Handling Bidirectional Control Flow

YIZHOU ZHANG, University of Waterloo, Canada
GUIDO SALVANESCHI, University of St. Gallen, Switzerland
ANDREW C. MYERS, Cornell University, USA

Pressed by the difficulty of writing asynchronous, event-driven code, mainstream languages have recently been building in support for a variety of advanced control-flow features. Meanwhile, experimental language designs have suggested effect handlers as a unifying solution to programmer-defined control effects, subsuming exceptions, generators, and async–await. However, despite these trends, complex control flow—in particular, control flow that exhibits a bidirectional pattern—remains challenging to manage.

We introduce *bidirectional algebraic effects*, a new programming abstraction that supports bidirectional control transfer in a more natural way. Handlers of bidirectional effects can raise further effects to transfer control back to the site where the initiating effect was raised, and can use themselves to handle their own effects. We present applications of this expressive power, which falls out naturally as we push toward the unification of effectful programming with object-oriented programming. We pin down the mechanism and the unification formally using a core language that makes generalizations to effect operations and effect handlers.

The usual propagation semantics of control effects such as exceptions conflicts with modular reasoning in the presence of effect polymorphism—it breaks parametricity. Bidirectionality exacerbates the problem. Hence, we set out to show the core language, which builds on the existing tunneling semantics for algebraic effects, is not only type-safe (no effects go unhandled), but also abstraction-safe (no effects are *accidentally* handled). We devise a step-indexed logical-relations model, and construct its parametricity and soundness proofs. These core results are fully mechanized in Coq. While a full-featured compiler is left to future work, experiments show that as a first-class language feature, bidirectional handlers can be implemented efficiently.

CCS Concepts: • **Software and its engineering** → **Control structures**; **Semantics**.

Additional Key Words and Phrases: Effect handlers, type systems, promises, iterators, exceptions, parametricity

ACM Reference Format:
Yizhou Zhang, Guido Salvaneschi, and Andrew C. Myers. 2020. Handling Bidirectional Control Flow. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 139 (November 2020), 30 pages. https://doi.org/10.1145/3428207

1 INTRODUCTION

Modern software places new demands on programming languages. In particular, the need to interact with high-latency external entities—users, file systems, databases, and geodistributed systems—has led software to become increasingly event-driven. Callback functions are a conventional pattern for event-driven programming, but unconstrained callbacks become complex and hard to reason about as applications grow. Hence, it is currently in vogue for programming languages to build in support for advanced control-flow transfer features like generators and async–await. These features support more structured programming of asynchronous, event-driven code.

Authors' addresses: Yizhou Zhang, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada, yizhou@uwaterloo.ca; Guido Salvaneschi, University of St. Gallen, Rosenbergstrasse 51, 9000 St. Gallen, Switzerland, guido.salvaneschi@unisg.ch; Andrew C. Myers, Department of Computer Science, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, andru@cs.cornell.edu.

2475-1421/2020/11-ART139
https://doi.org/10.1145/3428207

Meanwhile, algebraic effects [Bauer and Pretnar 2015; Plotkin and Power 2003; Plotkin and Pretnar 2013] have emerged as a powerful alternative that allows programmers to define their own control effects. They subsume a wide range of features including exceptions, generators, and async–await. Compared to the monadic approach to effects, algebraic effects compose naturally without requiring awkward monad transformers, and enjoy a nice separation between the syntax (i.e., a set of effect operations) and the semantics (i.e., handling of those operations).

However, even with these advanced language features at hand, programmers today still find certain complex control-flow patterns painful to manage. As we argue, features found in mainstream languages are not expressive enough to capture bidirectional control transfer without losing the desirable guarantee that all effects are handled, and existing language designs of algebraic effects cannot readily express this bidirectionality without falling back to patterns that algebraic effects are intended to help avoid.

To resolve these challenges, we generalize the idea of algebraic effects. With algebraic effects, effectful code initiates control transfer by raising effects that propagate up the dynamic call stack to their handlers. With *bidirectional algebraic effects*, effect handlers can raise subsequent effects that propagate in the opposite direction, to the site where the initiating effect was raised. This bidirectionality makes it easy to transmit information and control, to and fro, between program fragments. Accordingly, the type system requires the invocation site of an effect operation to handle not only the initiating effect, but also the reverse-direction effects. **All effects are guaranteed to be handled.**

The usual propagation semantics of control effects is known to interfere with abstraction boundaries in the presence of effect polymorphism, because higher-order functions can intercept effects they are not supposed to handle [Biernacki et al. 2018; Convent et al. 2020; Zhang and Myers 2019; Zhang et al. 2016]. A possible concern might be that bidirectional propagation would further muddle the problem, leading to effect-polymorphic abstractions being violated in previously unidentified ways. To address this concern, we provide bidirectional algebraic effects with a semantics that respects abstraction boundaries, and we rigorously substantiate this strong abstraction claim. **All effects are guaranteed not to be accidentally handled.**

Bidirectionality and the safety guarantees fall out naturally when a language designer views algebraic effects through an object-oriented lens. In fact, the enabling and most visible language change is a generalization of effect operations to make them appear like methods: the notion of an effect operation is extended to allow it to declare further effects its handling code may raise—just as methods in Java can declare exceptions their implementations may throw. Accordingly, handlers of bidirectional effects, which we call *bidirectional handlers*, are generalized to make them appear like objects. In particular, a *self* handler, analogous to the *self* reference found in object-oriented languages, is brought into the context of a handler definition. Self-reference makes a bidirectional effect handler a fixpoint definition—it can ask that its own effects be handled by itself.

The complexity of bidirectional control flow is innate to many modern software applications; bidirectional algebraic effects do not simply make this complexity disappear. Instead, the static guarantees afforded by the type system enable programmers to reason compositionally about bidirectional control flow and therefore to manage complex control flow more easily.

Contributions. The dynamic behavior of bidirectionality is attainable in many languages in various ways—this paper does not aim to rediscover bidirectional control flow. Rather, the contributions consist in (i) a language-design recipe that allows integrating bidirectional control effects in a sound, unified, and efficient way, addressing a variety of programming challenges, and in (ii) formal developments that capture the essence of the mechanism and that establish strong guarantees about it, putting the mechanism on a sound theoretical footing. We proceed as follows:

- Section 2 examines some control-flow features in mainstream languages and reviews algebraic effects, identifying opportunities to improve support for bidirectional control transfer.
- Section 3 demonstrates the new programming abstraction informally (in the setting of a typical object-oriented language) using its various applications, interspersed with discussions on design issues.
- Section 4 shows that, importantly, an abstraction-safe mechanism for bidirectional effects allows programmers to reason compositionally about correctness.
- Section 5 defines a core language, Olaf, capturing the informally introduced features and unification from previous sections. It gives an operational semantics and a static semantics.
- Section 6 continues the formal developments with a logical-relations model for Olaf, culminating in proofs of coveted properties including type safety and parametricity. These formal results are fully mechanized using the Coq proof assistant.
- Section 7 discusses compilation issues. Experimental results on hand-translated examples argue for supporting bidirectional handlers as a first-class language feature.
- Section 8 discusses related work in more detail, and Section 9 concludes.

2 BACKGROUND: ASYNC–AWAIT, GENERATORS, AND ALGEBRAIC EFFECTS

Complex, asynchronous, bidirectional control flow is already a reality for programmers today. This section identifies real-world programming challenges involving bidirectional control flow and shows how existing mechanisms fall short in addressing them.

Async–Await with Promises. An array of languages—for example, C# [Bierman et al. 2012], JavaScript [ECMA International 2018], Rust [Rust language team 2018], and Swift [Lattner and Groff 2019]—have recently added, or are planning to add, support for async–await and the accompanying *promises* abstraction [Liskov and Shrira 1988], also known as *futures* or *tasks*.

As an example, consider the C# program in Figure 1. Method `HttpGetJson` (lines 2–6) sends an HTTP GET request to retrieve a web page by asynchronously running `HttpGet` (line 1), and converts the raw bytes into JSON format. Because `HttpGetJson` is declared `async`, calling it (line 9) does not block computations that do not depend on the result of the request (line 10). The programmer *awaits* the task when they need the result to be ready. Sending HTTP GET requests may raise exceptions (e.g., due to connection issues); the reasonable point for such an exception to emerge is where the tasks are awaited (lines 4 and 11). While `await` sends a signal to a task scheduler on the .NET runtime stack, the exceptions appear to propagate in the opposite direction, from the .NET runtime to the `await` sites.

```
 1 static byte[] HttpGet(String url);
 2 static async Task<Json> HttpGetJson(String url) {
 3   Task<Json> t = Task.Run(() => HttpGet(url));
 4   byte[] bytes = await t;
 5   return JsonParse(bytes);
 6 }
 7 static async Task Main() {
 8   string url = "xyz.org"
 9   Task<Json> t = HttpGetJson(url);
10   ... // do things that do not depend on the query result
11   Json json = await t; // block execution until query terminates
12   ...
13 }
```

Figure 1. Using async–await in C#

However, existing languages that support async–await do not enforce at compile time that exceptions raised by asynchronous computations are handled. The lack of this static assurance makes asynchronous programming error-prone. For example, the C# compiler accepts the program above without requiring that an exception handler be provided—if the asynchronous query does result in an exception, the program crashes. The situation is worse in JavaScript: an exception raised asynchronously is silently swallowed if not otherwise caught. Such unhandled exceptions have been identified as a common vulnerability in JavaScript programs [Alimadadi et al. 2018].

```
effect Yield[X] {
  def yield(X) : void
}
```

(a) Effect signature `Yield`

```
try { node.iter() }
with yield(x) {
  print(x)
  resume()
}
```

(c) Client code handles `Yield`

```
 1 class Node[X] {
 2   var head : X
 3   var tail : Node[X]
 4   …
 5   def iter() : void raises Yield[X] {
 6     yield(head)
 7     if (tail != null)
 8       tail.iter()
 9   }
10 }
```

(b) Iterator raises `Yield`

Figure 2. Yielding iterators via algebraic effects

It is no surprise that C# and JavaScript do not check asynchronously raised exceptions, since neither statically checks exceptions in the first place, unlike Java [Gosling et al. 2018]—which is unfortunate because in practice, most exceptions arising from C# code are undocumented [Cabral and Marques 2007]. However, since asynchronously raised exceptions do not propagate through the regular exception mechanism, it is not clear how to do this checking even in languages like Java that do have checked exceptions.

Generators. Coroutine-style iterators, usually called generators, are a convenient construct available in many languages [ECMA International 2018; Griswold et al. 1981; Hejlsberg et al. 2003; Liskov et al. 1977; Murer et al. 1996; Thomas et al. 2004; van Rossum 2003]. They help avoid the verbose, error-prone pattern of maintaining complex state machines inside *iterator objects* as seen in Alphard [Shaw et al. 1977] and more recently in Java.

However, a weakness of generators is that they do not allow clients to concurrently modify the underlying collections or streams being iterated over. A client iterating over a priority queue might want to change the priority of a received element; similarly, a client iterating over a stream of database records might want to remove one of those records from the database. Generators in the mentioned languages lack the expressiveness to solve these programming challenges. In such languages, the programmer either resorts to implementing iterators as even more complex state machines, or simply shies away from defining powerful, reusable iterator abstractions.

Algebraic Effects. Algebraic effects [Plotkin and Power 2003; Plotkin and Pretnar 2013] are a powerful unifying language feature that can express exceptions, generators, async–await, and other related control-flow mechanisms including coroutines and delimited control [Bauer and Pretnar 2015; Bračevac et al. 2018; Dolan et al. 2017; Forster et al. 2017; Kammar et al. 2013; Leijen 2017b]. The hallmark of algebraic effects is adding support for *signatures* for control effects and for *handlers* as implementations of these signatures.

An effect signature defines one or more *operations*. For example, the signature in Figure 2a, named `Yield` and parameterized by a type variable `X`, contains exactly one operation, `yield`. The operation takes as argument a value of type `X`.

Lines 5–9 of Figure 2b uses `Yield` to define a coroutine-style iterator for nodes in a linked list: it recursively iterates over the tail after yielding the head. Invoking an effect operation raises the corresponding effect: because `iter` invokes `yield` (line 6), calling `iter` can raise the `Yield` effect. Static checking of effects requires this effect be part of the method's type, in its `raises` clause (line 5).

The client program in Figure 2c traverses a chain of nodes by calling `iter` and handling its `Yield` effect. Effectful computations are enclosed by `try`…`with`, followed by a handler that implements the effect operations. Each time that the effect `yield` is raised, the recursive iterator computation

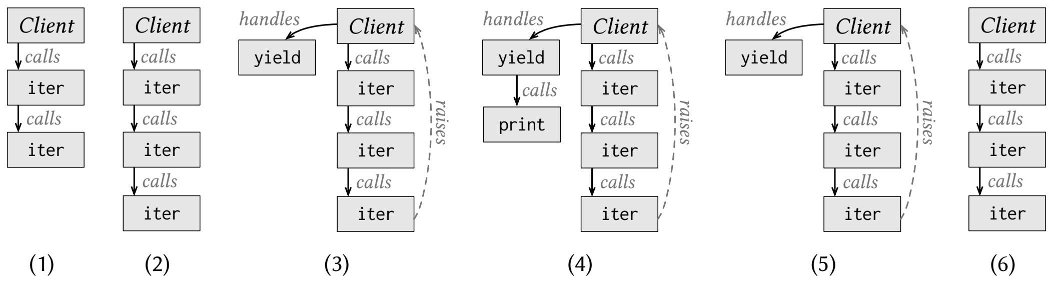


Figure 3. Stack snapshots of the program in Figure 2.

in Figure 2b is suspended and control transferred to the handler in Figure 2c, which prints the yielded element. Control then resumes in the iterator. The resulting execution is similar to using a generator in C#, Python, or Ruby. The sequence of stack frames that result is shown in Figure 3.

