Parallel Algebraic Effect Handlers

Ningning Xie*
University of Cambridge
ningning.xie@cl.cam.ac.uk

Daniel D. Johnson*
Google Research
ddjohnson@google.com

Dougal Maclaurin
Google Research
dougalm@google.com

Adam Paszke
Google Research
apaszke@google.com

Abstract

Algebraic effects and 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, an untyped lambda calculus which models two key features, effect handlers and parallelizable computations, the latter of which takes the form of a **for** expression as inspired by the Dex programming language. We present various interesting examples expressible in our calculus, and provide a Haskell implementation. We hope this paper provides a basis for future designs and implementations of parallel algebraic effect handlers.

1 Introduction

Algebraic effects and handlers [9, 10] 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 [4], Eff [11], Frank [6] and Links [5]. Recent work has implemented effect handlers in Multicore OCaml [13] to support asynchronous I/O for concurrent programming.

As an example of effect handlers, consider the monadic encoding of the `state` effect, using the syntax of the untyped algebraic effect lambda calculus [15]:

```
handler { get → λx.λk. (λy. k y y)
        , set → λx.λk. (λy. k () x) }
(λ_. perform set 21;
     x ← perform get ();
     (λz. x + x)) 0 // 42
```

Here a **handler** takes a list of operation clauses, and a computation to be handled, which is represented as a unit-taking function. Inside each clause, `x` is the argument to the operation, `k` is the *resumption* captured by the handler, and each operation returns a function, where `y` is the monadic state that is threaded through computations. Within the computation, we **perform** a set operation[1], setting the state to `21`, and then `get` the state, double it and return the result. The initial state `z` is set to `0`. Evaluating the program gets us the result `42`.

From this example, it may seem that algebraic effects, just like *monads* [14], generally need to be executed *sequentially*. Indeed, in this case, **perform** `set 21` must be executed before **perform** `get ()`, or otherwise we could get the state wrong. Yet, recent work on Dex [8], a strict functional programming language for array programming, has shown that it is *possible* and *useful* to define *parallel effect handlers*. Specifically, Dex supports a built-in effect `Accum` which works similarly to the `State` effect, but can only be updated through an (infix) increment operation `(+=)` and is implicitly initialized with an identity element of the increment. `Accum` is handled by the built-in handler `runAccum`. We can write the following program in Dex that sums up an input array (we will introduce the syntax of Dex in Section 2):

```
sum = λx:(n⇒Int).
  (_, total) = runAccum λy.
    for i:n. y += x.i
  total
```

Importantly, in this case, `runAccum` is able to run the "loop" (introduced by the **for** construct) in parallel! The key reason `runAccum` can be run in parallel is that in Dex (1) updates (`+=`) to the accumulator reference (`y`) must be *additive contributions* over an associative operator, opening up the parallelism by a potential reassociation of reduction steps; and (2) there is no "read" operation in `Accum` so the state cannot be retrieved until `runAccum` is complete, making sure the increment in one iteration does not affect other iterations.

However, Dex only supports a few primitive effect handlers with built-in parallelization techniques. So the key questions we ask in this paper are: *is it possible to support user-defined algebraic effects that preserve parallelism? If so, what are their semantics*?

We offer the following contributions:

- We illustrate the interaction between algebraic effects and paralleliable computations that take the form of the **for** construct as in Dex (Section 2).
- We formalize λ^p, an untyped lambda calculus with parallel effect handlers (Section 3). Key to the design is to interpret **for** as an algebraic effect, handled by a novel *traverse* clause that allows user-defined behaviors around parallel regions.

*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`.

- We present various interesting examples enabled by our design (Section 4).
- We provide a Haskell implementation that captures the essence of λ^p. All examples presented in Section 4 can be encoded in the Haskell implementation.
- Finally, we discuss the challenges posed by our framework and future extensions that we hope to incorporate (Section 5).

2 Background: Dex

In this section, we present a brief overview of the Dex programming language, and we refer the reader to [8] for more detailed explanations.

Dex is a new array programming language with safe, efficient, and differentiable typed indexing. Below presents a Dex example that increments the elements in an array:

```
incr = λx:(n⇒Int). for i:n. x.i + 1
```

Here, `x` of type `n⇒Int` is an array indexed by indices of type `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`. The construct **for** builds an array, by repeatedly evaluating the body over the full range of the index type, which has to be finite. In this case, the result array has the same length as `x`, with every element increased by 1.

