constructor from including function types, effectively making functions second-class in the effect system; this would ensure that the continuations cannot escape from handlers. This kind of limitation has been explored in Dex, where effects are forbidden from carrying function types to enable ahead-of-time compilation.

9 Related Work

Algebraic effect handlers. Algebraic effects and handlers have been studied extensively; for most recent development, see e.g. Ghica et al. [2022] for a C++ effect handlers library, Phipps-Costin et al. [2023] for an effect handlers based design for WebAssembly, and Tang et al. [2024] for combining

effect handlers and linear types. Effect handlers have been implemented for OCaml to support concurrency [Dolan et al. 2018; Sivaramakrishnan et al. 2021].

More related to our work, Lindley [2014] and Pieters et al. [2020] identified applicative computations, a limited subset of monadic computations that can be handled by *applicative handlers*. In applicative computations, there cannot be dynamic data flow or control flow between computations. As such, applicative computations can naturally be parallelized. On the other hand, using an applicative handler requires restricting the expressivity of user code. In contrast, our design allows both dynamic data- and control-flow inside **for**. We use this expressivity throughout the examples (§5). As an example, the program `binomial_times_uniform` (§5.3) uses the result of (**perform** `sampleUniform ()`) to determine whether to run (**perform** `accum 1`). This work is the first to study the combination of effect handlers and parallel (monadic) computations.

Higher-order effect handlers. We essentially interpreted **for** as an effect to be handled by a handler. Such a design corresponds to a form of *higher-order* effects, where the argument to an effect operation is a function (in our case, the body of the **for** expression). Higher-order effects have been used in the context of *scoped effects* [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022] which delimit the scope of an effect, as well as *latent effects* [van den Berg et al. 2021] that defer parts of an effectful program. This work provides another kind of higher-order effects that serve a different purpose: they are useful for modelling parallel computations.

Parallelism with monads and applicative functors. For Haskell, Scholz [1995] describes a "concurrency monad" with threading primitives, and Claessen [1999] introduces a variant with a continuation-passing `Fork` operator. These works target an explicitly-concurrent programming style, where user code must launch threads and collect results. Marlow et al. [2014] show that applicative functors can be used to automatically parallelize data-access programs in Haskell, by taking advantage of the independence of `(<*>)` for a `Fetch` monad that enables data access. Marlow et al. [2016] further implement the `ApplicativeDo` extension into the GHC compiler, which rewrites operators in terms of the applicative combinator `<*>`, enabling parallelism opportunities.

10 Conclusion

In summary, we have presented a design for parallel effect handlers, where paralleliable **for** expressions are handled by traverse clauses in handlers, and non-effectful **for** expressions can be evaluated in parallel. We have also shown how a number of interesting handlers can be implemented in our system. Our design is type-safe and can be easily combined with the existing support for parallelism and effect handling in Haskell, and we are optimistic that our work can provide a foundation for designing new parallel algebraic effect handling systems.

As future work, we are interested in studying more properties of our design. Specifically, are there laws that we should expect all "reasonable" parallel effect handlers to satisfy? A strong restriction would be to require parallel programs to always yield the same results as the corresponding unrolled programs, but we have already seen that combinations of otherwise-benign effects (such as `accum` and `exc`) can yield different results under parallelism, and our PRNG handler also produces different samples between parallel and unrolled programs. It would be interesting to explore whether there are weaker restrictions that make the behavior of parallel programs more predictable without unnecessarily compromising the expressive power of the handlers. Moreover, we are interested in developing more examples where parallel effect handlers can be useful.

Acknowledgments

We thank the anonymous reviewers for helpful comments. Ningning Xie is partially funded by the Natural Sciences and Engineering Research Council of Canada.

A Unparameterized Variants of Example Handlers

Our examples in the main paper use parameterized handlers out of convenience, but they are not necessary. For instance, below we present an equivalent unparameterized version of the `accum` handler:

```
runAccum = λ(<>). λmempty.
  handler { return ↦ λx. (x, mempty), accum ↦ λx.λk. (v, s) ← k (); (v, x <> s),
            traverse ↦ (λn.λl.λk. pairs ← for i:n. (l.i) ();
                                   (results, outs) ← unzip pairs;
                                   out ← reduce (<>) outs;
                                   (res, out2) ← k results;
                                   (res, out <> out2)) } ()
```