Handlers can resume computations suspended by the raising of effects, by calling the resumption denoted by the special `resume` function. This `resume` function is essentially a delimited continuation [Felleisen 1988]. It takes as input the result of the effect operation. The call to `resume` in Figure 2c takes no argument because the result type of `yield` is `void`.

Figure 3 visualizes the control flow in one iteration using stack diagrams, with each diagram capturing the stack at a single point in time:

(1) The iterator has finished processing the first two elements of the list (hence the two `iter` frames).
(2) A third `iter` frame is created; the iterator begins to process the third element.
(3) The iterator raises a `Yield` effect. The effect is then caught by the client's handler.
(4) The client prints the yielded element.
(5) Printing finishes.
(6) Handling of `Yield` finishes. Control is returned to the iterator.

The handler in Figure 2c abbreviates the full signature of the effect operation. The expanded form is shown below. We will write handlers mostly in the abbreviated syntax.

```
try { node.iter() }
with yield(x : X) : void { … }
```

In many practical uses of algebraic effects, as in the example above, invoking `resume` is the last action performed by an effect handler. We call this a *tail resumption* and call such handlers *tail-resumptive*. Not all effect handlers are tail-resumptive; for example, exception handlers are typically *abortive*: they do not resume the computation that raised the effect. Handlers that are either tail-resumptive or abortive can be compiled to efficient code because there is no need to save stack frames once `resume` is invoked or the handler aborts [Leijen 2017c].

Although algebraic effects subsume generators, they do not address the limitations of generators outlined earlier: handler code cannot raise an effect transferring control back to the iterator code to perform a concurrent modification to the data structure. Similarly, algebraic effects can express async–await, but they are awkward when exceptions can be raised asynchronously: a handler running asynchronous computations cannot propagate exceptions raised by those computations back to the `await` site. Beyond generators and async–await, there are other interesting control-flow applications that algebraic effects cannot yet readily support. What is needed is a unified mechanism that can express all these programming challenges easily.

```
interface Yield[X] {
  def yield(X) : void raises Replace[X] | Behead
}
interface Replace[X] { def replace(X) : void }
interface Behead { def behead() : void }
```

(a) Effect signatures

```
17 var node : Node[int] = …
18 try { node.iter() }
19 with yield(x) resume {
20   if (x < 0) behead() else replace(x * 2)
21 }
22 with behead() resume { // handle removal of the first
23   node = node.tail      // element in node
24 }
```

(c) Client code raises interrupts

```
 1 class Node[X] {
 2   var head : X
 3   var tail : Node[X]
 4   …
 5   def iter() : void raises Yield[X] | Behead {
 6     try { yield(head) }
 7     with replace(x) resume { // handle replacement
 8       head = x              // of head
 9     }
10     if (tail != null)
11       try { tail.iter() }
12       with behead() resume { // handle removal of the
13         tail = tail.tail     // first element in tail
14       }
15   }
16 }
```

(b) Iterator handles interrupts

Figure 4. Yielding iterators with reverse-direction interrupts for replacing and removing yielded elements

3 BIDIRECTIONAL ALGEBRAIC EFFECTS, INFORMALLY

We generalize algebraic effects to offer the missing flexibility: handlers of effect operations can themselves raise effects that are handled by callers of the effect operations. Before defining a formal semantics in Section 5, we first introduce the mechanism informally via examples written in a syntax similar to that of Java, Scala or Kotlin, although the ideas could apply to many other languages, especially those with an object-oriented flavor.

3.1 Generators with Concurrent Modification

We want to extend the iterator abstraction of Figure 2 so that iterator clients can issue interrupts to request that the yielded element be replaced or removed. Note that implementing iterators that support such concurrent modifications is awkward in standard OO languages [Liu et al. 2006]. We start by changing the signature of `Yield`, as shown in Figure 4a. Apart from being defined as an `interface` (the reason for which will soon become clear), this signature differs from the one in Figure 2a by declaring that `yield` may itself raise two additional effects, `Replace` and `Behead`, corresponding to the two kinds of concurrent modifications that client code can request. (A `raises` clause may include multiple effects, separated by vertical bars.) Allowing effect operations to declare their own `raises` clauses is a key generalization we make to accommodate bidirectionality.

With the modified `Yield` effect, the client code in Figure 4c is able to remove negative integers from a list and to double the non-negative ones even while iterating over the list: the handler (lines 19–21) passes to the resumption a *computation*, which invokes either operation `behead` or `replace` based on the integer yielded. Notice that `resume` takes as input a computation, rather than a value, as signified by the use of curly braces instead of parentheses.

The resumption accepts a computation whose type and effects must match the result type and effects of the effect operation. For example, in Figure 4c, the resumption to a `yield` call (line 19) accepts a computation that may raise `Replace[int]` and `Behead`.

Meanwhile, the type system guarantees that the resumption—the suspended computation in the iterator—contains handlers for both effects, so that invoking operations `replace` and `behead` can cause control to safely transfer back to the handlers in the iterator. The new iterator code in Figure 4b differs from Figure 2b in adding these two handlers.

To handle `Replace`, the handler (lines 7–9, Figure 4b) updates the head value of the list, and then resumes what is left off by the raising of `replace` in the `yield` handler (line 19, Figure 4c). What is

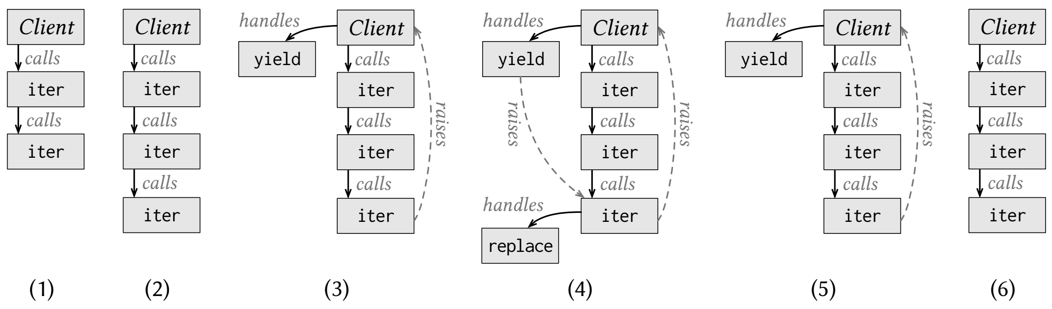


Figure 5. Stack snapshots of the program in Figure 4.

left to be done there is to resume what is left off by the raising of `yield` in the iterator code (line 6, Figure 4b).

The stack diagrams in Figure 5 visualize the control flow:

(1) The iterator has finished processing the first two elements of the list.
(2) A third `iter` frame is created; the iterator begins to process the third element.
(3) The iterator raises a `Yield` effect. The effect is then caught by the client's handler.
(4) The client issues a reverse-direction interrupt to ask that the third element of the list be replaced. This `Replace` effect is caught by the iterator.
(5) Handling of `Replace` finishes. The iterator returns control to the client's `Yield` handler.
(6) Handling of `Yield` finishes. The client returns control to the iterator.

Because effect `Replace` appears to propagate *down* the stack, in the reverse direction of `Yield`, we call these bidirectional effects.

However, it is largely unnecessary to look at stack diagrams to understand control flow. A more meaningful interpretation is to view both `Yield` and `Replace` as propagating outward through evaluation contexts to callers. A `Yield` effect raised by `iter` propagates to the caller of `iter`; in the context of the `iter` method, this caller is represented by `return`, the return address of `iter`. Similarly, a `Replace` effect raised by the `yield` handler propagates to the caller of `yield`; in the context of the `yield` handler, this caller is represented by `resume`, the computation to be resumed after `yield` is handled.

In each of the handlers in Figure 4, `resume` envelops the entire handler computation. In this common case, we allow eliding the curly braces surrounding `resume { … }`. For example, the handler on lines 19–21 is desugarred to the following syntax:

```
try { node.iter() }
with yield(x) { resume { … } }
```

The `Behead` interrupt must be handled differently than `Replace`, because removing a node from a linked list is a nonlocal update—it is most appropriately done at a level that "owns" the current list, that is, either the preceding node in the list (Figure 4b) or the client code (Figure 4c). Notice that in this example, the client code, in addition to the iterator code, must be prepared to handle `Behead`. The client code refers to a linked list by holding a reference to the first node of the list. So when the first node is "beheaded", it is only natural to expect the client code to handle this event specially. If list nodes could be accessed only indirectly (as in Java, through a `LinkedList` object), handling of `Behead` could be hidden from client code.

Instead of handling `Behead` immediately after it surfaces from the call site of `yield` (as we did to `Replace`), `Behead` is propagated to the call site that triggers the iteration of the current list.

Hence, `Behead` occurs in the `raises` clause of `iter` (line 5, Figure 4b), as static checking of effects entails. Accordingly, the type system requires the two call sites of `iter` to deal with `Behead`. Both handle `behead` by replacing the reference to a list with its tail (line 12–14, Figure 4b and line 22–24, Figure 4c).

Although the control flow is complex, reasoning about it remains tractable, especially because the static checking of bidirectional effects offers guidance on how to program the control flow.

An economy of language constructs. As the syntax and the semantics suggest, bidirectionality makes effect signatures and ordinary object interfaces become nearly indistinguishable: both effect operations and object methods can raise effects, and effects always propagate to the caller. This correspondence motivates their unification as a single language construct; throughout the paper, we define effect signatures as interfaces. Moreover, as we introduce later, every bidirectional handler has access to a self handler that it can use to handle effects, analogous to how every object has access to a self (receiver) object on which it can make method calls.

This unification is not merely a syntactic pun. We pin down this unification in the core language (Section 5), which further allows methods and handlers to be defined using the same construct: an ordinary method definition can be viewed as an effect handler where the entire method body is passed to a tail resumption (cf., `return`).

Nevertheless, in the surface language we distinguish them to allow for a familiar programming experience where `return` statements retain their idiomatic meaning in methods—that is, `return` signifies the act of returning to the caller rather than the resumption per se, and method definitions with a `void` return type need not have explicit `return` statements.

Raising effects within handlers. In existing languages, exceptions (and algebraic effects) raised within handlers are propagated to the local context of the handler, rather than to the handler resumption. Bidirectional algebraic effects are compatible with this semantics: so long as the computation raising the effect is not passed to a handler resumption that, per the `raises` clause, can handle the effect, the normal handling behavior is obtained.

Workarounds. One might think that an alternative to bidirectional effects would be to make `yield` return a value of some algebraic data type (ADT) indicating the interrupt event and for the client to pattern-match on the returned ADT value. Note that the `try–with` syntax is an entirely cosmetic choice made to match the Java-like surface language; in fact, algebraic-effects designs for functional languages often use a syntax similar to pattern-matching ADTs: a handler case-analyzes the result of an effectful computation. Consequently, using ADTs as return types of effect operations would not noticeably clarify the code, but it *would* reduce expressive power: control could not be transferred back to the client code after `Replace` or `Behead` were handled. It would also be syntactically heavier-weight: one would have to convert ADT values to algebraic effects. For comparison, Figure 15 shows how the `iter` code would look if `yield` returned an ADT value.

A more general way is to make a `yield` handler resume with a callback value that is the thunked handler computation. But it also means callers of `yield` must voluntarily comply with the contract by remembering to force the thunk—control transfer via callbacks is a pattern algebraic effects are intended to help avoid. Moreover, this approach can be rather inefficient; Section 7 explores the performance implications when it is used to compile bidirectional effects.

3.2 Async–Await with Exceptions

We want to use algebraic effects to express both async–await and asynchronously raised exceptions, while statically ensuring that all exceptions are handled.

```
interface Exn[X] { def exn(X) : void }
interface Async {
  def async[X,Y](Fun2[X,Y]) : Promise[X,Y]
  def await[X,Y](Promise[X,Y]) : X raises Exn[Y]
}
type Fun2[X,Y] = () → X raises Exn[Y] | Async
```

(a) Effect signatures Exn and Async

```
class Promise[X,Y] {
  var state : Sum[List[Awaiter[X,Y]],Fun1[X,Y]]
  Promise() { this.state = inl([]) }
}
type Awaiter[X,Y] = Fun1[X,Y] → void
type Fun1[X,Y] = () → X raises Exn[Y]
```

(b) Definition of the Promise structure

Figure 6. Type-level definitions for expressing exceptional async–await

```
def httpGet(String) : byte[] raises Exn[Http]
def httpGetJson(url : String) : Json raises
    Async | Exn[Http] { // asynchronous method
  val p = async(fun() → httpGet(s))
  val bytes = await(p)
  return jsonParse(bytes)
}

def main() : void raises Async {
  val url = "xyz.org"
  val p = async(fun() → httpGetJson(url))
  … // do things that do not depend on the query result
  try { val json = await(p); … }
  with exn(http) { … }
}
```

(a) User program with effect Async

```
val loop = new EventLoop()
loop.run(main)
```

(b) Running main in an event loop

```
class EventLoop {
  val jobs : Queue[() → void]

  EventLoop() { this.jobs = new Queue() }

  def run(f : () → void raises Async) : void {
    handleAsync(f)
    while (true) {
      try {
        jobs.dequeue().apply() // run next queued job
      } with exn(nse) {
        continue // queue is empty; keep polling it
      }
    }
  }

  def handleAsync(f : () → void raises Async) :
      void { … }                  // Figure 7d

  def exec[X,Y](f2 : Fun2[X,Y], p : Promise[X,Y]) :
      void raises Async { … } // Figure 7e
}
```

(c) Event loop

```
def exec[X,Y](f2 : Fun2[X,Y],
    p : Promise[X,Y]) : void raises Async {
  val f1 : Fun1[X,Y]
  try {
    val x = f2()
    f1 = fun() → x
  } with exn(y) {
    f1 = fun() → exn(y)
  }
  jobs.enqueue(fun() → {
    match (p.state) {
    | inl(awaiters) ⇒
        p.state = inr(f1)
        for (awaiter in awaiters)
          jobs.enqueue(fun() → awaiter(f1))
    | inr(_) ⇒ assert(false) // impossible
    }
  })
}
```

(e) Helper: executes f2; memoizes the result in p

```
def handleAsync(f : () → void raises Async) : void {
  try { f() }
  with async[X,Y](f2 : Fun2[X,Y]) : Promise[X,Y] {
    val p = new Promise[X,Y]()
    __new_thread {
      handleAsync(fun() → exec(f2, p))
    }
    resume { p }
  }
  with await[X,Y](p : Promise[X,Y]) : X raises Exn[Y] {
    match (p.state) {
    | inl(awaiters) ⇒
        awaiters.add(fun(f1) → resume { f1() })
    | inr(f1) ⇒
        resume { f1() }
    }
  }
}
```

(d) Async handler

Figure 7. Using and handling exceptional async–await. Asynchronously raised exceptions are back-propagated to await sites in the user program. Compared with the C# program in Figure 1, the added static checking requires the user program in Figure 7a to handle asynchronously raised exceptions, but otherwise adds no essential syntactic overhead compared to Figure 1. Figures 7b–7e implements the runtime that handles asynchrony.