What is important in the above example is that Dex is able to run the **for** loop in parallel, since the loop iterations do not depend on each other. As argued by Paszke et al. [8], the **for** expression enables a natural programming style for numerical computing, while being regular enough to enable compilation to efficient dense data-parallel code that can be later executed on hardware accelerators such as GPUs. Related work on automatic parallelization of sequential loops can also be found at [12].

However, with **for** and array indexing alone, only pure maps over arrays can be expressed. What makes the **for** expression really useful is the ability to have effects in its body. For example, the built-in `State` effect makes it possible for a single loop iteration to influence evaluation of all subsequent iterations by modifying a value pointed to by a reference. Unfortunately, while achieving great generality, `State` has a significant drawback — **for** expressions with the `State` effect cannot be evaluated in parallel anymore. To resolve this issue, Dex additionally implements a *parallelism-friendly* `Accum` effect. It supports only a subset of operations that `State` can express, but it comes with the benefit of being parallelizable. `Accum` itself extends the power of **for** expression to array reductions. As an example, let us revisit the program from the introduction:

```
sum = λx:(n⇒Int).
  (_, total) = runAccum λy.
    for i:n. y += x.i
  total
```

In this example, `runAccum` takes a reference `y`, which is implicitly instantiated with `0` (the identity element of `Int` under addition), and then builds up an array, where each iteration calls the infix effectful operation `(+=)` that adds `x.i` to `y` and returns a unit. Therefore, the **for** builds an array of unit. The `runAccum` handles the operation, returning both the array (ignored by _), and the final reference value `total`, which is returned as the final result.

Unfortunately, while Dex can execute the **for** loop inside `runAccum` in parallel, at the moment 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 using type-driven analysis to find the data-parallel regions in user programs. This is highly unsatisfactory, and we outline many effects interesting for numerical computing that are unsupported by Dex today in Section 4.

3 Parallel Effect Handlers

In this section we introduce a calculus λ^p that lays out a basis for user-extensible parallel effects in Dex and should provide a guidance for future implementation. The syntax and semantics of λ^p are summarized in Figure 1.

3.1 Syntax

Expressions e include values v, applications $e\ e$, the **for** $x : n.\ e$ construct to build an array of length n, and the handle frame **handle** $h\ s\ e$, which is a *parameterized handler* [10] that takes a handler h, a local state s, and a computation e to be handled. Values v include literals i, variables x, lambdas $\lambda x.\ e$, arrays $\langle v_0, \ldots, v_n \rangle$, and **perform** op that performs an operation. We often use f for lambdas, n for literals, and s for state. While not included in the grammar, a **handler** $h\ s\ e$ construct that takes a suspended computation e to be handled can be defined as a syntactic sugar:

$$\textbf{handler}\ h\ s\ e \triangleq \lambda\_.\ \textbf{handle}\ h\ s\ (e\ ())$$

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 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. We distinguish between evaluation contexts E and *pure* evaluation context F that contains no **handle** frame. The notation $E[e]$ denotes an expression obtained by substituting e into the hole of E, e.g., $((v\ \square)\ f)[e] = (v\ e)\ f$.

expressions	e	::=	$v \mid e\ e \mid \textbf{for}\ x : n.\ e \mid \textbf{handle}\ h\ s\ e$
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 context	F	::=	$\square \mid F\ e \mid v\ F$
	E	::=	$\square \mid E\ e \mid v\ E \mid \textbf{handle}\ h\ s\ E$

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

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

Figure 1. Syntax and semantics of λ^p.

3.2 Operational Semantics

The bottom of Figure 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.

Primitive evaluation rules. We first discuss primitive evaluation rules. Rule (*app*) defines the standard call-by-value β-reduction. Rule (*index*) models the indexing operation in Dex as an application: applying an array $\langle v_0, \ldots, v_n \rangle$ to a literal i projects out the ith element v_i in the array.

Rule (*return*) and (*perform*) define the original standard dynamic 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 f_r to the value. This is useful to model, for example, exceptions, where an operation may cause the whole computation to return Nothing, while f_r can be Just that wraps the value when the computation returns normally. 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 $op \notin \mathrm{bop}(E)$), and applies the operation clause f_p to the state s, the operation argument v, as well as the resumption k. The resumption k takes a new handler state s and the operation result x, and captures the handler with the new state 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 computation calls **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 operations! Instead, we allow the users to define 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 state s, (3) the **for** expression 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 (we discuss sequential effect handlers in Section 5.2). Second, ℓ wraps the original **for** expression, but pushes the handler inside the **for** expression, and thus the corresponding operations in e can be handled by h. Third, we need to update the state for the handlers in ℓ. To this end, ℓ takes an array of states ss, and during each iteration the handler takes the corresponding state from the array by indexing $(ss\ x)$.