We also we present the handler for PRNG as an unparameterized handler:

```
runRandom = λseed. λf.
  handle {
         return ↦ λx. (λkey. x),
         sampleUniform ↦ (λ_.λk. λkey.
             ⟨key1, key2⟩ ← splitKey key 2;
             u ← genUniform key1;
             k () u key2),
         traverse ↦ (λn.λl.λk. λkey.
             keys ← splitKey key (n + 1);
             results ← for i:n. (l.i () keys.i);
             k () results key2)
       } () (f ()) seed
```

B Parameterized Typed Parallel Effect Handlers

$$\text{expressions} \quad e ::= v \mid e_1\, e_2 \mid e\ \tau \mid \textbf{handle}\ h\ e_1\ e_2 \mid \textbf{for}\ x : \mathsf{Fin}\ n.\ e \mid e_1.e_2 \mid e \triangleright \tau$$

$\boxed{\Gamma \vdash e : \tau \mid \epsilon}$ *(Typing parameterized handler expressions)*

$$\frac{\Gamma \vdash_{\mathsf{h}} h : l \mid \tau \mid \epsilon \mid \rho \qquad \Gamma \vdash e_1 : \tau \mid \epsilon \qquad \Gamma \vdash e_2 : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \textbf{handle}\ h\ e_1\ e_2 : (\rho\ \sigma) \mid \epsilon}\ \text{T-HANDLE}$$

$\boxed{\Gamma \vdash_{\mathsf{h}} h : l \mid \tau \mid \epsilon \mid \rho}$ *(Typing parameterized handlers)*

$$\frac{\begin{array}{c} op : \forall \overline{a : \kappa}.\ \sigma_1 \longrightarrow \sigma_2 \in \Sigma(l) \qquad \Gamma \vdash_{\mathsf{v}} f_r : \forall a : \star.\ \tau \rightarrow \epsilon\ a \rightarrow \epsilon\ (\rho\ a) \qquad \Gamma \vdash_{\mathsf{wf}} \rho : \star \rightarrow \star \\ \Gamma \vdash_{\mathsf{v}} f_p : \forall (\overline{a : \kappa}).\ \forall (b : \star).\ \tau \rightarrow \epsilon\ \sigma_1 \rightarrow \epsilon\ (\tau \rightarrow \epsilon\ \sigma_2 \rightarrow \epsilon\ (\rho\ b)) \rightarrow \epsilon\ (\rho\ b) \\ \Gamma \vdash_{\mathsf{v}} f_t : \forall (a : \star)(b : \star).\ (n : \mathsf{Int}) \rightarrow \epsilon\ \tau \rightarrow \epsilon\ (\mathsf{Fin}\ n \Rightarrow (\tau \rightarrow \epsilon\ (\rho\ a))) \\ \rightarrow \epsilon\ (\tau \rightarrow \epsilon\ (\mathsf{Fin}\ n \Rightarrow a) \rightarrow \epsilon\ (\rho\ b)) \rightarrow \epsilon\ (\rho\ b) \end{array}}{\Gamma \vdash_{\mathsf{h}} \{return \mapsto f_r, op \mapsto f_p, traverse \mapsto f_t\} : l \mid \tau \mid \epsilon \mid \rho}\ \text{T-HANDLER}$$

C Typed Operational Semantics for Parameterized Handlers

We extend the untyped operation semantics of λ^p to be fully type annotated:

$$
\begin{array}{lllll}
(app) & (\lambda^\epsilon x : \tau.\ e)\ v & \longrightarrow\!\!\!\!\rightarrow & e[x := v] \\
(tapp) & (\Lambda a : \kappa.\ v)\ \tau & \longrightarrow\!\!\!\!\rightarrow & v[a := \tau] \\
(index) & \langle v_0, \ldots, v_{n-1}\rangle.i & \longrightarrow\!\!\!\!\rightarrow & v_i \\
(return) & \textbf{handle}\ h\ s\ v & \longrightarrow\!\!\!\!\rightarrow & f_r\ \sigma\ s\ v \\
 & & & \quad \text{if } (return \mapsto f_r) \in h \wedge \bullet \vdash_{\mathsf{v}} v : \sigma \\
(perform) & \textbf{handle}\ h\ s\ E[\textbf{perform}\ op\ \epsilon_0\ \overline{\tau}\ v] & \longrightarrow\!\!\!\!\rightarrow & f_p\ \overline{\tau}\ \sigma\ s\ v\ k \\
 & & & \quad \text{if } op \notin \mathrm{bop}(E) \wedge (op \mapsto f_p) \in h
\end{array}
$$