Exceptions are expressed through the `Exn` effect, defined in Figure 6a. Its operation `exn` takes as input a union of tags, which are instances of singleton classes indicating particular exceptional conditions. For example, we use `Exn[Http]` for exceptions that occurred when processing HTTP requests, `Exn[NSE]` for no-such-element exceptions raised when an empty queue is polled, and `Exn[Http|NSE]` for the union of the two exceptional conditions.

The `Async` effect has two operations, `async` and `await`. Both operations are parameterized by two type variables, one denoting the result type of the asynchronously running computation, and the other the kind of exception it may raise. Operation `async` takes as input a computation and returns a promise: the computation is scheduled to run by an `Async` handler, and when it finishes, its result, which is either a value or an exception, is memoized by the promise. Awaiting the promise either gives back the value or raises the exception.

The `Async` signature is recursive in that operation `async` accepts a computation whose effects can include not only `Exn[Y]` but also `Async`. This type-level recursion is useful because it allows for promises that await other promises, a usage pattern found in many JavaScript and C# programs (including the program in Figure 1).

Using exceptional async–await. The C# program in Figure 1 can be ported to use this `Async` effect, as Figure 7a shows. It has the same run-time behavior, but stronger static checking. Because method `httpGet` may raise `Exn[Http]`, the promises on lines 4 and 10 have types `Promise[byte[],Http]` and `Promise[Json,Http]` respectively, and thus awaiting them may raise `Exn[Http]`. The type system then requires a handler for this asynchronous exception to be provided—all exceptions, asynchronously raised or not, are guaranteed to be handled.

The type-level recursion in `Async` allows invoking operation `async` with a computation that has effect `Async` (line 10), capturing the fact that the resulting promise awaits another one (line 5).

We remark that in Figure 7, only Figure 7a is user-level code, showing that we add no essential syntactic burden compared to Figure 1. The rest of Figure 7 implements the runtime that handles asynchrony, with probably reasonable and excusable complexity.

The promises abstraction. Like JavaScript promises, promises are in one of three possible states, expressed using a `Sum` type in Figure 6b. A promise is either (1) pending completion with a list of awaiters that will run after the promise is complete, (2) is complete with a value, or (3) is complete with an exception. Promises are initialized to pending completion with no awaiters. The two completion states are expressed via a function of type `()` $\to$ `X raises Exn[Y]`, or `Fun1[X,Y]` for short. The awaiters are resumptions to calls to `await`; they are higher-order functions taking as input a function of type `Fun1[X,Y]`. Whereas the `Async` handler in the language runtime can create and inspect promises directly, user programs are supposed to introduce and eliminate promises only indirectly via operations `async` and `await`.

Handling exceptional async–await. Typically a scheduler for asynchronous computations exists in the language's runtime, as is the case with JavaScript and C# (although an algebraic-effects encoding makes it possible for software components to handle their own `Async` effects). We present in Figures 7c–7e a possible implementation of the runtime in a style similar to those of JavaScript (e.g., Node.js [Node]), which maintains a queue of jobs run by an *event loop*. Asynchronous computations of the `main` program is run in this event loop, as Figure 7b suggests.

Initially, the queue is empty (line 19), and the `main` program is run inside an `Async` handler (line 22) that handles all requests to start asynchronous computations and to await their results. New jobs are enqueued on completion of asynchronous computations (lines 65–73). The queued jobs are then run in the event loop (line 25). For simplicity, we use FIFO scheduling.

Figure 7d defines the `Async` handler. To handle `async`, the handler creates a new promise, creates a thread using a `__new_thread` intrinsic, and returns the promise (lines 40–46). The new thread executes the computation `f2` asynchronously by calling a helper function `exec`, defined in Figure 7e. It stores the result of `f2` into a function `f1` that represents a *control-stuck* computation—invoking `f1` either immediately returns a value or immediately raises an exception (lines 59–64). Lines 65–73 then schedule the events that should happen after the asynchronous computation's result is ready: they include transitioning the promise into one of the two completion states (line 68), and scheduling all code awaiting the promise to run (lines 69–70). In the case that `f1` is exceptional, invoking an awaiter with `f1` (line 70) effectively causes control to transfer to the exception handler in the user program (line 13). While `exec` handles `Exn[Y]` for `f2`, it does not handle its `Async` effect. So the call to `exec` is enclosed in the very `Async` handler being defined (line 43).

How to handle `await` depends on the promise's state. To handle `await` for a promise that is still pending completion (lines 49–50), the resumption to the `await` call is added to the awaiter list of the promise. Otherwise (lines 51–52), the promise must be complete, and the resumption is invoked with the result memoized by the promise.

Prior encodings. The ability to encode promise-based async–await [Dolan et al. 2017; Leijen 2017a,b] speaks to the expressive power of algebraic effects,[1] but encodings in existing language designs compromise on how they accommodate exceptional computations. Koka [Leijen 2017a,b] supports structured asynchrony via algebraic effects, but uses an `either` monad for possible exceptional outcomes of the `await` operation—but encoding exceptional outcomes into monadic values is a pattern that algebraic effects in Koka are intended to help avoid! Unlike Koka, Multicore OCaml [Dolan et al. 2017] does not check algebraic effects statically. To notify user programs about asynchronously raised exceptions, the language adds a special `discontinue` construct. It is our goal to treat async–await and asynchronous exceptions in a more unified way: both are statically checked algebraic effects.

3.3 Communication Protocols

It has been shown before that algebraic effects can express interprocess communication, but not without using a more exotic form of effect handlers that deviates from the original categorical interpretation of effect handlers by Plotkin and Pretnar [2013]. In particular, prior work relies on *shallow handlers* to keep the encoding syntactically light [Kammar et al. 2013; Lindley et al. 2017].

Bidirectional algebraic effects offer an alternative: since we allow all effect signatures to declare further effects, a series of raised effects can, in general, bounce back and forth an arbitrary number of times, turning effect signatures into statically checked communication protocols. We demonstrate this capability using the effect signatures `Ping` and `Pong` (Figure 8a) to obtain a pair of functions that send messages to (i.e., raise effects at) each other in lockstep.

Effects `Ping` and `Pong` are mutually recursive. While invoking `ping` (resp. `pong`) appears to one process as sending a message, it appears to the other process as receiving the message, as that other process must handle `ping` (resp. `pong`). Because `ping` (resp. `pong`) declares it may raise `Pong` (resp. `Ping`), a process typed with effect `Ping` (resp. `Pong`) should be prepared to receive a `Pong` (resp. `Ping`) message after sending a `Ping` (resp. `Pong`). The same process is allowed to do more `ping`s (resp. `pong`s) on receiving the `Pong` (resp. `Ping`). The operations in this example do not carry a payload; a more involved example can be found in Section 4.

[1]In JavaScript and C#, `async` functions implicitly wrap their return values in promises (e.g., line 5 in Figure 1). An algebraic-effects encoding does not automatically support this behavior, but does not preclude it either. This eager wrapping possibly encourages the anti-pattern of unnecessary promises [Alimadadi et al. 2018; Okur et al. 2014].

```
interface Ping {
  def ping() : void raises Pong
}
interface Pong {
  def pong() : void raises Ping
}
```

(a) Effect signatures

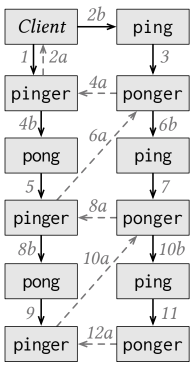
```
15 try {
16   pinger(0, 50)
17 } with ping() resume {
18   ponger()
19 }
```

(c) Client code

```
 1 def pinger(i,N : int) : void raises Ping {
 2   try {
 3     if (i < N)
 4       ping()
 5   } with pong() resume {
 6     pinger(i+1, N)
 7   }
 8 }

 9 def ponger() : void raises Pong {
10   try { pong() }
11   with ping() resume {
12     ponger()
13   }
14 }
```

(b) Processes with effects `Ping` and `Pong`



(d) Stack snapshot

Figure 8. Processes `pinger` and `ponger` send messages to each other in lockstep.

Figure 8b shows two methods, `pinger` and `ponger`, typed with effects `Ping` and `Pong`, respectively. They are glued together by the client code in Figure 8c. At each step, `pinger` does a `ping` (line 4), and `ponger` reacts to it by doing a `pong` (line 10), upon receiving which, `pinger` recursively calls itself (line 6). This interaction happens 50 times, after which the communication ceases. Figure 8d visualizes the control flow. As in Figures 3 and 5, dashed arrows signal raising an effect to a stack frame where its handler is found, while solid arrows signal handler invocations and ordinary method calls.

Effects `Ping` and `Pong` can be viewed as specifying a communication protocol, where the effect signatures choreograph the sending and receiving of messages. Processes statically typed with these effects conform to the protocol dynamically. In this sense, bidirectional algebraic effects offer an expressive behavioral-typing discipline resembling session types [Honda et al. 1999].

3.3.1 Deep versus Shallow. Previous work raises the distinction between *deep handlers* and *shallow handlers* [Kammar et al. 2013]. They differ in the construction of handler resumptions: the resumption to a deep handler contains the very handler at its outermost layer, so subsequent effects raised in the resumption can be handled by the same handler. Handlers of algebraic effects, as originally introduced by Plotkin and Pretnar [2013], are deep: an effect handler is a fold (in category-theoretic terms, a catamorphism [Meijer et al. 1991]) over the algebra of effect operations. It has been argued that deep handlers behave more regularly and admit easier reasoning [Kammar et al. 2013; Lindley et al. 2017]. However, interprocess communication has been identified as the quintessential example where shallow handlers lead to an easier encoding [Hillerström and Lindley 2018; Kammar et al. 2013]. By contrast, our encoding does not rely on shallow handlers, and has the added benefit of capturing the sequencing of effects in the signatures. Nonetheless, we do not claim to settle the debate over deep vs. shallow; it is not clear that the two encodings faithfully reflect each other, and other applications of shallow handlers might emerge in the future.

3.3.2 Deep, Bidirectional Handlers Are Recursive Definitions. The deep-handling semantics hints at recursion. However, before bidirectional handlers, handlers have been unable to exploit this recursion to specify that they should handle their own effects, because handler resumptions can only accept a value whose type matches the operation's result type. By contrast, bidirectional handlers allow passing to `resume` a computation that, in addition to the effects in the `raises` clause, has the effect currently being handled.

We can use this feature to simplify the code of Figures 8b and 8c for a client that is privy to how `ponger` is implemented. Notice there are two identical handlers for `Ping`—one on lines 11–13 in `ponger`, and the other on lines 17–19 in the client—meaning all `Ping` effects are handled identically. So we should be able to obtain the same bidirectional communication by keeping the `Ping` handler in the client code but doing away with the one in `ponger`. The new `ponger` looks as follows:

```
def ponger() : void raises Pong | Ping { pong() }
```

It has an additional `Ping` effect because we got rid of the handler. Despite this change, the client code in Figure 8c continues to type-check. The additional `Ping` effect raised by calling `ponger` (line 18) is handled by the very handler being defined.

This recursion in handling makes deep, bidirectional handlers effectively fixpoint definitions. While the binding structure of this fixpoint remains rather vague at this point, Section 4.3.2 makes it precise.

4 RETAINING PARAMETRICITY

The typical semantics for handling algebraic effects is to search for the dynamically closest enclosing handler with a matching effect signature; in fact, this semantics works for the examples discussed so far. However, this semantics is known to be in conflict with modular reasoning: higher-order, effect-polymorphic abstractions can accidentally handle effects they are not designed to handle, breaking abstraction boundaries [Biernacki et al. 2018; Zhang and Myers 2019; Zhang et al. 2016]. For example, the higher-order function `map` is declared to be polymorphic over an effect variable α that ranges over the effects its argument function `f` may raise. The intended run-time behavior is for these effects to be propagated to the caller of `map`.

```
def map[X,Y,α](l : List[X], f : X → Y raises α) : List[Y] raises α
```

However, a problematic semantics could lead to `map` accidentally handling `f`'s effects if the implementation of `map` happens to contain an effect handler with a matching signature.

