Toward Verified Library-Level Choreographic Programming with Algebraic Effects

GAN SHEN, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

Choreographic programming (CP) is a paradigm for programming distributed applications as single, unified programs, called *choreographies*, that are then compiled to node-local programs via *endpoint projection* (EPP). Recently, *library-level* CP frameworks have emerged, in which choreographies and EPP are expressed as constructs in an existing host language. So far, however, library-level CP lacks a solid theoretical foundation.

In this paper, we propose modeling library-level CP using *algebraic effects*, an abstraction that generalizes the approach taken by existing CP libraries. Algebraic effects let us define choreographies as computations with user-defined effects and EPP as location-specific effect handlers. Algebraic effects also lend themselves to reasoning about correctness properties, such as soundness and completeness of EPP. We present a prototype of a library-level CP framework based on algebraic effects, implemented in the Agda proof assistant, and discuss our ongoing work on leveraging the algebraic-effects-based approach to prove the correctness of our library-level CP implementation.

1 INTRODUCTION

Choreographic programming (CP) [Montesi 2013, 2023] is a paradigm for programming distributed applications that run on multiple nodes. In CP the programmer writes one, unified program, called a *choreography*, that is then compiled to individual programs for each node via a compilation step called *endpoint projection* (EPP). For example, the following choreography describes a distributed data processing pipeline, involving nodes Alice, Bob, and Carol, which respectively run functions f, g, and h on their input:

$$
\begin{aligned}
x &\leftarrow \text{Alice} \triangleright \mathsf{getInput} \\
y &\leftarrow \text{Alice} \Rightarrow \text{Bob} \square f(x) \\
z &\leftarrow \text{Bob} \Rightarrow \text{Carol} \square g(y) \\
w &\leftarrow \text{Carol} \Rightarrow \text{Alice} \square h(z) \\
&\quad \text{Alice} \triangleright \mathsf{showResults}(w)
\end{aligned}
$$

Here, we use $\leftarrow$ for variable bindings; Alice $\triangleright\, t$ denotes a local computation t at Alice; and Alice $\Rightarrow$ Bob $\square\, t$ denotes communication from Alice to Bob with message t. In this choreography, Alice first gets some input locally, processes it with f, and passes the result to Bob, who processes it with g and passes the result to Carol, who processes it with h and passes it back to Alice to be displayed to the user. To get an executable program for each node, we can apply endpoint projection to the choreography, resulting in individual programs for Alice, Bob, and Carol:

```
x ← getInput                 | y ← recv(Alice)              | z ← recv(Bob)
    send(Bob, f(x))          |     send(Carol, g(y))       |     send(Alice, h(z))
w ← recv(Carol)              |                              |
    showResults(w)           |                              |
```

A correct CP language guarantees soundness and completeness of EPP, which further implies that the collection of projected programs is deadlock-free when running together. Existing research

Authors' addresses: Gan Shen, University of California, Santa Cruz, USA; Lindsey Kuper, University of California, Santa Cruz, USA.

places CP on a solid theoretical foundation [Montesi 2013; Cruz-Filipe and Montesi 2020; Cruz-Filipe et al. 2022; Hirsch and Garg 2022], which has informed the design of practical, full-featured standalone CP languages such as Choral [Giallorenzo et al. 2024].

Recent work has introduced *library-level* CP languages [Shen et al. 2023; Kashiwa et al. 2023], in which choreographies and EPP are completely expressed as constructs in an existing host language. For example, HasChor [Shen et al. 2023], implements support for CP by means of a domain-specific language embedded in Haskell. In HasChor, choreographies are monadic computations in which choreographic operators such as _ ▷ _ and _ ⇒ _ □ _ may be used, and EPP is carried out by means of *dynamic interpretation* of choreographies at run time. The recently proposed ChoRus library for choreographic programming in Rust [Kashiwa et al. 2023] takes a similarly dynamic approach. Library-level CP frameworks have the potential to improve the accessibility and practicality of CP by integrating it into general-purpose programming languages. However, there are no proofs of correctness of EPP for library-level CP frameworks. Indeed, it is unclear to what extent the established theory of CP is applicable in the setting of library-level CP.