Now depending on the implementation of f_t, the program can have different behaviors. (i) f_t may never call ℓ. Then the whole computation of the **for** expression is discarded. (ii) f_t may call ℓ exactly once. 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 discuss shortly). (iii) f_t may call ℓ multiple times, then the same **for** expression

will be evaluated multiple times. We consider some examples of different behaviors. For instance, if the handler has no special behavior for parallelism, a default implementation of the traverse clause can be (case (ii)):

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

The handler may also just ignore ℓ and pass something totally different to k (case (i)):

$$\mathit{traverse} \mapsto \lambda n.\ \lambda s.\ \lambda \ell.\ \lambda k.\ k\ s\ \langle 1, 2, 3\rangle$$

Note how this corresponds nicely to how handlers handle resumptions: a resumption may never be called (e.g., for exception handlers), or called exactly once (for most handlers including, e.g., reader), or called multiple times (for non-determinisim).

Evaluation inside evaluation contexts. Now we turn to the rules of $\longmapsto$, which evaluates 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 a built-in parallelism support for evaluating the $\forall$ parallelism (which can be, for example, the built-in parallelism support for **for** in Dex).

Example. Now let us consider a paralleliable reader handler as an example. For the sake of readability, in the example we ignore handler states, and let ℓ take a unit.

$$\begin{array}{rcl} h & = & \{\ \mathit{return} \mapsto \lambda x.\ x,\ \mathit{ask} \mapsto \lambda x.\ \lambda k.\ k\ 42 \\ & & ,\ \mathit{traverse} \mapsto \lambda n.\ \lambda \ell.\ \lambda k.\ k\ (\ell\ ())\ \} \\ \ell & = & \lambda\_.\ \textbf{for}\ x : 5.\ \textbf{handle}\ h\ (\textbf{perform}\ \mathit{ask}\ ()) \\ k & = & \lambda xs.\ \textbf{handle}\ h\ xs \end{array}$$

Now we have (we use $\longmapsto^*$ as the transitive closure of $\longmapsto$):

$$\begin{array}{rll} & \textbf{handle}\ h\ (\textbf{for}\ x : 5.\ \textbf{perform}\ \mathit{ask}\ ()) & \\ \longmapsto^* & (\lambda n.\ \lambda \ell.\ \lambda k.\ k\ (\ell\ ()))\ 5\ \ell\ k & (\mathit{traverse}) \\ \longmapsto^* & k\ (\ell\ ()) & (\mathit{app}) \\ \longmapsto^* & k\ (\textbf{for}\ x : 5.\ \textbf{handle}\ h\ (\textbf{perform}\ \mathit{ask}\ ())) & (\mathit{app}) \\ \longmapsto^* & k\ \langle 42, 42, 42, 42, 42\rangle & (\mathit{parallel}) \\ \longmapsto^* & (\lambda x.\ x)\ \langle 42, 42, 42, 42, 42\rangle & (\mathit{return}) \\ \longmapsto^* & \langle 42, 42, 42, 42, 42\rangle & (\mathit{app}) \end{array}$$

See also Appendix A for a more sophisticated reduction that uses handler states to implement our Accum effect.

4 Examples

Now that we have described our system, in this section we will show how we can implement a variety of interesting and useful effects. We will express these examples using a richer surface language that includes tuples, conditionals, infix operators, algebraic data types, numbers, and strings.

4.1 Accum

We begin by showing how to express the parallel accumulation effect in our language. To handle the effect, we must provide an associative binary operation (<>) and an identity for that operation (essentially forming a *monoid*). For instance, to sum an array of numbers, we can use:

```
sum = (λxs.
  (_, total) ← runAccum (+) 0 (λ_.
    for i:(length xs). perform accum (xs i));
  total)
```

Because the binary operator is associative, we are free to independently compute results for each list iteration, and then combine them in our traverse handler.