Previous work gives an alternative tunneling semantics to algebraic effects, addressing the accidental handling problem without giving up the appeal of algebraic effects [Zhang and Myers 2019]. In this section, we show that bidirectional algebraic effects create new possibilities of accidental handling, but that abstraction safety can be retained by adapting the tunneling semantics.

4.1 The Problem of Accidental Handling

We call the typical algebraic-effects semantics the *signature-based semantics* because it identifies a propagating effect by its signature. We show that if a signature-based semantics were used, resumptions to bidirectional handlers could handle effects by accident.

Suppose we want to use the ping-pong protocol to define two processes where one process fetches webpages for HTTP URLs it receives from the other process. Both processes may make asynchronous queries. To this end, effect signatures `Ping` and `Pong` are modified (Figure 9a) to carry a payload and to be parameterized by an effect variable ranging over the extra effects the processes may have. A pair of processes with these two effects are composed using method `pingpong` defined in Figure 9b. It is effect-polymorphic, allowing the processes to have additional effects.

Whereas major platforms supporting async–await dispatch asynchronous jobs using a single dispatcher in the runtime, the ability to treat async–await as a regular algebraic effect enables software components to handle their own `Async` effects. This ability is useful, for example, when a special event loop is wanted, or when `Async` effects must be monitored [Leijen 2017a].

The processes are defined in Figure 9c. Process `pinger` reads URLs asynchronously from an input source (line 10) and handles its own `Async` effect (lines 15–16). Process `ponger` issues asynchronous

```
interface Ping[α] {
  def ping(String) : void raises Pong[α] | α
}

interface Pong[α] {
  def pong(String) : void raises Ping[α] | α
}
```

(a) Effect signatures

```
1 def pingpong[α](
2     f1 : () → void raises Ping[α] | α,
3     f2 : String → void raises Pong[α] | α) :
4     void raises α {
5   try { f1() }
6   with ping(s) resume { f2(s) }
7 }
```

(b) Playing ping-pong

```
 8 def pinger[α]() : void raises Ping[α] | α {
 9   try {
10     val url = await(async(read))
11     ping(url)
12   } with pong(data) {
13     write(data)
14     resume { pinger() }
15   } with async(f) { …    // This Async handler is intended ONLY for
16   } with await(p) { …    // the async and await calls on line 10
17   } with exn(io) { … }
18 }

19 def ponger[α](url : String) : void raises Async | Pong[_] | α {
20   try {
21     val json = await(async(fun() → httpGetJson(url)))
22     pong(json["data"])
23   } with ping(url) resume {
24     ponger(url)
25   } with exn(http) { … }
26 }
```

(c) Processes Pinger and Ponger

Figure 9. Program pingpong(pinger, ponger) risks accidental handling under the signature-based semantics

HTTP GET requests (line 21) but chooses to let an outer event loop to handle its Async effects; hence Async appears in its raises clause. Underscores are placeholders for inferred effects. The signature-based semantics would infer, via unification, the placeholder effect in the definition of ponger to be Async | α.

Program pingpong(pinger, ponger) starts the bidirectional communication. The signature-based semantics would instantiate the effect variable α of pingpong to be Async, that of pinger to be Async, and that of ponger to be the empty effect. The program as a whole would have effect Async—the programmer expects an outer event loop to handle ponger's Async effects.

However, under the signature-based semantics, the program would not execute in the expected way. When ponger invokes either operation async or await, the dynamically closest enclosing handler for Async would intercept and handle it: climbing up the call chain, ponger is called by ping (lines 6 and 24), which is invoked on line 11, which is enveloped by the Async handler intended only for the async and await calls on line 10. The programmer is in for a surprise.

This phenomenon of handler resumptions accidentally handling effects is new; key ingredients of the example include bidirectionality and effect signatures parameterized over effects, which are missing in previous work addressing accidental handling [Biernacki et al. 2018; Zhang and Myers 2019]. Still, we can rely on the prior work to help us understand the crux of the problem.

4.2 A Loss of Parametricity

An insight of prior work [Biernacki et al. 2018; Zhang and Myers 2019] is that accidental handling reflects a loss of parametricity. Reynolds' Abstraction Theorem for System F [Reynolds 1983] implies that parametricity of type polymorphism relies on polymorphic functions not being able to make decisions based on the types instantiating the type parameters. Analogously, parametricity of effect polymorphism requires that effect-polymorphic functions not make decisions based on the effects they are instantiated with. The signature-based semantics runs afoul of this requirement. In the example above, `pinger` is a function polymorphic over an abstract effect α. But under the signature-based semantics, it would be able to inspect, at run time, the signatures of propagating effects otherwise statically denoted by α, causing accidental handling.

A loss of parametricity is a loss of modular reasoning. The signature-based semantics means, for example, that one cannot reason modularly about the program context `pingpong(pinger,•)` by just looking at the types and without knowing how `pinger` is implemented. The hole expects a function of a type as shown on line 3 of Figure 9b, which does not speak of `Async`. Yet filling the hole with a function with effect `Async` would lead to surprise.

4.3 Tunneling via Lexically Scoped Handlers

To restore parametricity, we adapt the idea of tunneled algebraic effects [Zhang and Myers 2019]. Tunneling echoes the modular reasoning requirement that handlers should only handle effects they are locally "aware" of; otherwise, effects tunnel through handlers. For example, the definition of `pinger` is polymorphic over the effect variable α it binds, so it ought to be "oblivious" to any propagating effects that correspond to α at run time—these effects tunnel through the handler in `pinger`. By contrast, the call site `pingpong(pinger,ponger)` is "aware" that `ponger` may raise `Async`—the call site is thus required to handle this effect.

This modular reasoning requirement suggests that the tunneling semantics choose handlers *lexically*. This lexical scoping of handlers generalizes naturally to bidirectional algebraic effects, with handler bindings brought into the lexical scope in three ways:

(1) a `try–with` statement binds an identifier, corresponding to the handler following `with`, for use in the `try`-block computation,
(2) a `raises` clause binds a set of identifiers, each corresponding to an effect signature in the clause, for use in the method body or handler body, and
(3) a handler definition binds an identifier named `self`, corresponding to the current handler, for use in the handler body.

The first two ways are a straightforward adaptation of the tunneling semantics. The third way explains why a bidirectional handler can demand its own effects to be handled by itself (Section 3.3.2).

The approach of Zhang and Myers [2019] is that programs written in the usual syntax are desugared to give handler bindings explicit names. Handlers for effectful computations are then chosen by resolving an omitted handler to the lexically closest enclosing binding. Because handlers are resolved lexically, effects appear to tunnel to handlers without allowing dynamically enclosing handlers to intercept them, even if they have identical signatures.

As an example, Figure 10 shows the desugaring of `pinger` (Figure 9c). It makes explicit all handler bindings and references to the bindings. Desugaring the `raises Ping[`α`]` clause introduces an identifier H_{Pi} into the `pinger` body; it denotes the handler used to handle `Ping[`α`]` effects raised by `pinger`. Invoking operation `ping` (line 4, in the `try` block) requires a `Ping[`α`]` handler to be provided. This handler is resolved to H_{Pi}, the lexically closest enclosing binding for the signature. The invocation also requires a handler for `Pong[`α`]` to be provided, because operation `ping` is defined to raise `Pong[`α`]`. This handler is resolved to H_{Po}, the binding introduced into the `try` block by one of the surrounding handler definitions. The locally defined `Async` handler, denoted by H_{A}, is used to handle the invocations of `async` and `await` (line 3).

```
def pinger[α][HPi : Ping[α]]() : void raises HPi | α {
  try {
    val url = HA.await[HE](HA.async(read))
    HPi.ping[HPo](url)
  } with HPo : Pong[α] = {
    def pong[HPi : Ping[α]](data : String) : void raises HPi | α {
      write[HE](data)
      resume { pinger[α][HPi]() }
    }
  } with HA : Async = {
    def async[X,Y](f : Fun2[X,Y]) : Promise[X,Y] { … }
    def await[X,Y][HE : Exn[Y]](p : Promise[X,Y]) : X raises HE { … }
  } with HE : Exn[IO] = { … }
}

def ponger[α][HA : Async][HPo : Pong[HA | α]](url : String) : void raises HA | HPo | α { … }
```

```
// Client code
pingpong[HA2](pinger[HA2], ponger[∅][HA2])
```

Figure 10. Desugaring pinger, ponger, and the client program

Figure 10 also shows the desugaring of the client program `pingpong(pinger,ponger)`. Assuming H_{A2} denotes a surrounding `Async` handler, it instantiates the effect variable of `pingpong` to be H_{A2}, that of `pinger` to be H_{A2}, and that of `ponger` to be the empty effect. The client program as a whole has effect H_{A2}, indicating it uses the outer `Async` handler to handle `Async` effects raised by `ponger`.

4.3.1 Lifetimes as Effects. Effect handlers obey a stack discipline. A handler's lifetime begins when the corresponding `try`-block is entered, and ends when the `try`-block computation is done: the handler cannot outlive the lexical structure binding it. The desugaring outlined above makes it explicit that references to handlers are passed as lexically scoped arguments. But lexical scoping alone does not prevent closures that outlive handlers from capturing them, making them dangling references.

To prevent such dangling references, the type system follows prior work [Biernacki et al. 2020; Brachthäuser et al. 2020; Zhang and Myers 2019] in treating handler lifetimes as computational effects, in a similar way to region-based type systems [Grossman et al. 2002; Lucassen and Gifford 1988; Tofte and Talpin 1997]. A computation that uses handlers to handle algebraic effects is typed with the lifetimes of those handlers as its computational effects.

Such lifetime effects, denoted by handler identifiers, are written in `raises` clauses of desugared types. For example, desugaring the `raises Ping[α]` clause not only introduces the handler binding H_{Pi}, but also means the method has lifetime effect H_{Pi}. Because the method body is typed with the lifetime of H_{Pi}, it cannot outlive H_{Pi}; it is thus allowed to use the non-dangling reference H_{Pi} to invoke `ping`.

The desugaring that introduces explicit handler bindings can thus be understood as also assigning default lifetimes automatically as follows: an ordinary object is not lifetime-bounded, since its lifetime is the same as the heap region; the lifetime of a handler is bounded by the `try–with` stack region to which it is attached; and a method with a `raises` clause is lifetime-polymorphic. Assigning default lifetimes is present in Cyclone (*default regions* [Grossman et al. 2002]) and Rust (*lifetime elision* [Klabnik and Nichols 2019]). We do not require the complexity of these full-blown region-based type systems, however, because only handlers are lifetime-bounded and because handler lifetimes are restricted to stack regions.

$$
\begin{array}{llcl}
\textit{programs} & P & ::= & \bar{I}\,;\, \Downarrow^{\mathsf{L}} t \\
\textit{interface definitions} & I & ::= & \textbf{interface}\ \mathbb{F}[\overline{\alpha}]\,\{T\} \\
\textit{operation signatures} & T, S & ::= & \forall\alpha.\,T \mid \forall\zeta.\,T \mid \tau \longrightarrow T \mid [\tau]_c \\
\textit{types} & \tau, \sigma & ::= & \mathbb{1} \mid \boldsymbol{\nu}^{\ell}\ \mathbb{F}[\overline{c}] \mid \boldsymbol{\nu}^{\ell}\ T \mid \textbf{cont}\,[\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2} \\
\textit{composite effects} & c & ::= & \varnothing \mid c,\ e \\
\textit{atomic effects} & e & ::= & \alpha \mid \ell \\
\textit{lifetimes} & \ell & ::= & \zeta \mid \mathsf{L} \\
\textit{operation implementations} & D & ::= & \Lambda\alpha.\,D \mid \Lambda\zeta.\,D \mid \lambda\mathsf{x}.\,D \mid \lambda\mathsf{k}.\,t \\
\textit{values} & v, u & ::= & \mathsf{x} \mid () \mid \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{\nu}^{\mathsf{L}}\ D \mid \boldsymbol{\nu}^{\mathsf{L}}\ D \mid \textbf{cont}\ K \\
\textit{terms} & t, s & ::= & v \mid t.\textbf{op} \mid t\ c \mid t\ \ell \mid t\ s \mid \Downarrow^{\mathsf{L}} t \mid \Uparrow t \mid \textbf{let}\ \mathsf{x} = t\ \textbf{in}\ s \mid \textbf{throw}\ t\ s \\
\textit{evaluation contexts} & K & ::= & [\cdot] \mid K.\textbf{op} \mid K\ c \mid K\ \ell \mid K\ t \mid v\ K \mid \Uparrow K \mid \textbf{let}\ \mathsf{x} = K\ \textbf{in}\ t \mid \textbf{throw}\ K\ t
\end{array}
$$

effect variables α *lifetime variables* ζ *lifetime constants* L *value variables* $\mathsf{x}, \mathsf{k}, \mathsf{H}, \mathsf{self}, \ldots$ *interface names* $\mathbb{F}$

Figure 11. Syntax of Olaf

4.3.2 *Fixpoint Handlers.*

The availability of a `self` handler makes it precise that bidirectional (deep) handlers are fixpoint definitions: the fixpoint is taken of a handler definition quantifying over `self`. Because the resumption to a deep handler is enveloped by the same handler, the computation passed to `resume` cannot outlive the handler and can thus safely use it to handle its effects. The figure above shows how Figure 8c is desugared using `self`, assuming `ponger` has the signature as shown in Section 3.3.2.

```
try { pinger[HPi]() }
with HPi : Ping = {
  def ping[HPo : Pong]() : void raises HPo {
    resume { ponger[HPo][self]() }
  }
}
```

Since being self-referential is also a key characteristic of objects, it seems that objects and handlers are almost unifiable. Section 5 makes this unification precise for a core language.