$$
\begin{array}{l}
\text{where } k = \lambda^\epsilon s : \tau.\ \lambda^\epsilon x : \sigma_2[\overline{a} := \overline{\tau}].\ \textbf{handle}\ h\ s\ E[x] \\
\vdash E[\textbf{perform}\ op\ \epsilon\ \overline{\tau}\ v] : \sigma \mid \langle l \mid \epsilon\rangle \\
\vdash_{\mathsf{h}} h : l \mid \tau \mid \rho \mid \epsilon \\
op : \forall \overline{a} : \overline{\kappa}.\ \sigma_1 \rightarrow \sigma_2 \in \Sigma(l)
\end{array}
$$

$$
\begin{array}{llll}
(traverse) & \textbf{handle}\ h\ s\ F[\textbf{for}\ x : \mathsf{Fin}\ n.\ e] & \longrightarrow\!\!\!\!\rightarrow & f_t\ \sigma_1\ \sigma_2\ n\ s\ \ell\ k \\
 & & & \quad \text{if } (traverse \mapsto f_t) \in h
\end{array}
$$

$$
\begin{array}{ll}
\text{where } & \ell = \langle \ell_1, \ell_2, \ldots, \ell_n \rangle \\
& \ell_i = \lambda^\epsilon (s : \tau).\ \textbf{handle}\ h\ s\ e[x := i \triangleright \mathsf{Fin}\ n] \\
& k = \lambda^\epsilon (s : \tau).\ \lambda^\epsilon (xs : \mathsf{Fin}\ n \Rightarrow \sigma_1).\ \textbf{handle}\ h\ s\ F[xs] \\
& x : \mathsf{Fin}\ n \vdash e : \sigma_1 \mid \epsilon \\
& \bullet \vdash F[\textbf{for}\ x : n.\ e] : \sigma_2 \mid \epsilon
\end{array}
$$

$$
\frac{e \longrightarrow\!\!\!\!\rightarrow e'}{E[e] \longmapsto\!\!\!\!\rightarrow E[e']}\ (\textsc{step}) \qquad \frac{\forall\ 0 \le i < n.\ e[x := i \triangleright \mathsf{Fin}\ n] \longmapsto\!\!\!\!\rightarrow v_i}{F[\textbf{for}\ x : \mathsf{Fin}\ n.\ e] \longmapsto\!\!\!\!\rightarrow F[\langle v_0, \ldots, v_{n-1}\rangle]}\ (\textsc{parallel})
$$

D Erasure

$$
\begin{array}{rcl}
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] \\
[\![e\ \tau]\!] & = & [\![e]\!] \\
[\![\textbf{handle}\ h\ e_1\ e_2]\!] & = & \textbf{handle}\ [\![h]\!]\ [\![e_1]\!]\ [\![e_2]\!] \\
[\![\textbf{for}\ x : \mathsf{Fin}\ n.\ e]\!] & = & \textbf{for}\ x : n.\ [\![e]\!] \\
[\![e_1.e_2]\!] & = & [\![e_1]\!].[\![e_2]\!] \\
[\![e \triangleright \tau]\!] & = & [\![e]\!] \\
[\![i]\!] & = & i \\
[\![x]\!] & = & x \\
[\![\lambda^\epsilon x : \tau.\ e]\!] & = & \lambda x.\ [\![e]\!] \\
[\![\Lambda a : \kappa.\ v]\!] & = & [\![v]\!] \\
[\![\langle v_0, \ldots, v_n \rangle]\!] & = & \langle [\![v_0]\!], \ldots, [\![v_n]\!] \rangle \\
[\![\textbf{perform}\ op\ \epsilon\ \overline{\tau}]\!] & = & \textbf{perform}\ op \\
[\![\{return \mapsto f_r, op \mapsto f_p, traverse \mapsto f_t\}]\!] & = & \{return \mapsto [\![f_r]\!], op \mapsto [\![f_p]\!], traverse \mapsto [\![f_t]\!]\}
\end{array}
$$

Lemma 6.1 (Type erasure of values). *If e is a value in F^p, then $[\![e]\!]$ is a value in λ^p.*

Proof. By a straightforward induction. Note that $[\![\Lambda a : \kappa.\ v]\!] = [\![v]\!]$ is a value by I.H.. □

Lemma D.1. *If $e_1 \longrightarrow\!\!\!\!\rightarrow e_2$, then either $[\![e_1]\!] \longrightarrow [\![e_2]\!]$, or $[\![e_1]\!] = [\![e_2]\!]$.*

Proof. By a straightforward induction. We talk about some interesting cases.