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

Here we assume the existence of a parallelizable function `reduce` which reduces a table into a single value by applying (<>). Due to space limitations we do not implement `reduce` here; roughly, it corresponds to a parallel reduction circuit of depth $O(\log n)$ constructed by forming a balanced binary tree over array elements and applying (<>) at each node.

4.2 Weak Exceptions

Our effect system can also express a form of exception handling. However, since loop iterations are always evaluated in parallel, these exceptions are "weak": an exception in one iteration of a loop does not interrupt execution in any other iterations, although it will still prevent execution of the code after the loop body. Our handler has the following form:

```
data Either a b = Left a | Right b
runWeakExcept = λf.
  handle { return ↦ λ_.λx. Right x,
           throw ↦ λ_.λerr.λk. Left err,
           traverse ↦ (λn.λ_.λl.λk.
              eithers ← l (for i:n. ())
                combined ← firstFailure eithers
                case combined of
                  Left err → Left err
                  Right res → k () res
         } () (f ())
```

Here `firstFailure` takes a table of Eithers and returns either the first Left, or the table of values if all values were wrapped in Right; it can be implemented in terms of `reduce`.

We can observe the "weak" nature of these exceptions by combining them with another effect:

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

In this example, all loop iterations execute their effects in parallel, and then computation aborts at the end of the for loop. (Here (++) is the string concatenation operator.)

4.3 PRNG

One effect that is particularly useful for real-world numerical computation is the generation of (pseudo)random numbers. However, doing so in a parallelizable way is nontrivial. Suppose we want to run a computation like this in parallel:

```
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
```

This example computes a binomial random variable by summing a collection of weighted coin flips, then scales it by another random variable, and we want each coin flip to draw distinct random numbers, but also execute in parallel.

One way to handle this is using a "splittable PRNG" [1], whose state (called a "key") can be split into arbitrarily many independent streams of random numbers; this technique is used to implement accelerator-friendly random numbers in the library JAX [2]. 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:

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

Here we transform effectful expressions into functions from a PRNG key to a value. We handle sampleUniform by splitting the key, then using one result to generate the uniform and the other to run the continuation. We handle **for** loops similarly, except that the first key is split again to generate independent streams of random numbers for each loop iteration.

An interesting property of this handler is that the following computations may have different results:

```
result_1 = runRandom shared_seed (λ_.
  u0 ← perform sampleUniform ();
  u1 ← perform sampleUniform ();
  u2 ← perform sampleUniform ();
  ⟨u0, u1, u2⟩)

result_2 = runRandom shared_seed (λ_.
  for i:3. perform sampleUniform ())
```

This highlights an important property of the **for** construct in λ^p, in contrast to similar looping constructs in other languages: the semantics of a program with a **for** expression is not necessarily equivalent to the semantics of a program with a sequentially-unrolled loop in its place.

4.4 Amb

Our final example is the Amb effect [7] (also known as the list monad). Conceptually, the amb operator takes as argument a table 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 table of *all* possible results we might obtain:

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

Unlike the other effect handlers we have introduced, the handler for amb can introduce parallelism into sequential code, by calling the continuation inside a parallel loop:

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

Here `cartesianProd` is a function which takes a length-m table of variable-length tables, and returns a variable-length (called n) table of length m tables, such that each element of the result is formed by taking one element from each of the original variable-length tables.