5 A CORE LANGUAGE

We study the formal foundation of bidirectional algebraic effects using a core language, Olaf, that captures the key aspects of the language mechanism introduced in Sections 3 and 4.

Olaf is both functional and object-oriented. Like a lambda calculus, it does not support imperative state. Like an object-oriented language, it supports the separation between objects and interfaces, and objects in Olaf are self-referential. Olaf supports effect signatures and handlers using the same constructs for object interfaces and objects. The effect-handling construct `try-with` is captured by something that resembles delimited control [Gunter et al. 1995], following prior work [Brachthäuser et al. 2020; Zhang and Myers 2019].

Olaf is intended to capture the essence of the language mechanism. It makes simplifications similar to existing formalisms of algebraic effects with lexically scoped effect handlers [Biernacki et al. 2020; Zhang and Myers 2019]: it is assumed that handlers are always given explicitly for effectful computations (rather than resolving elided handlers to the closest lexically enclosing binding) and that effect signatures contain exactly one effect operation. Lifting these restrictions is straightforward but adds syntactic complexity that obscures the key issues. Because of its recursive interface definitions and fixpoint handlers, Olaf is Turing-complete.

5.1 Syntax

Figure 11 presents the syntax of Olaf. Metavariables standing for identifiers have a lighter color. An overline denotes a (possibly empty) sequence of syntactic objects. For instance, $\overline{e}$ denotes a sequence of effects; an empty sequence is $\varnothing$. Effect sequences, or *composite effects*, are denoted by c.

The type system tracks handler lifetimes as effects. An effect e is either an effect variable α or a lifetime ℓ, which is either a lifetime variable ζ or a lifetime constant L. Lifetime effects compose easily, since effect sequences are essentially sets—the order and multiplicity of effects in a sequence are irrelevant. Substituting an effect sequence $\overline{e}$ for an effect variable α in another effect sequence works by flattening $\overline{e}$ and replacing α with the flattened effects. Substituting a lifetime ℓ for a lifetime variable ζ works in the usual way.

A value v is either a variable x, the unit value (), a handler value **fix** self **is** $\boldsymbol{\nu}^{\mathsf{L}}\, D$, an operation value $\boldsymbol{\nu}^{\mathsf{L}}\, D$, or a continuation **cont** K. Continuations are represented by evaluation contexts K. In a term of form **throw** t s, term t must evaluate to a continuation, after which s is placed in the evaluation context representing the continuation.

While in Section 4.3 lifetimes are identified by handler bindings, in Olaf lifetimes are decoupled from handlers and form a separate syntactic category: handler values (subsuming objects) are of form **fix** self **is** $\boldsymbol{\nu}^{\mathsf{L}}\, D$, consisting of an operation implementation D and the lifetime L of the value. Handler values are fixpoints; the self variable is bound in D. An operation value is of form $\boldsymbol{\nu}^{\mathsf{L}}\, D$; it is the result of unrolling the fixpoint definition of a handler value to extract the operation.

Lifetime constants L are declared by, and bound in, $\Downarrow$-terms. Olaf encodes the `try-with` construct using $\Downarrow$-terms. The computation t guarded by a $\Downarrow^{\mathsf{L}}$ may have lifetime effect L. While $\Downarrow$-terms bind and discharge lifetime effects, $\Uparrow$-terms invoke handlers and thus introduce lifetime effects.

A term has either the unit type $\mathbb{1}$, a handler type (a.k.a. an interface type) $\boldsymbol{\nu}^{\ell}\, \mathbb{F}[\overline{c}]$, an operation type $\boldsymbol{\nu}^{\ell}\, T$, or a continuation type $\textbf{cont}\, [\tau_1]_{\overline{e_1}} \rightsquigarrow [\tau_2]_{\overline{e_2}}$. Handler values have handler types, while operation values have operation types. A term of form $t.\textbf{op}$ extracts the operation value from a handler value, by unrolling the fixpoint handler definition. A continuation of type $\textbf{cont}\, [\tau_1]_{\overline{e_1}} \rightsquigarrow [\tau_2]_{\overline{e_2}}$ can be applied to a computation of type $[\tau_1]_{\overline{e_1}}$.

An operation implementation D is possibly polymorphic over effect variables ($\Lambda\alpha.\, D$), lifetime variables ($\Lambda\zeta.\, D$), and value variables ($\lambda\mathsf{x}.\, D$). Correspondingly, the operation signature T of a handler can be effect-polymorphic ($\forall\alpha.\, T$), lifetime-polymorphic ($\forall\zeta.\, T$), and value-polymorphic ($\tau \rightarrow T$). The last parameter k of an operation implementation is a continuation—to wit, the handler resumption. An operation has a result type $[\tau]_c$; the handler resumption is able to discharge effects in c. Unlike previous work that gives algebraic effects operational meanings [Biernacki et al. 2019; Leijen 2017b; Lindley et al. 2017; Zhang and Myers 2019], handler resumptions in Olaf are evaluation contexts taking as input (possibly effectful) computations, instead of functions that take only pure values.

An Olaf program consists of a set of interface definitions and a "main" term to be evaluated. The interfaces are mutually recursive and can be parameterized by effect variables. The "main" term is guarded by a $\Downarrow$, which binds a lifetime constant. This lifetime constant is used as the lifetime of handler values that correspond to ordinary objects; they exist for the full lifetime of the program and hence need not obey the usual stack discipline imposed on other handlers.

Example. Olaf is less removed from the informal surface language used in Sections 3 and 4 than its syntax might suggest. Below we encode function `pinger` (Figure 10) in Olaf. This example demonstrates how Olaf encodes various language constructs including handler bindings, the `try-with` construct, functions and function calls, and handlers and handler invocations.

The (recursive) function `pinger` is encoded as a (self-referential) object, which is expressed as a handler value that applies its resumption to its body t_{pinger}:

$$\textbf{fix}\ \mathsf{pinger}\ \textbf{is}\ \boldsymbol{v}^{\mathsf{L}_0}\ \Lambda\alpha.\ \Lambda\zeta_{\mathsf{Pi}}.\ \lambda\mathsf{H}_{\mathsf{Pi}}.\ \lambda\mathsf{k}.\ \textbf{throw}\ \mathsf{k}\ t_{\text{pinger}}$$

In the informal language of Section 4, a handler binding denotes both the handler and also its lifetime. In Olaf, a handler binding is modeled with two bindings, one for a lifetime and the other for a value variable: the value variable stands for a handler of the given lifetime. For example, `Pinger[`α`]` in the `raises` clause of `pinger` is modeled in Olaf by a handler binding H_{Pi} and a lifetime binding ζ_{Pi}, where H_{Pi} has lifetime ζ_{Pi}.

Function `pinger`, encoded as an object, need not obey a stack discipline, so it has lifetime L_0, which is assumed to be the lifetime constant declared by the ⬇ guarding the "main" program. Term t_{pinger}, that is, the body of `pinger`, looks as follows, where the bindings α, ζ_{Pi}, H_{Pi}, k, and the self reference pinger, are in scope:

$$\begin{aligned} t_{\text{pinger}} \overset{def}{=} ⬇^{\mathsf{L}}\ &\textbf{let}\ \mathsf{H}_{\mathsf{E}}\ = \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{v}^{\mathsf{L}}\ \ldots\ \textbf{in} \\ &\textbf{let}\ \mathsf{H}_{\mathsf{A}}\ = \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{v}^{\mathsf{L}}\ \ldots\ \textbf{in} \\ &\textbf{let}\ \mathsf{H}_{\mathsf{Po}} = \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{v}^{\mathsf{L}}\ D_{\text{pong}}\ \textbf{in} \\ &\textbf{let}\ \mathsf{url}\ = ⇧\ \mathsf{H}_{\mathsf{A}}.\textbf{op}\ \mathsf{L}\ \mathsf{H}_{\mathsf{E}}\ \mathsf{read}\ \textbf{in} && \text{(line 3)} \\ &⇧\ \mathsf{H}_{\mathsf{Pi}}.\textbf{op}\ \mathsf{L}\ \mathsf{H}_{\mathsf{Po}}\ (\mathsf{url}) && \text{(line 4)} \end{aligned}$$

$$\begin{aligned} D_{\text{pong}} \overset{def}{=} &\Lambda\zeta_{\mathsf{Pi}}.\ \lambda\mathsf{H}_{\mathsf{Pi}}.\ \lambda\mathsf{data}.\ \lambda\mathsf{k}. \\ &\textbf{let}\ \_ = ⇧\ \mathsf{write}.\textbf{op}\ \mathsf{L}\ \mathsf{H}_{\mathsf{E}}\ \mathsf{data}\ \textbf{in} && \text{(line 7)} \\ &\textbf{throw}\ \mathsf{k}\ (⇧\ \mathsf{pinger}.\textbf{op}\ \alpha\ \zeta_{\mathsf{Pi}}\ \mathsf{H}_{\mathsf{Pi}}) && \text{(line 8)} \end{aligned}$$

The body of `pinger` is a `try` block followed by a series of handlers. A `try-with` statement is encoded in Olaf as $⬇^{\mathsf{L}}\ \textbf{let}\ \mathsf{H} = \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{v}^{\mathsf{L}}\ D\ \textbf{in}\ t$, where D is the implementation of the effect operation, and t is the `try`-block computation, which may invoke handler H. A handler should not outlive its `try-with` statement, so the handler value has the same lifetime as declared by the ⬇-term. In term t_{pinger} above, all three handlers, H_{E}, H_{A}, and H_{Po}, have the same lifetime L because there is only one `try`.

Operation invocations on handlers (subsuming method calls on objects) are encoded as ⇧-terms. For example, the `try`-block computation `H`$_{\mathsf{Pi}}$`.ping[H`$_{\mathsf{Po}}$`](...)` (line 4, Figure 10) invokes `H`$_{\mathsf{Pi}}$, so it is encoded in Olaf as $⇧\ \mathsf{H}_{\mathsf{Pi}}.\textbf{op}\ \mathsf{L}\ \mathsf{H}_{\mathsf{Po}}\ \ldots$, where L is the lifetime of H_{Po}. Similarly, the recursive call to `pinger` (line 8) is also encoded as a ⇧-term. (For simplicity, in the encoding above we have assumed the two operations of `Async` are combined into one.)

5.2 Operational Semantics

To give an operational semantics to Olaf, terms and evaluation contexts in Figure 11 are extended with a ⇩-construct:

$$\textit{terms}\ \ t, s ::= \ldots\ \mid\ ⇩^{\mathsf{L}}\ t \qquad \textit{evaluation contexts}\ \ K ::= \ldots\ \mid\ ⇩^{\mathsf{L}}\ K$$

Figure 12 defines the small-step operational semantics. Individual reduction steps take the form $\overline{\mathsf{L}_1}\ ;\ t_1 \longrightarrow \overline{\mathsf{L}_2}\ ;\ t_2$, meaning that term t_1 steps to term t_2 while the set of freshly created lifetime constants possibly grows from $\overline{\mathsf{L}_1}$ to $\overline{\mathsf{L}_2}$. Per rule [DOWN], a lifetime constant L_1 declared by a ⬇-term is replaced by a fresh copy L_2 when the ⬇-term is reduced to a ⇩-term. While ⬇-terms lexically bind lifetime constants, ⇩-terms are non-binding constructs; evaluation contexts of form $⇩^{\mathsf{L}}\ K$ serve as stack delimiters. This use of freshness is analogous to how calculi with reference cells allocate fresh memory locations; closed terms can mention fresh identifiers. The distinction between ⬇ and ⇩ is not apparent in Zhang and Myers [2019], albeit present in their Coq formalization; Biernacki et al. [2020] clarify the distinction.