To close this gap, in this paper we propose *algebraic effects* [Plotkin and Power 2003; Plotkin and Pretnar 2013] as a foundational approach for implementing and verifying library-level CP. Algebraic effects provide an abstraction that generalizes existing approaches to library-level CP. In particular, they allow us to define choreographies as computations with user-defined effects and EPP as location-specific effect handlers. Algebraic effects also lend themselves to proofs of correctness. They provide abstract syntax trees for choreographies in which CP-specific effects and control flows are manifest, enabling reasoning about them. Furthermore, given that algebraic effects are "going mainstream" [Sivaramakrishnan et al. 2018], with efficient implementations now available in languages such as OCaml [Sivaramakrishnan et al. 2021] and Koka [Leijen 2017], we believe our proposed approach would make library-level CP less ad-hoc and bring it to a broader audience.

In the rest of the paper, we set up a framework for programming with algebraic effects in Agda (Section 2), which we then use to implement a prototype library-level CP framework (Section 3). Finally, we discuss our ongoing work on leveraging our approach to prove the correctness of our library-level CP implementation (Section 4). This paper is a literate Agda program.

2 A MINIMAL ALGEBRAIC EFFECTS FRAMEWORK IN AGDA

In this section, we define a minimal algebraic effects framework in Agda, which we will use to implement CP in the next section. No prior knowledge of algebraic effects is assumed. We introduce each concept first from a mathematical perspective and then relate it to programming. Due to lack of space, we do not include any examples in this section, but the next section can be seen as a demonstration of the framework. Our presentation is influenced by Bauer [2019] and Kidney et al. [2024], and we refer the reader to them for a comprehensive introduction to algebraic effects.

2.1 Signatures and Algebras

A signature Sig specifies the equipped operations of an algebra, which includes a type Op of operations and a function Arity giving the number of arguments (represented as the cardinality of a type) of each operation:[1]

```
record Sig : Set₂ where
  constructor _◁_
```

[1]Agda uses an infinite hierarchy of universes where Set : Set_1 : ... : Set_n : Set_{n+1} to avoid paradoxes. For ease of presentation in this paper, we overconstrain the universe of Op to be Set_1 (similarly for Arity). The actual implementation is universe-polymorphic, but readers can safely ignore the universe hierarchy without missing the key point of the paper.

```
field
  Op : Set₁
  Arity : Op → Set
```

A set X that implements the operations of a signature $\mathbb{F}$ is called an $\mathbb{F}$-algebra. An $\mathbb{F}$-algebra on the carrier set X, written as $\mathbb{F}$ -Alg[X] in Agda, is a function of type ⟦ $\mathbb{F}$ ⟧ $X \to X$:

```
⟦_⟧ : Sig → Set₁ → Set₁
⟦ Op ◁ Ar ⟧ X = Σ[ o ∈ Op ] (Ar o → X)

_-Alg[_] : Sig → Set₁ → Set₁
𝔽 -Alg[ X ] = ⟦ 𝔽 ⟧ X → X
```

⟦ $\mathbb{F}$ ⟧ X denotes an operation paired with its arity number of elements from the carrier set — a fully applied operation. ⟦ $\mathbb{F}$ ⟧ $X \to X$ allows us to make a new element out of a fully applied operation, the very nature of an algebra.

In programming, we can view an effectful operation as giving rise to an algebra, with the allowed effects being the signature, the result of an effect being the arity of the operation, and a carrier set of such an algebra being a functional model of the effectful computation.