Due 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:

```
// How many pairs of single-digit numbers add up
// to 13?
(_, count) = 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 Discussion

In this section we first outline the Haskell implementation of λ^p, and then discuss potential limitations and future directions for the work outlined in this text.

5.1 Haskell implementation

We provide an implementation of our parallel effect system in Haskell, along with Haskell versions of each of the examples described in Section 4. For ease of implementation and use, we build our implementation on top of Haskell's own interpreter and inherit its execution semantics. As such, the implementation may not actually run loop iterations in parallel, and reductions may not be applied in the same order as specified in Figure 1. Nevertheless, it exhibits the same overall behavior as λ^p, including in particular the interaction between **for** and `traverse`.

Our Haskell implementation is based on the Free monad and the FEFree monad as described by Kiselyov and Ishii [3], which represents an effectful computation by capturing the first performed effect (the equivalent of $E[\textbf{perform}\ op\ v]$ in λ^p) and storing it as a pair of a pure value (e.g. v) and continuation ($\lambda x.\ E[x]$). We introduce a Haskell type for arrays, along with a special function **for** to construct them. We then construct a type for parallel effectful computations (`EffComp effectRow a`) which captures either the first effect (as in FEFree) or the first for loop (the equivalent of $F[\textbf{for}\ x : n.\ e]$ in λ^p), storing the latter as a pair of loop iteration function $\lambda x.\ e$ and continuation $\lambda ys.\ F[ys]$. We then implement the reduction rules for *(perform)*, *(traverse)*, and *(parallel)* by recursively pattern-matching on this data type. See appendix B for more details, and the supplemental material for the implementation itself.

Although λ^p is untyped, the Haskell implementation suggests one way to formalize the type system of parallel algebraic effects. One interesting feature is that *return* (along with **perform** and *traverse*) is polymorphic in its output type, since the handler may be locally applied to the body of multiple **for** expressions with different types. If a handler needs to change the return type, it must do so by wrapping the polymorphic return type `a` with a functor-like higher-order type `f a`. We leave the details of a typed version of λ^p to future work.

5.2 Sequential algebraic effects

So far, our focus was the design of a formalism for an extensible *parallel* effect system. While the final result is sufficiently powerful to model a Dex-like accumulation effect and a wide range of other examples, it does fall short of modeling effects that are *sequential*, in the sense that no viable `traverse` method exists for them. One good example is the state effect, which requires the **for** iterations to be executed in order, since arbitrary stateful updates from earlier iterations have to be visible in the subsequent ones. Dex implements a `State` effect along with `Accum`, and uses a type-directed compilation strategy to parallelize loops that are pure or only use associative accumulation, while the **for** expressions with bodies that have a `State` effect are compiled as sequential loops. We hope to address this limitation in future work, which should present a coherent effect system unifying the sequential and parallel execution strategies for effects, and hence model the design of Dex even more closely.

5.3 Handlers returning functions

One interesting property of the formalism presented here is that every handler has to be associated with a functorial data type, which follows from the fact that the traverse rule of Figure 1 constructs an array of elements of completely arbitrary type, by wrapping their computation in the same handler. So far we have seen examples where this functor is well-behaved, but it turns out that it is not always the case. In particular, if the handler returns the continuation itself without calling it (corresponding to the functor `(a →)` for some `a`), it can break up a single parallel loop into two: one containing the effects outside each iteration's continuation and the other containing the effects inside them. While initially this might seem relatively benign, this introduces synchronization points that can significantly alter the semantics of the program in quite surprising ways.

In fact, this behavior is one of our main reasons for presenting a "stated" version of our effect system. The usual trick of encoding state by making the handler return a result of type `s → (a, s)` does not work in our paradigm, because it forces an additional and unwanted synchronization of the **for** loop. One way to limit this issue would be to forbid the functor type associated with handler to include the function type, effectively making the functions second-class from the

point of the effect system. To some extent this is already the case in Dex, where e.g. the type of values valid to be carried in a `State` effect is restricted to never include functions, so as to enable elimination of higher-order functions (a specialized form of defunctionalization).

6 Conclusion

In summary, we have designed a calculus λ^p for parallel effect handlers, where paralleliable **for** expressions are handled by the traverse clause in handlers, and eventually non-effectful **for** expressions can run in parallel. As future work, we would like to investigate a typed formalism of λ^p that can also express sequential algebraic effect handlers, and implement λ^p as a source-to-source transformation in Dex.

References

[1] Koen Claessen and Michał H Pałka. 2013. Splittable pseudorandom number generators using cryptographic hashing. *ACM SIGPLAN Notices* 48, 12 (2013), 47–58.

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

[3] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.

[4] 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

[5] 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

[6] 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)*. 500–514. https://doi.org/10.1145/3009837.3009897

[7] John McCarthy. 1961. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*. 225–238.

[8] 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

[9] 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

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

[11] 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

[12] Tiark Rompf and Kevin J. Brown. 2017. Functional Parallels of Sequential Imperatives (Short Paper). In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2017)*. Association for Computing Machinery, New York, NY, USA, 83–88. https://doi.org/10.1145/3018882.3018891

[13] 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

[14] Philip Wadler. 1995. Monads for functional programming. In *International School on Advanced Functional Programming*. Springer, 24–52.

[15] 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)*. https://doi.org/10.1145/3408981

A Example reduction of accum effect

Here we show the steps taken while reducing our accumulation example, according to the reduction rules of λ^p. Consider the following program and handler (reproduced from section 4.1):

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

sum = (λxs.
  (_, total) ← runAccum (+) 0 (λ_.
    for i:(length xs). perform accum (xs i));
  total)

value = sum ⟨1, 2, 3⟩
```