Rule [OP] exposes the operation implementation of a handler by unrolling the fixpoint handler value. Rule [DOWNUP] handles operation invocations. To execute the operation's implementation, a resumption must be found to substitute for the free variable k. Because the operation value $\boldsymbol{v}^{\mathsf{L}_0}\ \lambda\mathsf{k}.\ t$ has lifetime L_0, the surrounding evaluation context is searched for a stack delimiter $⇩^{\mathsf{L}_0}$. The part of the evaluation context delimited by $⇩^{\mathsf{L}_0}$ is then used as the resumption. In comparison, prior

$\boxed{\overline{\mathsf{L}_1}\,;\, t_1 \longrightarrow \overline{\mathsf{L}_2}\,;\, t_2}$

$$[\textsc{ktx}]\quad \frac{\overline{\mathsf{L}_1}\,;\, t_1 \longrightarrow \overline{\mathsf{L}_2}\,;\, t_2}{\overline{\mathsf{L}_1}\,;\, K[t_1] \longrightarrow \overline{\mathsf{L}_2}\,;\, K[t_2]} \qquad [\textsc{let}]\quad \overline{\mathsf{L}}\,;\, \textbf{let } \mathsf{x} = v \textbf{ in } t \longrightarrow \overline{\mathsf{L}}\,;\, t\,\{v/\mathsf{x}\}$$

$$[\textsc{op}]\quad \overline{\mathsf{L}}\,;\, \left(\textbf{fix } \mathsf{self} \textbf{ is } \boldsymbol{\nu}^{\mathsf{L}_0}\, D\right).\textbf{op} \longrightarrow \overline{\mathsf{L}}\,;\, \boldsymbol{\nu}^{\mathsf{L}_0}\, D \left\{\textbf{fix } \mathsf{self} \textbf{ is } \boldsymbol{\nu}^{\mathsf{L}_0}\, D \middle/ \mathsf{self}\right\}$$

$$[\textsc{eapp}]\quad \overline{\mathsf{L}}\,;\, \left(\boldsymbol{\nu}^{\mathsf{L}_0}\, \Lambda\alpha.\, D\right) c \longrightarrow \overline{\mathsf{L}}\,;\, \boldsymbol{\nu}^{\mathsf{L}_0}\, D\,\{c/\alpha\}$$

$$[\textsc{lapp}]\quad \overline{\mathsf{L}}\,;\, \left(\boldsymbol{\nu}^{\mathsf{L}_0}\, \Lambda\zeta.\, D\right) \mathsf{L}_1 \longrightarrow \overline{\mathsf{L}}\,;\, \boldsymbol{\nu}^{\mathsf{L}_0}\, D\,\{\mathsf{L}_1/\zeta\}$$

$$[\textsc{app}]\quad \overline{\mathsf{L}}\,;\, \left(\boldsymbol{\nu}^{\mathsf{L}_0}\, \lambda\mathsf{x}.\, D\right) v \longrightarrow \overline{\mathsf{L}}\,;\, \boldsymbol{\nu}^{\mathsf{L}_0}\, D\,\{v/\mathsf{x}\}$$

$$[\textsc{throw}]\quad \overline{\mathsf{L}}\,;\, \textbf{throw } (\textbf{cont } K)\; t \longrightarrow \overline{\mathsf{L}}\,;\, K[t]$$

$$[\textsc{down}]\quad \overline{\mathsf{L}}\,;\, \pmb{\Downarrow}^{\mathsf{L}_1}\, t \longrightarrow \overline{\mathsf{L}},\, \mathsf{L}_2\,;\, \Downarrow^{\mathsf{L}_2}\, t\,\{\mathsf{L}_2/\mathsf{L}_1\} \quad \left(\mathsf{L}_2 \notin \overline{\mathsf{L}}\right)$$

$$[\textsc{downval}]\quad \overline{\mathsf{L}}\,;\, \Downarrow^{\mathsf{L}_0}\, v \longrightarrow \overline{\mathsf{L}}\,;\, v$$

$$[\textsc{downup}]\quad \overline{\mathsf{L}}\,;\, \Downarrow^{\mathsf{L}_0} K\left[\Uparrow \boldsymbol{\nu}^{\mathsf{L}_0}\, \lambda\mathsf{k}.\, t\right] \longrightarrow \overline{\mathsf{L}}\,;\, t\left\{\textbf{cont} \Downarrow^{\mathsf{L}_0} K \middle/ \mathsf{k}\right\} \quad \left(\mathsf{L}_0 \curvearrowright K\right)$$

$\boxed{\mathsf{L} \curvearrowright K}$

$$\mathsf{L} \curvearrowright [\cdot] \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\, c}$$

$$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\, \mathsf{L}'} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\, t}$$

$$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright v\, K} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright \Uparrow K}$$

$$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright \textbf{let } \mathsf{x} = K \textbf{ in } t}$$

$$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright \textbf{throw } K\, t} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K.\textbf{op}}$$

$$\frac{\mathsf{L} \curvearrowright K \qquad \mathsf{L} \neq \mathsf{L}_0}{\mathsf{L} \curvearrowright \Downarrow^{\mathsf{L}_0} K}$$

Figure 12. Operational semantics of Olaf

formalisms of lexically scoped handlers would use the value $\lambda\mathsf{y}.\Downarrow^{\mathsf{L}_0} K[\mathsf{y}]$, rather than **cont** $\Downarrow^{\mathsf{L}_0} K$, as the resumption. Because handlers are deep, the resumption has $\Downarrow^{\mathsf{L}_0}$ at its outermost layer. When the operation value corresponds to an ordinary function, its lifetime L_0 must have been introduced by evaluating the $\pmb{\Downarrow}^{\mathsf{L}_0}$ guarding the "main" term. So what [DOWNUP] does in this case is essentially calling the function with the "current continuation" (in Scheme parlance).

5.3 Static Semantics

Figure 13 presents the term-typing rules of Olaf. Term typing rules have form $\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : [\tau]_{\overline{e}}$, where Δ, Θ, Γ, and Ξ are binding contexts. The judgment form says that under these environments term t has type τ and effects $\overline{e}$. Rule [T-UP] types operation invocations. The effects of this term include the operation value's own lifetime and the effects in the operation's result type.

Rule [T-DOWN] shows that the lifetime constant L declared by a $\pmb{\Downarrow}^{\mathsf{L}}\, t$ term can appear in the effects of the computation guarded by $\pmb{\Downarrow}^{\mathsf{L}}$: to type-check t, the environment Θ of lifetime constants is augmented with L. The environment also tracks the type and effects $[\tau]_c$ of the entire computation $\pmb{\Downarrow}^{\mathsf{L}}\, t$. Importantly, however, L must *not* occur free in $[\tau]_c$; while t has lifetime L, $\pmb{\Downarrow}^{\mathsf{L}}\, t$ lives beyond L. Notice that we do not give a typing rule for the auxiliary $\Downarrow$-form, which only emerges when evaluating a Olaf program.

Rule [T-FIX] type-checks a handler value. The operation of the handler is type-checked with the self reference in scope. The self reference has the same type $\boldsymbol{\nu}^{\mathsf{L}}\, \mathbb{F}[\overline{c}]$ as the handler value. Using the self reference triggers the lifetime effect L.

Rule [T-KLAM] type-checks an operation value whose parameters (except for the resumption) are already in scope. A salient difference from previous work is that the type of the resumption, apart from being a continuation type, does not have to exactly match the result type and effect $[\tau_1]_{c_1}$ of the operation: the continuation can be applied to a computation with the additional lifetime effect L that is the lifetime of the current value. A consequence is that the computation the resumption is applied to is allowed to use the self reference to handle its effects.

$$\boxed{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : [\tau]_c} \qquad \Delta ::= \varnothing \mid \Delta, \alpha \qquad \Theta ::= \varnothing \mid \Theta, \zeta \qquad \Gamma ::= \varnothing \mid \Gamma, x\!:\!\tau \qquad \Xi ::= \varnothing \mid \Xi, \mathsf{L}\!:\![\tau]_c$$

$$[\text{T-UNIT}]\ \Delta \mid \Theta \mid \Gamma \mid \Xi \vdash () : [\mathbb{1}]_\varnothing \qquad [\text{T-VAR}]\ \frac{\Gamma(x) = \tau}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash x : [\tau]_\varnothing} \qquad [\text{T-UP}]\ \frac{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, [\tau]_{c_1}\right]_{c_2}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \Uparrow t : [\tau]_{c_1, c_2, \mathsf{L}}}$$

$$[\text{T-OP}]\ \frac{\begin{array}{c}\Delta, \overline{\alpha} \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \mathbb{F}[\overline{c_1}]\right]_{c_2} \\ \textit{signature}(\mathbb{F}) = \textbf{interface}\ \mathbb{F}[\overline{\alpha}]\ \{T\}\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t.\mathbf{op} : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\, \{\overline{c_1}/\overline{\alpha}\}\right]_{c_2}} \qquad [\text{T-FIX}]\ \frac{\begin{array}{c}\textit{signature}(\mathbb{F}) = \textbf{interface}\ \mathbb{F}[\overline{\alpha}]\ \{T\} \\ \Delta \mid \Theta \mid \Gamma, \mathsf{self}\!:\!\boldsymbol{\nu}^{\mathsf{L}}\, \mathbb{F}[\overline{c}] \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\, \{\overline{c}/\overline{\alpha}\}\right]_\varnothing\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \textbf{fix}\ \mathsf{self}\ \textbf{is}\ \boldsymbol{\nu}^{\mathsf{L}}\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \mathbb{F}[\overline{c}]\right]_\varnothing}$$

$$[\text{T-ELAM}]\ \frac{\Delta, \alpha \mid \Theta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\right]_\varnothing}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, \Lambda\alpha.\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \forall\alpha.\, T\right]_\varnothing} \qquad [\text{T-LLAM}]\ \frac{\Delta \mid \Theta, \zeta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\right]_\varnothing}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, \Lambda\zeta.\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \forall\zeta.\, T\right]_\varnothing}$$

$$[\text{T-LAM}]\ \frac{\Delta \mid \Theta \mid \Gamma, x\!:\!\tau \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\right]_\varnothing}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, \lambda x.\, D : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \tau \rightarrow T\right]_\varnothing} \qquad [\text{T-KLAM}]\ \frac{\begin{array}{c}\Xi(\mathsf{L}) = [\tau_2]_{c_2} \\ \Delta \mid \Theta \mid \Gamma, \mathsf{k}\!:\!\textbf{cont}\, [\tau_1]_{c_1, \mathsf{L}} \rightsquigarrow [\tau_2]_{c_2} \mid \Xi \vdash t : [\tau_2]_{c_2}\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \boldsymbol{\nu}^{\mathsf{L}}\, \lambda\mathsf{k}.\, t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, [\tau_1]_{c_1}\right]_\varnothing}$$

$$[\text{T-EAPP}]\ \frac{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \forall\alpha.\, T\right]_{c_2} \qquad \Delta \mid \Theta \mid \Xi \vdash c_1}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t\ c_1 : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\, \{c_1/\alpha\}\right]_{c_2}} \qquad [\text{T-LAPP}]\ \frac{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \forall\zeta.\, T\right]_c \qquad \Theta \mid \Xi \vdash \ell}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t\ \ell : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\, \{\ell/\zeta\}\right]_c}$$

$$[\text{T-APP}]\ \frac{\begin{array}{c}\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\boldsymbol{\nu}^{\mathsf{L}}\, \tau \rightarrow T\right]_c \\ \Delta \mid \Theta \mid \Gamma \mid \Xi \vdash s : [\tau]_c\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t\ s : \left[\boldsymbol{\nu}^{\mathsf{L}}\, T\right]_c} \qquad [\text{T-DOWN}]\ \frac{\begin{array}{c}\Delta \mid \Theta \mid \Gamma \mid \Xi, \mathsf{L}\!:\![\tau]_c \vdash t : [\tau]_{c, \mathsf{L}} \\ \Delta \mid \Theta \mid \Xi \vdash \tau \qquad \Delta \mid \Theta \mid \Xi \vdash c\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \Downarrow^{\mathsf{L}} t : [\tau]_c}$$

$$[\text{T-CONT}]\ \frac{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash K : [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \textbf{cont}\ K : \left[\textbf{cont}\, [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2}\right]_\varnothing} \qquad [\text{T-LET}]\ \frac{\begin{array}{c}\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash s : [\sigma]_c \\ \Delta \mid \Theta \mid \Gamma, x\!:\!\sigma \mid \Xi \vdash t : [\tau]_c\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \textbf{let}\ x = s\ \textbf{in}\ t : [\tau]_c}$$

$$[\text{T-THROW}]\ \frac{\begin{array}{c}\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : \left[\textbf{cont}\, [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2}\right]_{c_2} \\ \Delta \mid \Theta \mid \Gamma \mid \Xi \vdash s : [\tau_1]_{c_1}\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash \textbf{throw}\ t\ s : [\tau_2]_{c_2}} \qquad [\text{T-SUB}]\ \frac{\begin{array}{c}\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : [\tau_1]_{c_1} \\ \vdash c_1 \le c_2 \qquad \vdash \tau_1 \le \tau_2\end{array}}{\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : [\tau_2]_{c_2}}$$

Figure 13. Rules for typing Olaf terms

Rules for type-level well-formation and orderings are deferred to the accompanying technical report [Zhang et al. 2020]. Because composite effects are sets, the subeffecting relation $\vdash c_1 \le c_2$ simply says c_1 is a subset of c_2.

6 ESTABLISHING PARAMETRICITY FOR A LOGICAL-RELATIONS MODEL

As Section 4.2 argues, the key property to establish about Olaf should be parametricity. To this end, this section develops a logical-relations model for Olaf, and shows it satisfies parametricity and is sound with respect to contextual equivalence. These results, fully mechanized in Coq, provide a rigorous account for abstraction safety and also imply type safety.