2.2 The Free Algebra

Among all the algebras of a signature $\mathbb{F}$, we are particularly interested in one called the *free algebra*. Rather than performing operations in the carrier set, the free algebra merely records them as a data type, which we call Term:

```
data Term (𝔽 : Sig) (A : Set) : Set₁ where
  var : A → Term 𝔽 A
  op : ⟦ 𝔽 ⟧ (Term 𝔽 A) → Term 𝔽 A
```

The var constructor denotes a variable drawn from some set A. The op constructor denotes a fully applied operation to some other terms.

In programming, Terms correspond to programs where effects are left uninterpreted, with variables being pure computations and operations being effectful computations. Terms also form a monad (it is actually the free monad), which allows us to chain them together:

```
return : ∀ {𝔽} {A} → A → Term 𝔽 A
return = var

_»=_ : ∀ {𝔽} {A B} → Term 𝔽 A → (A → Term 𝔽 B) → Term 𝔽 B
var x    »= f = f x
op (o , k) »= f = op (o , _»= f ∘ k)
```

We also provide a helper function perform, which constructs a Term that performs an operation and immediately returns its result:

```
perform : ∀ {𝔽} (o : Op 𝔽) → Term 𝔽 (Arity 𝔽 o)
perform o = op (o , var)
```

2.3 Effect Handlers

One of the reasons why Terms are called the free algebra of $\mathbb{F}$ is that they are freely interpretable. Given another $\mathbb{F}$-algebra X and a substitution of variables from X, we can interpret a term as an element of X:

```
interp : ∀ {𝔽} {X A} → 𝔽 -Alg[ X ] → (A → X) → Term 𝔽 A → X
interp alg f (var x)      = f x
interp alg f (op (o , k)) = alg (o , interp alg f ∘ k)
```

For variables, interp uses the substitution f to map them to X. For operations, interp first recursively interprets the arguments and then uses *alg* to make a new element of X from a fully applied operation.

In programming, interp forms the foundation of effect handlers. It lets us systematically handle uninterpreted effects of a computation.

3 CHOREOGRAPHIC PROGRAMMING WITH ALGEBRAIC EFFECTS

In this section, we use the algebraic effects framework from the previous section to implement a CP language. Our CP language is simplified and only contains local computations and communication. We start by defining processes (Section 3.1), which are the results of endpoint projection. Then, we move on to defining choreographies (Section 3.2). Unlike previous library-level CP languages, our choreographies abstract over a particular representation of located values, allowing us to erase them and avoid non-totality. Finally, we define endpoint projection as location-specific effect handlers for choreographies (Section 3.3).

We assume a local language of signature $\mathbb{L}$ that each node uses for local computations, and we parameterize our CP language by it. We use locations Loc to refer to nodes in a distributed system and define them as Strings. However, any data type with decidable equality would suffice.

3.1 Processes

```
data Op : Set₁ where
  `locally : ∀ {A} → Term 𝕃 A → Op
  `send    : ∀ {A : Set} → Loc → A → Op
  `recv    : ∀ {A : Set} → Loc → Op

Arity : Op → Set
Arity (`locally {A} _) = A
Arity (`send _ _)      = ⊤
Arity (`recv {A} _)    = A

ℙ : Sig
ℙ = Op ◁ Arity
```

```
Process : Set → Set₁
Process A = Term ℙ A

locally : ∀ {A} → Term 𝕃 A → Process A
locally t = perform (`locally t)

send : ∀ {A} → Loc → A → Process ⊤
send l a = perform (`send l a)

recv : ∀ {A} → Loc → Process A
recv {A} l = perform (`recv {A} l)
```

Fig. 1. Processes as Algebraic Effects

Figure 1 presents processes as algebraic effects. Signature $\mathbb{P}$ specifies the three operations of processes and their arity:

- `locally performs a local computation of type Term $\mathbb{L}$ A and returns a value of type A.
- `send sends a message of type A to a location and returns a unit value.
- `recv receives a message from a location and returns a value of type $A$. Here, the performer of the `recv needs to specify what type of value it is expected to receive.