Abbreviating our handler as h, and following the reduction rules described in fig. 1, we obtain the following sequence:

```
sum ⟨1, 2, 3⟩
// (app)
(_, total) ← handle h 0 (
    for i:3. perform accum (⟨1, 2, 3⟩ i));
total
// (traverse)
n ← 3;
s ← 0;
l ← λss. for i:n. handle h (ss i) (
    perform accum (⟨1, 2, 3⟩ i));
k ← λs. λx. handle h s x;
(_, total) ← (
    pairs ← l (for i:n. 0);
    results ← for i:n. (fst (pairs i));
    outs ← for i:n. (snd (pairs i));
    out ← reduce (+) outs;
    k (s + out) results);
total
// ...
pairs ← (λss. for i:3. handle h (ss i) (
    perform accum (⟨1, 2, 3⟩ i)) (for i:3. 0)
results ← for i:3. (fst (pairs i))
outs ← for i:3. (snd (pairs i))
out ← reduce (+) outs;
(_, total) ← (λs. λx. handle h s x)
    (0 + out) results;
total
// ...
pairs ← for i:3. handle h ((for i:3. 0) i) (
    perform accum (⟨1, 2, 3⟩ i))
results ← for i:3. (fst (pairs i))
outs ← for i:3. (snd (pairs i))
out ← reduce (+) outs;
(_, total) ← (λs. λx. handle h s x)
    (0 + out) results;
total
```

At this point, the rule (*parallel*) applies, so we independently reduce each iteration to a value. For $i = 0$:

```
handle h ((for i:3. 0) 0) (
    perform accum (⟨1, 2, 3⟩ 0))
// (parallel)
handle h (⟨0, 0, 0⟩ 0) (perform accum (⟨1, 2, 3⟩ 0))
// (index)
handle h 0 (perform accum 1)
// (perform)
s ← 0;
v ← 1;
k ← λx. λs. handle h s x;
k (s + v) ()
// ...
handle h 1 ()
// (return)
((), 1)
```

The other iterations reduce to $((), 2)$ and $((), 3)$ respectively. We then resume reducing the full program using (*step*):

```
// (parallel)
pairs ← ⟨((), 1), ((), 2), ((), 3)⟩;
results ← for i:3. (fst (pairs i))
outs ← for i:3. (snd (pairs i))
out ← reduce (+) outs;
(_, total) ← (λs. λx. handle h s x)
    (0 + out) results;
total
// ...
results ← ⟨(), (), ()⟩;
outs ← ⟨1, 2, 3⟩;
out ← reduce (+) outs;
(_, total) ← (λs. λx. handle h s x)
    (0 + out) results;
total
// ...
out ← 6;
(_, total) ← (λs. λx. handle h s x)
    (0 + out) ⟨(), (), ()⟩;
total
// ...
(_, total) ← handle h 6 ⟨(), (), ()⟩;
total
```

```
// (return)
(_, total) ← (⟨(), (), ()⟩, 6);
total
// ...
6
```

B Overview of Haskell implementation

Here we give a brief overview of the structure of the Haskell implementation. For all of the details, including Haskell versions and execution results for each of the example handlers and programs in section 4, see the supplemental material.

We define a type for tables (arrays), using Haskell's type-level literals to annotate the table length (as an approximation of Dex's table arrow and `Fin n` type):

```
data Table (n :: Nat) a where
  UnsafeFromList :: forall n a. KnownNat n
                 => [a] -> Table n a
```

Following the FEFree monad [3], we assume that effects are functor-like higher kinded types parameterized by the result of each effect, e.g. for an effect `State s :: * -> *` we might have `Get :: State s s` and `Put :: s -> State s ()`. We additionally add a notion of lifting effects into ordered "effect rows":

```
type EffSig = * -> *
data EffCons (sig :: EffSig) (sigs :: EffSig) r =
    Here (sig r) | There (sigs r)
infixr 5 `EffCons`
data EffNil r  -- pure computation; no operations

class HasEff (sig :: EffSig) (row :: EffSig)
    where liftEff :: sig r -> row r

instance HasEff sig (sig `EffCons` rest)
    where liftEff op = Here op

instance HasEff sig rest
         => HasEff sig (other `EffCons` rest)
    where liftEff op = There (liftEff op)
```

For instance, `State s `EffCons` Except e `EffCons` EffNil` is the signature that states `State s` and `Except e` are both possible operations, and a value of type `(State s `EffCons` Except e `EffCons` EffNil) r` represents either a state operation or an except operation whose return type is `r`, tagged with which handler should handle it. `liftEff` then finds the handler that should handle a given effect. (For readability, we have omitted a helper type family that enables Haskell to avoid an overlapping instance error by always choosing the leftmost handler if possible.)