Semantic Types

$$
\begin{aligned}
\llbracket \mathbb{1} \rrbracket_{\theta}^{\delta}(\omega,\, v_1,\, v_2) &\overset{def}{=} v_1 = () \;\wedge\; v_2 = () \\
\llbracket \boldsymbol{\nu}^{\ell}\, \mathbb{F}[\overline{c}] \rrbracket_{\theta}^{\delta}(\omega,\, v_1,\, v_2) &\overset{def}{=} \exists T.\, \mathit{signature}(\mathbb{F}) = \textbf{interface}\; \mathbb{F}[\overline{\alpha}]\,\{T\} \;\wedge \\
&\qquad \exists \mathsf{L}_1, D_1, \mathsf{L}_2, D_2.\, v_i = \textbf{fix}\; \mathsf{self}\; \textbf{is}\; \boldsymbol{\nu}^{\mathsf{L}_i}\, D_i\;{}_{(i\,=\,1,\,2)} \;\wedge \\
&\qquad \triangleright \left\llbracket \boldsymbol{\nu}^{\ell}\, T\,\{\overline{c}/\overline{\alpha}\} \right\rrbracket_{\theta}^{\delta}\left(\omega,\, \boldsymbol{\nu}^{\mathsf{L}_1} D_1,\, \boldsymbol{\nu}^{\mathsf{L}_2} D_2\right) \\
\llbracket \boldsymbol{\nu}^{\ell}\, T \rrbracket_{\theta}^{\delta}(\omega,\, v_1,\, v_2) &\overset{def}{=} \exists \mathsf{L}_1, D_1, \mathsf{L}_2, D_2.\, \theta_i \ell = \mathsf{L}_i \;\wedge\; v_i = \boldsymbol{\nu}^{\mathsf{L}_i}\, D_i\;{}_{(i\,=\,1,\,2)} \;\wedge \\
&\qquad \llbracket T \rrbracket_{\theta}^{\delta}(\omega,\, \ell,\, D_1,\, D_2) \\
\llbracket \textbf{cont}\, [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2} \rrbracket_{\theta}^{\delta}(\omega,\, v_1,\, v_2) &\overset{def}{=} \exists K_1, K_2.\, v_i = \textbf{cont}\, K_i\;{}_{(i\,=\,1,\,2)} \;\wedge\; \mathcal{K}\left\llbracket [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2} \right\rrbracket_{\theta}^{\delta}(\omega,\, K_1,\, K_2)
\end{aligned}
$$

Semantic Operation Signatures

$$
\begin{aligned}
\llbracket \forall \alpha.\, T \rrbracket_{\theta}^{\delta}(\omega,\, \ell,\, D_1,\, D_2) &\overset{def}{=} \exists D_1', D_2'.\, D_i = \Lambda\alpha.\, D_i'\;{}_{(i\,=\,1,\,2)} \;\wedge\; \forall \omega', \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}, \psi.\, \omega \subseteq \omega' \Rightarrow \\
&\qquad \psi \subseteq \omega' \Rightarrow \llbracket T \rrbracket_{\theta}^{\alpha \mapsto \left\langle \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}, \psi \right\rangle}\, \delta \left(\omega',\, \ell,\, D_1' \left\{\overline{\mathsf{L}_1} \middle/ \alpha_1\right\},\, D_2' \left\{\overline{\mathsf{L}_2} \middle/ \alpha_2\right\}\right) \\
\llbracket \forall \zeta.\, T \rrbracket_{\theta}^{\delta}(\omega,\, \ell,\, D_1,\, D_2) &\overset{def}{=} \exists D_1', D_2'.\, D_i = \Lambda\zeta.\, D_i'\;{}_{(i\,=\,1,\,2)} \;\wedge\; \forall \omega', \mathsf{L}_1, \mathsf{L}_2, \varphi'.\, \omega \subseteq \omega' \Rightarrow \\
&\qquad \mathsf{L}_i \in \omega_i'\;{}_{(i\,=\,1,\,2)} \Rightarrow \llbracket T \rrbracket_{\theta,\, \zeta \mapsto \langle \mathsf{L}_1, \mathsf{L}_2, \varphi' \rangle}^{\delta}\left(\omega',\, \ell,\, D_1'\,\{\mathsf{L}_1/\zeta_1\},\, D_2'\,\{\mathsf{L}_2/\zeta_2\}\right) \\
\llbracket \tau \rightarrow T \rrbracket_{\theta}^{\delta}(\omega,\, \ell,\, D_1,\, D_2) &\overset{def}{=} \exists D_1', D_2'.\, D_i = \lambda \mathsf{x}.\, D_i'\;{}_{(i\,=\,1,\,2)} \;\wedge\; \forall \omega', v_1, v_2.\, \omega \subseteq \omega' \Rightarrow \\
&\qquad \llbracket \tau \rrbracket_{\theta}^{\delta}(\omega',\, v_1,\, v_2) \Rightarrow \llbracket T \rrbracket_{\theta}^{\delta}\left(\omega',\, \ell,\, D_1'\,\{v_1/\mathsf{x}\},\, D_2'\,\{v_2/\mathsf{x}\}\right) \\
\llbracket [\tau]_c \rrbracket_{\theta}^{\delta}(\omega,\, \ell,\, D_1,\, D_2) &\overset{def}{=} \exists t_1, t_2.\, D_i = \lambda \mathsf{k}.\, t_i\;{}_{(i\,=\,1,\,2)} \;\wedge\; \forall \omega', K_1, K_2.\, \omega \subseteq \omega' \Rightarrow \\
&\qquad \mathcal{K}_{\mathcal{T}\llbracket [\tau]_{c,\ell} \rrbracket_{\theta}^{\delta} \rightsquigarrow \mathcal{L}\llbracket \ell \rrbracket_{\theta}^{\delta}}(\omega',\, K_1,\, K_2) \Rightarrow \mathcal{L}\llbracket \ell \rrbracket_{\theta}^{\delta}(\omega',\, t_1\,\{\textbf{cont}\, K_1/\mathsf{k}\},\, t_2\,\{\textbf{cont}\, K_2/\mathsf{k}\})
\end{aligned}
$$

Semantic Effects

$$
\begin{aligned}
\llbracket \alpha \rrbracket_{\theta}^{\delta}\left(\omega,\, t_1,\, t_2,\, \varphi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) &\overset{def}{=} \exists \psi.\, \delta(\alpha) = \psi \;\wedge\; \psi\left(\omega,\, t_1,\, t_2,\, \varphi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) \\
\llbracket \ell \rrbracket_{\theta}^{\delta}(\omega,\, t_1,\, t_2,\, \varphi,\, \mathsf{L}_1,\, \mathsf{L}_2) &\overset{def}{=} \exists \mathsf{L}_1, \mathsf{L}_2, s_1, s_2.\, \theta_i \ell = \mathsf{L}_i \;\wedge\; t_i = \Uparrow \boldsymbol{\nu}^{\mathsf{L}_i}\, \lambda \mathsf{k}.\, s_i\;{}_{(i\,=\,1,\,2)} \;\wedge \\
&\qquad \forall \omega', K_1, K_2.\, \omega \subseteq \omega' \Rightarrow \triangleright \mathcal{K}_{\varphi \rightsquigarrow \mathcal{L}\llbracket \ell \rrbracket_{\theta}^{\delta}}(\omega',\, K_1,\, K_2) \Rightarrow \\
&\qquad \triangleright \mathcal{L}\llbracket \ell \rrbracket_{\theta}^{\delta}(\omega',\, s_1\,\{\textbf{cont}\, K_1/\mathsf{k}\},\, s_2\,\{\textbf{cont}\, K_2/\mathsf{k}\}) \\
\llbracket c \rrbracket_{\theta}^{\delta}\left(\omega,\, t_1,\, t_2,\, \varphi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) &\overset{def}{=} \exists e \in c.\, \llbracket e \rrbracket_{\theta}^{\delta}\left(\omega,\, t_1,\, t_2,\, \varphi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right)
\end{aligned}
$$

Auxiliary Relations

$$
\begin{aligned}
\mathcal{K}_{\varphi_1 \rightsquigarrow \varphi_2}(\omega,\, K_1,\, K_2) &\overset{def}{=} \forall \omega', t_1, t_2.\, \omega \subseteq \omega' \Rightarrow \varphi_1(\omega',\, t_1,\, t_2) \Rightarrow \varphi_2(\omega',\, K_1[t_1],\, K_2[t_2]) \\
\mathcal{K}\left\llbracket [\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2} \right\rrbracket_{\theta}^{\delta}(\omega,\, K_1,\, K_2) &\overset{def}{=} \mathcal{K}_{\mathcal{T}\llbracket [\tau_1]_{c_1} \rrbracket_{\theta}^{\delta} \rightsquigarrow \mathcal{T}\llbracket [\tau_2]_{c_2} \rrbracket_{\theta}^{\delta}}(\omega,\, K_1,\, K_2) \\
\mathcal{L}\llbracket \zeta \rrbracket_{\theta}^{\delta}(\omega,\, t_1,\, t_2) &\overset{def}{=} \exists \varphi.\, \theta(\zeta) = \varphi \;\wedge\; \varphi(\omega,\, t_1,\, t_2) \\
\mathcal{L}\llbracket \mathsf{L} \rrbracket_{\theta}^{\delta}(\omega,\, t_1,\, t_2) &\overset{def}{=} \exists \tau, c.\, \Xi(\mathsf{L}) = [\tau]_c \;\wedge\; \mathcal{T}\llbracket [\tau]_c \rrbracket_{\theta}^{\delta}(\omega,\, t_1,\, t_2) \\
\psi \subseteq \omega &\overset{def}{=} \forall \omega', t_1, t_2, \varphi, \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}.\, \psi\left(\omega',\, t_1,\, t_2,\, \varphi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) \Rightarrow \overline{\mathsf{L}_i} \subseteq \omega_i\;{}_{(i\,=\,1,\,2)}
\end{aligned}
$$

Figure 14. Relational interpretations of types, operation signatures, and effects. (The definitions are implicitly indexed by Δ, Θ, and Ξ.)

6.1 A Logical-Relations Model for Olaf

Technical devices used in the definition include step indexing [Ahmed 2006; Appel and McAllester 2001], possible worlds, and biorthogonality [Pitts and Stark 1998] to deal with challenges such as Turing-completeness, freshly generated lifetimes, and delimited continuations. Figure 14 presents the semantic interpretation of various type-level entities. Other definitions—including observational refinement, the relation on closed terms $\mathcal{T}[\![[\tau]_c]\!]^{\delta}_{\theta}(\omega, t_1, t_2)$, and its lifting to open terms $\Delta \mid \Theta \mid \Gamma \mid \Xi \vDash t_1 \approx_{log} t_2 : [\tau]_c$—are largely standard and for space are deferred to the technical report.

A logical-relations model for a typed language interprets types as relations. As is standard, the relational interpretations in Figure 14 are indexed by substitutions δ and θ that provides—in addition to syntactic substitution functions (denoted by δ_i and θ_i where $i = 1, 2$) for free effect variables and lifetime variables in the type-level entity being interpreted—semantic interpretation of the free variables. The logical relations are also indexed by a world ω of freshly created lifetime constants; $\omega \subseteq \omega'$ means world ω' is a future world of (i.e., extends) ω.

Language features like recursion make it difficult to define these relations inductively on the structure of types. Fixpoint handlers and mutually recursive interfaces in Olaf pose a similar challenge. The technique of step indexing [Ahmed 2006; Appel and McAllester 2001] addresses this challenge. Our logical relation is step-indexed; the logical relation is defined using a double induction, first on a step index, and second on the structure of types. The definition is given in terms of a logic equipped with the “later” modality $\triangleright$, which offers a clean abstraction of step indexing [Appel et al. 2007; Dreyer et al. 2009]. For example, the relational interpretation of an interface type $[\![\boldsymbol{\nu}^{\ell}\, \mathbb{F}[\overline{c}]]\!]^{\delta}_{\theta}$ is defined as that of an operation type $[\![\boldsymbol{\nu}^{\ell}\, T\,\{\overline{c}/\overline{\alpha}\}]\!]^{\delta}_{\theta}$ guarded by $\triangleright$. Although T may recursively mention $\mathbb{F}$, the use of $\triangleright$ ensures the definition remains well-founded.

The relational interpretation of continuation types $[\![\mathbf{cont}\,[\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2}]\!]^{\delta}_{\theta}$ is defined in terms a relation on evaluation contexts $\mathcal{K}[\![[\tau_1]_{c_1} \rightsquigarrow [\tau_2]_{c_2}]\!]^{\delta}_{\theta}$. The latter relation is a standard auxiliary definition in logical-relations proofs. Two evaluation contexts K_1 and K_2 are in this relation if applying them to terms t_1 and t_2 related by $\mathcal{T}[\![[\tau_1]_{c_1}]\!]^{\delta}_{\theta}$ implies the resulting terms $K_1[t_1]$ and $K_2[t_2]$ are related by $\mathcal{T}[\![[\tau_2]_{c_2}]\!]^{\delta}_{\theta}$.

The interpretation of an operation type $[\![\boldsymbol{\nu}^{\ell}\, T]\!]^{\delta}_{\theta}$ is defined in terms of that of the operation signature $[\![T]\!]^{\delta}_{\theta}$, indexed on the lifetime ℓ. Of particular interest is the interpretation $[\![[\tau]_c]\!]^{\delta}_{\theta}$. It relates two operation implementations $\lambda \mathsf{k}.\, t_1$ and $\lambda \mathsf{k}.\, t_2$ in which t_1 and t_2 are related when the free variables k standing for resumptions are replaced by related continuations. The continuations are allowed to be related by $\mathcal{K}_{\mathcal{T}[\![[\tau]_{c,\ell}]\!]^{\delta}_{\theta} \rightsquigarrow \mathcal{L}[\![\ell]\!]^{\delta}_{\theta}}$, where the occurrence of ℓ in addition to c indicates recursive handling by the fixpoint handler, corresponding to the premise of typing rule [T-KLAM].

The semantic interpretation of atomic or composite lifetime effects $[\![e]\!]^{\delta}_{\theta}$ or $[\![c]\!]^{\delta}_{\theta}$ follows techniques developed in Biernacki et al. [2018] and Zhang and Myers [2019].

6.2 Results

Parametricity is a strong indicator that abstraction is preserved [Dreyer 2018; Reynolds 1983]. It implies that effect-polymorphic functions behave uniformly, irrespective of the choice of effects with which they are instantiated.

Theorem 1 (Parametricity, a.k.a. Abstraction Theorem, a.k.a. Fundamental Property). If Ξ and Γ are well-formed, then $\Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t : [\tau]_c$ implies $\Delta \mid \Theta \mid \Gamma \mid \Xi \vDash t \approx_{log} t : [\tau]_c$.

Type safety means that well-typed Olaf programs do not get stuck—they either reduce to values or diverge. Type safety follows from parametricity.

Theorem 2 (Type Safety). If $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash t : [\tau]_{\varnothing}$ and $\varnothing\, ;\, t \longrightarrow^{*} \overline{L'}\, ;\, t'$, then either there exists v such that $t' = v$ or there exists $\overline{L''}$ and t'' such that $\overline{L'}\, ;\, t' \longrightarrow \overline{L''}\, ;\, t''$.