- (*app*). With Lemma 6.1 we have $[\![v]\!]$ being a value. The goal follows from the substitution property that $[\![e[x := v]]\!] = [\![e]\!][x := [\![v]\!]]$.
- (*tapp*). $(\Lambda a : \kappa.v)\ \tau \longrightarrow\!\!\!\!\rightarrow v[a := \tau]$. We have $[\![(\Lambda a : \kappa.v)\ \tau]\!] = [\![v]\!] = [\![v[a := \tau]]\!]$.

□

Theorem 6.2 (Semantics preservation). *If $e_1 \longmapsto\!\!\!\!\rightarrow e_2$, then either $[\![e_1]\!] \longmapsto [\![e_2]\!]$, or $[\![e_1]\!] = [\![e_2]\!]$.*

Proof. By induction on the derivation. The goal follows straightforwardly from I.H. and Lemma D.1. □

E Type Soundness

Lemma 6.4 (Progress with effects). *If $\bullet \vdash e : \tau \mid \epsilon$, then either e is a value, or there exists e' such that $e \longmapsto e'$, or $e = F[\textbf{for}\ i : \mathsf{Fin}\ n.\ e]$, or $e = E[\textbf{perform}\ op\ \epsilon'\ \overline{\tau}\ v]$ such that $op \notin \mathrm{bop}(E)$.*

Proof. By induction on the size of the expression e. Most cases follow directly from the induction hypothesis. We discuss below the interesting cases.

- rule T-VAL. In this case e is a value.
- rule T-TABLE. In this case e is a value.
- rule T-PRJ where $e_1.e_2$. If e_1 or e_2 is not a value, then the case follows from the induction hypothesis (and (*step*)). Otherwise we have $v_1.v_2$. According to typing, it must be $v_1 = \langle v'_0, \ldots, v'_{n-1} \rangle$ and $v_2 = i \triangleright \mathsf{Fin}\ n$ for some n where $i < n$. Thus $v_1.v_2 \longmapsto v'_i$ according to (*index*) and (*step*).
- rule T-FOR. In this case the third goal is satisfied.
- rule T-PERFORM. In this case the last goal is satisfied.
- rule T-HANDLE where $e = \textbf{handle}\ h\ e_1\ e_2$.
  According to induction hypothesis, we know either e_1 is a value, or it reduces, or it contains a **for**, or it contains an unhandled perform.
  - In the case it reduces, the whole expression reduces.
  - In the case it contains a **for**, the whole expression contains a **for**.
  - In the case it contains an unhandled perform, the whole expression contains an unhandled perform.
  - If e_1 is a value, then we discuss e_2. According to induction hypothesis, either e_2 is a value, or it reduces, or it contains a **for**, or it contains an unhandled perform.
    * If e_2 is a value, then the whole expression reduces by (*return*) and (*step*).
    * If e_2 reduces, then it takes either (*step*) or (*parallel*). If it takes (*step*), then the whole expression reduces by (*step*). If it takes (*parallel*), then the whole expression reduces by (*step*) and (*traverse*).
    * If e_2 contains a for, then the whole expression reduces by (*step*) and (*traverse*).
    * In the last case, e_2 contains an unhandled operation. If h handles it, then the expression reduces by (*step*) and (*perform*). Otherwise, the expression satisfies the last case of the goal.

□

Theorem 6.5 (Progress). *If $\bullet \vdash e : \tau \mid \langle\rangle$, then either e is a value, or there exists e' such that $e \longmapsto e'$.*

Proof. By induction on the expression, and the goal follows by Lemma 6.4. Since e is pure, for the third case, we can reduce the **for** by (*traverse*) according to I.H., and the fourth case is impossible. □

Theorem 6.3 (Type Preservation). *Given $\bullet \vdash e : \tau \mid \epsilon$, and $e \longrightarrow e'$, then $\bullet \vdash e' : \tau \mid \epsilon$.*

Proof. By induction on the evaluation step. We discuss the interesting cases.

- Case (*index*).