Given a particular effect row, our parallelism-aware FEFree-like monad is defined as

```
data EffComp (sig :: EffSig) r where
  -- This computation is just a pure value.
  Pure :: r -> EffComp sig r
  -- This computation is equivalent to
  -- an E[perform op v] or F[for i:n. e]
  Impure :: EffOrTraversal sig r
         -> (r -> EffComp sig s)
         -> EffComp sig s

-- Helper which holds either an operation,
-- or a parallel loop.
data EffOrTraversal sig r where
  -- equivalent of "perform op v"
  Effect :: sig r -> EffOrTraversal sig r
  -- equivalent of "for i:n. e"
  TraverseTable :: Table n (EffComp sig s) ->
    EffOrTraversal sig (Table n s)

-- Sequencing computations using a monad
instance Monad (EffComp sig) where
  return = Pure
  (Pure v >>= f) = f v
  (Impure va vc >>= f) =
    Impure va $ \a -> vc a >>= f
```

We construct the Haskell equivalent of the **perform** and **for** expressions:

```
perform :: HasEff sig union
        => sig r -> EffComp union r
perform e = Impure (Effect $ liftEff e) Pure

iota :: forall n. KnownNat n => Table n Int
iota = let nv = fromIntegral $ natVal (Proxy @n)
       in UnsafeFromList @n [0 .. nv - 1]

traverseTable :: (a -> EffComp sig b)
              -> Table n a
              -> EffComp sig (Table n b)
traverseTable f a =
  Impure (TraverseTable (f <$> a)) Pure

for :: KnownNat n => (Int -> EffComp sig b)
                  -> EffComp sig (Table n b)
for = flip traverseTable iota
```

A handler is defined using a Haskell record containing the three functions required by the handler. The type of a handler is parameterized by the operations it handles, the remaining effects in the row, the type of state it carries, and a functor `f` such that, if the original computation produces an `a`, the handler produces an `f a`. (For instance, for `Except`

e we have `f = Either e`, and thus th ehandler produces values of type `Either e a`.)

```
data ParallelizableHandler (op :: EffSig)
       (m :: * -> *) s (f :: * -> *) =
  ParallelizableHandler
  { handleReturn   :: forall a.
      s -> a -> m (f a)
  , handlePerform  :: forall a b.
      s -> op a -> (s -> a -> m (f b)) -> m (f b)
  , handleTraverse :: forall a b n. KnownNat n =>
      s -> (Table n s -> m (Table n (f a)))
        -> (s -> Table n a -> m (f b))
        -> m (f b)
  }
```

Finally, we provide functions to handle individual effects in an effect row, as well as to execute pure code. Our handler is represented as a Haskell record containing the three functions required by the handler.

```
runPure :: EffComp EffNil r -> r
runPure = \case
  Pure r -> r
  Impure (Effect eff) cont ->
    case eff of {} -- no operations, not possible
  Impure (TraverseTable iters) cont ->
    runPure $ cont $ fmap runPure iters

handle :: forall op rest s f a
    . ParallelizableHandler op (EffComp rest) s f
   -> s
   -> EffComp (op `EffCons` rest) a
   -> EffComp rest (f a)
handle h s comp = case comp of
    -- (return)
    Pure r -> (handleReturn h) s rv
    -- (perform)
    Impure (Effect (Here op)) cont ->
        (handlePerform h) s op (\s a ->
              handle h s $ cont a)
    -- (traverse)
    Impure (TraverseTable (iters :: Table n b))
           cont ->
        (handleTraverse h) s runIters runCont
    where
        runIters ss = for $ \i ->
            handle h (tableIndex ss i)
                     (tableIndex iters i)
        runCont s a = handle h s (cont a)
    -- ignore other operations
    Impure (Effect (There op)) cont ->
        Impure (Effect op) (handle h s . cont)
```