We also define Process as a shorthand for terms using operations from $\mathbb{P}$. Finally, we define the helper functions send, receive, and locally.

3.2 Choreographies

```
At : Setω
At = ∀ {ℓ} → Set ℓ → Loc → Set ℓ

focus : Loc → At
focus l A s with l =? s
... | yes _  = A
... | no _   = ⊤

module _ (_@_ : At) where

  data Op : Set₁ where
    `comm : ∀ {A} (s r : Loc) →
            (Term 𝕃 A) @ s → Op

  Arity : Op → Set _
  Arity (`comm {A} _ r _) = A @ r

  ℂ : Sig
  ℂ = Op ◃ Arity
  -- the module ends here
```

```
Choreo : (At → Set) → Setω
Choreo F =
  ∀ {_@_ : At}
  {{_ : ∀ {ℓ} {l} → RawMonad {ℓ} (_@ l)}} →
  Term (ℂ _@_) (F _@_)

_▷_ : ∀ {_@_ : At} {A} →
      (s : Loc) → (Term 𝕃 A) @ s →
      Term (ℂ _@_) (A @ s)
s ▷ t = perform (`comm s s t)

_⟹_◇_ : ∀ {_@_ : At} {A} →
        (s r : Loc) → (Term 𝕃 A) @ s →
        Term (ℂ _@_) (A @ r)
s ⟹ r ◇ t = perform (`comm s r t)
```

Fig. 2. Choreographies as Algebraic Effects

Figure 2 presents choreographies as algebraic effects. One issue that every library-level CP language needs to deal with is how to represent located values. Located values are variables in a choreography that denote values at different locations. We give them types $A \mathbin{@} l$, which intuitively means a value of type A at location l.

Existing library-level CP languages such as HasChor [Shen et al. 2023] define located values as a union of a plain value and a unit value — an option type — and have the unspoken invariant that when projecting to location l, values at l are a plain value, and otherwise, a unit value. Internally, HasChor uses an unwrap function to extract the plain value from the union. The unwrap function is non-total because the union could be a unit value, but HasChor meticulously use it only in situations where the union is a plain value (implicitly use the invariant), so the non-totality never shows up. This approach does not work in Agda, as it demands that every function be total. For this reason, we take an alternative approach to located values, in which they are kept abstract and erased before projection in a way that respects the invariant by construction. We first define At, a type-level function that captures the interface of located values. Then, we define focus, a particular At that we will use in endpoint projection (we will show another At in the next section). Intuitively, focus l erases a located value of type $A \mathbin{@} s$ to A if l is equal to s; otherwise, to a unit value.

The signature $\mathbb{C}$ specifies the two main operations of choreographies using one overloaded constructor:

- `comm s s t denotes a local computation t at location s.
- `comm s r t denotes location l sends the result of a computation t to location r.

We also define Choreo as a shorthand for terms using operations from $\mathbb{C}$ abstracted over a particular At. We also require At to be an instance of monads for any location l, which allows us to chain together located values. Our focus is a monad because the identity functor and units are both monads. We also define two helpful functions, _▷_ and _⟹_◇_, for writing choreographies.

3.3 Endpoint Projection

```
epp : ∀ {F} → Choreo F → (l : Loc) → Process (F (focus l))
epp c l = interp alg return c
  where
    alg : ∀ {A} → ℂ (focus l) -Alg[ Process A ]
    alg (‘comm s r a , k) with l =? s | l =? r
    ... | yes _ | yes _ = locally a »= k
    ... | yes _ | no _  = locally a »= (λ x → send r x) » k tt
    ... | no _  | yes _ = recv s »= k
    ... | no _  | no _  = k tt
```

Fig. 3. Endpoint Projection

We can now define endpoint projection, the process of turning a choreography into a process for a target location. Figure 3 presents our implementation of EPP. The function epp takes a choreography c and a target location l, and uses the effect handler interp to interpret operations in c. For variables, we return them in the generated process. For operations, we construct a $\mathbb{C}$-Algebra (with all located values erased from l's perspective) alg on processes, which does one step of interpretation. The only operation we need to interpret is ‘comm, depending on whether l is equal to s and r:

- If l equals s and r, meaning s and r are the same, we interpret this operation as a local computation followed by the continuation.
- If l equals s but not r, meaning the target location is the sender, we interpret the operation as a local computation followed by a send and the continuation.
- If l equals r but not s, meaning the target location is the receiver, we interpret the operation as a receive followed by the continuation.
- If l equals neither s nor r, meaning the target location is not involved, we just return the continuation.

4 NEXT STEPS

Our next step will be to leverage the algebraic-effects-based formulation of CP presented in the last two sections to prove the correctness of endpoint projection, as well as follow-on properties such as deadlock freedom.

At a high level, given a choreographic semantics $\_\Longrightarrow^{c}\_$ and a network semantics $\_\Longrightarrow^{n}\_$, *soundness* of endpoint projection would say that the projected network *preserves* the semantics of the original choreography. That is, for choreographies c and c', $\mathsf{epp}\ c \Longrightarrow^{n} \mathsf{epp}\ c' \longrightarrow c \Longrightarrow^{c} c'$. *Completeness* of EPP, on the other hand, would say that the network *reflects* the semantics of the original choreography, that is, given choreographies c and c', $c \Longrightarrow^{c} c' \longrightarrow \mathsf{epp}\ c \Longrightarrow^{n} \mathsf{epp}\ c'$. If these correctness conditions hold, the transition systems $\_\Longrightarrow^{c}\_$ and $\_\Longrightarrow^{n}\_$ are in bisimulation, and if choreographies enjoy a progress property, networks also have it, implying that they are deadlock-free. However, the above definitions of soundness and completeness may be too strong — for instance, they prohibit EPP from doing rewritings that change the behaviors of the network but compute the same result. Thus, we are working on more relaxed correctness conditions that permit more interesting behaviors in networks while still maintaining deadlock freedom.

In the longer term, we want to bring our algebraic-effects-based formulation of CP to languages with efficient native support for algebraic effects, such as OCaml and Koka.

REFERENCES

Andrej Bauer. 2019. What is algebraic about algebraic effects and handlers? arXiv:1807.05923 [cs.LO]

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022: 19th International Colloquium, Tbilisi, Georgia, September 27–29, 2022, Proceedings* (Tbilisi, Georgia). Springer-Verlag, Berlin, Heidelberg, 212–237. https://doi.org/10.1007/978-3-031-17715-6_15

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (Jan. 2024), 59 pages. https://doi.org/10.1145/3632398

Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. 6, POPL, Article 23 (Jan. 2022), 27 pages. https://doi.org/10.1145/3498684

Shun Kashiwa, Gan Shen, Soroush Zare, and Lindsey Kuper. 2023. Portable, Efficient, and Practical Library-Level Choreographic Programming. arXiv:2311.11472 [cs.PL]

Donnacha Oisín Kidney, Zhixuan Yang, and Nicolas Wu. 2024. Algebraic Effects Meet Hoare Logic in Cubical Agda. *Proc. ACM Program. Lang.* 8, POPL, Article 56 (jan 2024), 33 pages. https://doi.org/10.1145/3632898

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph. D. Dissertation. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf

Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press. https://doi.org/10.1017/9781108981491

Gordon Plotkin and John Power. 2003. Algebraic operations and generic effects. *Applied categorical structures* 11 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). https://doi.org/10.2168/lmcs-9(4:23)2013

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). 7, ICFP, Article 207 (Aug. 2023), 25 pages. https://doi.org/10.1145/3607849

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

KC Sivaramakrishnan, Daan Leijen, Matija Pretnar, and Tom Schrijvers. 2018. Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172). *Dagstuhl Reports* 8, 4 (2018), 104–125. https://doi.org/10.4230/DagRep.8.4.104