$\bullet \vdash \langle v_0, \ldots, v_n \rangle : \mathsf{Fin}\ n \Rightarrow \tau \mid \epsilon$	given
$\bullet \vdash v_i : \tau \mid \epsilon$	inversion and rule T-VAL

- Case (*return*).

$\bullet \vdash \textbf{handle}\ h\ s\ v : \rho\ \sigma \mid \epsilon$	given
$\bullet \vdash s : \tau \mid \epsilon$	inversion
$\bullet \vdash v : \sigma \mid \langle l \mid \epsilon \rangle$	inversion
$\bullet \vdash_{\mathsf{v}} f_r : \forall a : \star.\ \tau \to \epsilon\ a \to \epsilon\ \rho\ a$	inversion
$\bullet \vdash v : \sigma \mid \epsilon$	v is a value and by rule T-VAL
$\bullet \vdash f_r\ \sigma\ s\ v : \rho\ \sigma \mid \epsilon$	rules T-APP and T-TAPP

- Case (*perform*).

$\bullet \vdash \textbf{handle}\ h\ s\ E[\textbf{perform}\ op\ \epsilon_0\ \overline{\tau}\ v] : \rho\ \sigma \mid \epsilon$	given
$\bullet \vdash E[\textbf{perform}\ op\ \epsilon_0\ \overline{\tau}\ v] : \sigma \mid \langle l \mid \epsilon \rangle$	inversion
$\bullet \vdash s : \tau \mid \epsilon$	inversion and s is a value
$op : \forall \overline{a} : \overline{\kappa}.\ \sigma_1 \to \sigma_2 \in \Sigma(l)$	inversion
$\bullet \vdash_{\mathsf{wf}} \overline{\tau} : \overline{\kappa}$	inversion
$\bullet \vdash v : \sigma_1[\overline{a} := \overline{\tau}] \mid \epsilon$	inversion and v is a value
$\bullet \vdash_{\mathsf{v}} f_p : \forall (\overline{a : \kappa}).\ \forall (b : \star).$ $\tau \to \epsilon\ \sigma_1 \to \epsilon\ (\tau \to \epsilon\ \sigma_2 \to \epsilon\ (\rho\ b)) \to \epsilon\ (\rho\ b)$	inversion
$k = \lambda^{\epsilon} s : \tau.\ \lambda^{\epsilon} x : \sigma_2[\overline{a} := \overline{\tau}].\ \textbf{handle}\ h\ s\ E[x]$	
$\bullet \vdash k : \tau \to \epsilon\ \sigma_2[\overline{a} := \overline{\tau}] \to \epsilon\ (\rho\ \sigma) \mid \epsilon$	rules T-VAL, T-ABS, and T-TABS
$\bullet \vdash f_p\ \overline{\tau}\ \sigma\ s\ v\ k : \rho\ \sigma \mid \epsilon$	rules T-TAPP and T-APP

- Case (*traverse*).

$\bullet \vdash \textbf{handle}\ h\ s\ F[\textbf{for}\ x : n.\ e] : \rho\ \sigma_2 \mid \epsilon$	given
$\ell = \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$	given
$\ell_i = \lambda^{\epsilon}(s : \tau).\ \textbf{handle}\ h\ s\ e[x := i \triangleright \mathsf{Fin}\ n]$	given
$k = \lambda^{\epsilon}(s : \tau).\ \lambda^{\epsilon}(xs : \mathsf{Fin}\ n \Rightarrow \sigma_1).\ \textbf{handle}\ h\ s\ F[xs]$	given
$x : \mathsf{Fin}\ n \vdash e : \sigma_1 \mid \langle l \mid \epsilon \rangle$	inversion
$\bullet \vdash F[\textbf{for}\ x : n.\ e] : \sigma_2 \mid \langle l \mid \epsilon \rangle$	inversion
$\bullet \vdash n : \mathsf{Int} \mid \epsilon$	inversion and rule T-VAL
$\bullet \vdash s : \tau \mid \epsilon$	inversion
$\Gamma \vdash_{\mathsf{v}} f_t : \forall (a : \star)(b : \star).$ $(n : \mathsf{Int}) \to \epsilon\ \tau \to \epsilon\ \tau_l \to \epsilon\ \tau_k \to (\rho\ b)$	inversion
$\tau_l = (\mathsf{Fin}\ n \Rightarrow (\tau \to \epsilon\ (\rho\ a)))$	
$\tau_k = (\tau \to \epsilon\ (\mathsf{Fin}\ n \Rightarrow a) \to \epsilon\ (\rho\ b))$	
$\bullet \vdash_{\mathsf{v}} \ell_i : \tau \to \epsilon\ (\rho\ a)$	rules T-ABS and T-HANDLE
$\bullet \vdash_{\mathsf{v}} \ell : (\mathsf{Fin}\ n \Rightarrow (\tau \to \epsilon\ (\rho\ a)))$	rule T-ARRAY
$\bullet \vdash_{\mathsf{v}} k : (\tau \to \epsilon\ (\mathsf{Fin}\ n \Rightarrow \sigma_1) \to \epsilon\ (\rho\ \sigma_2))$	rules T-ABS, T-FOR, and T-HANDLE
$\bullet \vdash f_t\ \sigma_1\ \sigma_2\ n\ s\ \ell\ k : \rho\ \sigma_2 \mid \epsilon$	rules T-APP and T-TAPP

□

References

Danel Ahman. 2017. Handling fibred algebraic effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (dec 2017), 29 pages. https://doi.org/10.1145/3158095

Mario Blažević and Jacques Légaré. 2017. Packrats parse in packs. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/3122955.3122958

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (nov 2020), 30 pages. https://doi.org/10.1145/3428194

Paolo Capriotti and Ambrus Kaposi. 2014. Free Applicative Functors. In *MSFP*. https://api.semanticscholar.org/CorpusID:17313426

Koen Claessen. 1999. A poor man's concurrency monad. *Journal of Functional Programming* 9, 3 (1999), 313–323.

Koen Claessen and Michał H. Pałka. 2013. Splittable pseudorandom number generators using cryptographic hashing. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2503778.2503784

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2018. Concurrent system programming with effect handlers. In *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*. Springer, 98–117.

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (aug 2017), 29 pages. https://doi.org/10.1145/3110257

Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (oct 2022), 29 pages. https://doi.org/10.1145/3563445

Google. 2020. JAX PRNG Design. https://github.com/google/jax/blob/main/design_notes/prng.md

Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (jan 1991), 124–149. https://doi.org/10.1145/114005.102808

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development* (Nara, Japan) *(TyDe 2016)*. Association for Computing Machinery, New York, NY, USA, 15–27. https://doi.org/10.1145/2976022.2976033

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of functional programming* 27 (2017), e7.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th workshop on Mathematically Structured Functional Programming*. https://doi.org/10.4204/EPTCS.153.8

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden) *(WGP '14)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2633628.2633636

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI'12)*. 91–102. https://doi.org/10.1145/2103786.2103798

Sam Lindley, Connor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)* (Paris, France). 500–514. https://doi.org/10.1145/3009837.3009897

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 325–337. https://doi.org/10.1145/2628136.2628144

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 92–104. https://doi.org/10.1145/2976002.2976007

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (jan 2008), 1–13. https://doi.org/10.1017/S0956796807006326

Dave Menendez. 2013. Free Applicative Functors in Haskell. https://www.eyrie.org/~zednenem/2013/05/27/freeapp. Accessed: 2024-02-25.

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473593

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/237721.237794

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 238 (oct 2023), 26 pages. https://doi.org/10.1145/3622814

Ruben P. Pieters, Exequiel Rivas, and Tom Schrijvers. 2020. Generalized monoidal effects and handlers. *Journal of Functional Programming* 30 (2020).

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *18th European Symposium on Programming Languages and Systems* (York, UK) *(ESOP'09)*. 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper. *Electron. Notes Theor. Comput. Sci.* 319, C (Dec. 2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Enno Scholz. 1995. A concurrency monad based on constructor primitives: or, being first-class is not enough. (1995).

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. *Retrofitting Effect Handlers onto OCaml*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 266–278. https://doi.org/10.1145/2034773.2034811

Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL, Article 54 (jan 2024), 29 pages. https://doi.org/10.1145/3632896

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent effects for reusable language components. In *Asian Symposium on Programming Languages and Systems*. Springer, 182–201.

Andrew K Wright. 1995. Simple imperative polymorphism. *Lisp and symbolic computation* 8, 4 (1995), 343–355.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358

Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. 2020. Effect Handlers, Evidently. In *25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020)* (Jersey City, NJ). https://doi.org/10.1145/3408981

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. Structured handling of scoped effects. In *European Symposium on Programming*. Springer International Publishing Cham, 462–491.

Received 2024-02-28; accepted 2024-06-18