Abstraction is preserved when no clients can distinguish between implementations of the same abstraction. The gold standard of indistinguishability is *contextual equivalence* [Morris 1968], whose definition in the context of Olaf can be found in the technical report. If the logical-relations model is sound, in the sense that logically related terms are contextually equivalent, then indistinguishability can be established through logical relatedness.

Theorem 3 (Soundness w.r.t. contextual equivalence).
$\Delta \mid \Theta \mid \Gamma \mid \Xi \vDash t_1 \approx_{log} t_2 : [\tau]_c \Rightarrow \Delta \mid \Theta \mid \Gamma \mid \Xi \vdash t_1 \approx_{ctx} t_2 : [\tau]_c$.

These results prove our claim that the type system of Olaf upholds strong abstraction boundaries.

6.3 Formalization in Coq

The formal results above have been mechanized using the Coq proof assistant, in 17,800 lines of code architected similarly to prior work [Biernacki et al. 2018; Zhang and Myers 2019]. We also wrote a 400-line extension of the IxFree library [Polesiuk 2017]—a shallow embedding of a logic with the $\triangleright$ modality [Dreyer et al. 2009]—to implement the logical relations as dependently typed fixpoint functions. Cofinite quantification [Aydemir et al. 2008] makes it easy to generate fresh lifetime constants. This Coq implementation is available at https://github.com/yizhouzhang/olaf-coq.

7 IMPLEMENTATION ISSUES

While we leave a full-featured compiler to future work, we discuss two key compilation issues.

7.1 Tail-Resumption Optimization

Being able to apply the tail-resumption optimization is important because calling resumptions at the tail position is probably the most common way handler resumptions are used in practice—in the preceding examples, all handlers except those for `exn` or `await` are tail-resumptive. The optimization avoids having to capture the handler resumption as a first-class value, which would otherwise involve copying stack frames, a rather expensive operation.

Bidirectional handlers can benefit from this optimization too. Per rule [DOWNUP], a tail-resumptive handler in Olaf is reduced as follows:

$$\overline{\mathsf{L}}\,;\ \Downarrow^{\mathsf{L}_0} K\big[\Uparrow \nu^{\mathsf{L}_0}\,\lambda \mathsf{k}.\,\textbf{throw}\ \mathsf{k}\ t\big] \longrightarrow \overline{\mathsf{L}}\,;\ \textbf{throw}\left(\textbf{cont}\ \Downarrow^{\mathsf{L}_0} K\right) t \longrightarrow \overline{\mathsf{L}}\,;\ \Downarrow^{\mathsf{L}_0} K[t]$$

There is no need to reify the delimited continuation $\textbf{cont}\ \Downarrow^{\mathsf{L}_0} K$. It remains as the surrounding evaluation context after two steps of reduction.

7.2 Translation into Unidirectional Effect Handlers

An obvious compilation target for bidirectional handlers is a language with deep, ordinary effect handlers, and an obvious approach to this compilation is as follows: effect operations are translated to return callbacks, and invocations of operations are translated to call the callbacks. For example, effects `Ping` and `Pong` from Figure 8a are translated into these signatures:

```
effect Ping { def ping() : () → void raises Pong | Ping }
effect Pong { def pong() : () → void raises Ping | Pong }
```

The type of the returned callback additionally includes the effect being translated, so that the callback computation can raise effects that are to be handled by the (deep) handler being defined. Whether such a type-preserving translation is feasible in general should be examined per target language—*macro expressivity* [Felleisen 1991] in the context of control-flow mechanisms is sensitive to the precise set of cross-cutting features under consideration [Forster et al. 2017].

Importantly, the translation outlined above bears unpleasant performance implications: in a tail-resumptive setting, the cost of communicating through callbacks could be avoided if a handler computation were allowed to directly raise effects to the surrounding evaluation context.

To understand this cost empirically, we use a modified implementation of μC++ [Buhr 2019]. The μC++ language extends C++ [Stroustrup 1987] with effect handlers that are either abortive or tail-resumptive. The modified version allows tail-resumptive effect handlers to directly raise effects to their resumptions.

Historically, in the absence of a static effect system, bidirectional control flow has been banned because it is considered too complex to reason about or use. For example, Mesa [Mitchell et al. 1979], one of the few languages with resumable exceptions, forbids recursive handling. Similarly, μC++ does not check effects statically, and the original implementation uses an extra run-time check to prevent effects raised by a handler from being handled by the handler resumption [Buhr 2019, §5.5]. Our type system addresses this concern by offering reasoning principles for bidirectional, recursive effect handling.

We performed two hand translations of the ping-pong communication example (Section 3.3, with `ponger` implemented as in Section 3.3.2) into μC++, with one of the two using callbacks. This program was chosen because it exercises high-frequency bidirectional control transfer. We ran the hand-translated code using the modified μC++ implementation with the extra run-time check disabled, and measured the running time on a 3.2GHz Intel Xeon Gold processor, averaging 500 runs. The translation relying on callbacks incurred a 2.1× slowdown: 42.6 ms vs. 19.8 ms. This result argues for bidirectional handlers as a first-class language feature: obtaining bidirectionality via a desugaring into callbacks is less efficient. (In the other way around, compiling callbacks away into efficient bidirectionality would involve a complex interprocedural analysis.)

8 RELATED WORK

Control effects. Effect handlers [Bauer and Pretnar 2015; Plotkin and Pretnar 2013] offer a form of *delimited control* [Danvy and Filinski 1990; Felleisen 1988] (or *coroutining* [Haynes et al. 1986]), together with a nice separation between the syntax of effects and their semantics. Whereas Forster et al. [2017] show effect handlers and (a particular variant of) delimited-control operators fail to macro-express [Felleisen 1991] one another while preserving typing, Piróg et al. [2019] show they are equally expressive when their type systems support polymorphic operations and answer-type polymorphism [Asai and Kameyama 2007], respectively. The core language Olaf further blurs the boundaries: key elements of algebraic effects (i.e., effect signatures and handlers) and those of delimited control (i.e., a pair of control operators) coexist and play complementary roles in Olaf.

Bidirectionality is possible with effect handlers or delimited-control operators using, for example, callbacks (Section 7.2)—however, as we discuss later, Olaf is likely not macro-expressible by recent formalisms that also lexically scope effect handlers [Biernacki et al. 2020; Zhang and Myers 2019]. Bidirectional handlers inherit the appeal of algebraic effects, and address bidirectional control flow with a straightforward programming style, an economy of language constructs, efficient compilation, and strong reasoning principles.

Applications of bidirectional handlers. Interruptible iterators [Liu et al. 2006] generalize the expressive power of generators to allow concurrent updates to the underlying collection being iterated over. The example in Section 3.1 shows that bidirectional algebraic effects capture the expressive power of interruptible iterators but as part of a single unified effect mechanism with formally defined semantics, rather than by introducing interrupts as a separate mechanism.

A key motivation for promises and async–await as language features was to enable better exception handling; the state of a promise indicates if an exception occurred asynchronously. However, the lack of static checking on asynchronous exceptions makes software error-prone [Alimadadi et al. 2018]. Previous encodings of async–await as algebraic effects are unsatisfactory: as discussed in Section 3.2, they either do not express asynchronous exceptions as an algebraic effect [Leijen 2017a], or require ad hoc constructs and do not track exceptions statically [Dolan et al. 2017].

Session types [Honda et al. 1999] are a behavioral-typing discipline for communication protocols. The possibility of encoding session types using algebraic effects has been hypothesized before [Fowler et al. 2019], and bidirectional effects make this connection more substantial. The encoding does not yet offer the full power of session types, though; it enforces weaker session fidelity and does not prevent deadlocks. Linear effect handlers are a promising future direction.

Preventing accidental handling. Accidental handling of algebraic effects in the presence of effect polymorphism is a known problem. Tunneling (lexically scoped, lifetime-bounded handlers) as a way to avoid accidental handling of exceptions was introduced by Zhang et al. [2016]; follow-on work adapted it to explicit effect polymorphism and proved parametricity [Zhang and Myers 2019]. Brachthäuser et al. [2018, 2020] implement tunneled algebraic effects in a Scala library, called Effekt, that encodes lifetime effects through Scala's intersection types and path-dependent types.

Biernacki et al. [2020] distinguish between an *open* and a *generative* semantics of lexically scoped handlers; generativity is critical to ensuring parametricity when effect operations can be effect-polymorphic. Olaf uses the generative semantics—its operational semantics creates fresh lifetimes. Generativity is occasionally found in prior work on control effects (e.g., [Bauer and Pretnar 2015; Gunter et al. 1995]), but without effects being statically tracked by a type-and-effect system.

We conjecture that Olaf cannot be macro-expressed by prior calculi of lexically scoped, generative handlers (an inexpressivity proof like that of Forster et al. [2017] is beyond the scope of this paper): the calculus of Zhang and Myers [2019] does not support effect-parameterized signatures (Section 4.1) or effect-polymorphic operations [Biernacki et al. 2020], either of which could help cause accidental handling; and Biernacki et al. [2020] do not support recursively defined signatures, needed to mimic fixpoint handlers (Section 7.2). Previous parametricity results do not carry over.

Another way to avoid accidental handling—in languages where composite effects are *rows* [Wand 1991] rather than sets—is directives to signal that effects should bypass the dynamically closest enclosing handler. These directives include the *inject* function of Leijen [2014], *lift* and *coercions* of Biernacki et al. [2018, 2019], and *adaptors* of Convent et al. [2020]. For example, the semantics of applying the lift construct $[\cdot]_A$ to a computation whose effect is a polymorphic row α is as follows: statically, the lifted computation has effects $A|\alpha$; dynamically, an A effect raised by the original computation is handled by the dynamically *second* closest enclosing A handler. An enclosing handler for A thus cannot intercept A effects raised by the computation, because its effect α is not considered as a subeffect of the row $A|\alpha$; an explicit *lift* is needed to please the type checker.

Generalizing algebraic effects. Lindley et al. [2017] treat Frank's handling construct as a higher-order function that pattern-matches on the effects of its computation-typed arguments. This design decision is orthogonal to ours: while we generalize effect signatures and handlers, Frank generalizes the `try–with` construct. It is plausible that either idea can be adapted to the other's setting, though Frank's shallow-handling semantics is incompatible with handlers being fixpoints.

9 CONCLUSION

This paper proposes a new design for effect handlers, in which a handler can raise effects and have its resumption—including the handler itself—handle the effects. As our examples show, these ideas address a need common to assorted programming challenges for better bidirectional communication between software components. The expressive power falls out naturally when effect operations and handlers are unified with methods and objects; however, the ideas also generalize to non-object-oriented languages. We captured the essence of the new mechanism in a core language with some distinctive features such as fixpoint handlers and the ability to treat handler resumptions as evaluation contexts. Bidirectionality exposes previously unidentified ways to accidentally handle effects that propagate per the usual, signature-based semantics; hence, we make bidirectional handlers lexically scoped and focused on a convincing proof that they are compatible with strong abstraction boundaries. While a complete implementation is left to future work, experiments suggest bidirectional handlers can be compiled efficiently.

The recent flurry of language designs for advanced control-flow features show that modern software needs language-based support for complex control flow. Bidirectional algebraic effects address this challenge in two important ways. First, they unify various previously separately proposed language features (interruptible iterators, exceptional async/await, etc.) via a natural generalization of effect handlers. This unification should help increase confidence in language metatheories and lower the hurdle for use of powerful control-flow features. Second, the guarantees that no effects are unhandled or accidentally handled are critical to writing safer code. They help the programmer manage the control-flow complexity via a type system; static typing offers guidance on where to apply effect handling, and the parametricity guarantee enables truly compositional reasoning. While these consequences are entirely anticipated, future software-engineering studies could assess the empirical effectiveness of bidirectional algebraic effects in achieving these goals.

Together, these contributions offer an appealing way to support complex control flow in mainstream programming languages.

ACKNOWLEDGMENTS

We thank Jonathan Brachthäuser, Peter Buhr, Filip Sieczkowski, and the anonymous reviewers for their valuable feedback. We also thank Peter Buhr for patching μC++.

This work was supported by NASA grant NNX16AB09G. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any government agency.

A ADT EXAMPLE

```
// The algebraic data type
data YieldResult[X] =
| ToContinue
| ToReplace(X)
| ToBehead

effect Yield[X] {
  def yield(X) : YieldResult[X]
}

// The need for a Behead effect cannot be
// easily dismissed: the iter code has to raise
// it to the caller and wait for control to
// come back.
effect Behead {
  def behead() : void
}
```

(a) ADT definition and effect signatures

```
class Node[X] {
  var head : X
  var tail : Node[X]
  …
  def iter() : void raises Yield[X] | Behead {
    match yield(head) {
    | ToContinue ⇒ skip
    | ToReplace(x) ⇒ head = x
    | ToBehead ⇒ behead() // convert ADT value to algebraic effect
    }
    if (tail != null)
      try { tail.iter() }
      with behead() {
        tail = tail.tail
        resume()
      }
  }
}
```

(b) Iterator pattern-matches the result of yield (cf. Figure 4b)

Figure 15. Using an ADT to encode iterator interrupts. By comparison, bidirectional algebraic effects allow for more concise code.

REFERENCES

A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming*, 2006. Extended/corrected version available as Harvard University TR-01-06.

S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding broken promises in asynchronous JavaScript programs. *Proc. ACM on Programming Languages*, 2(OOPSLA), Oct. 2018.

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Programming Languages and Systems*, 23(5), Sept. 2001.

A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *34th ACM Symp. on Principles of Programming Languages (POPL)*, 2007.

K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *5th Asian Symposium on Programming Languages and Systems (APLAS)*, 2007.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, 2008.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015.

G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: Formalizing asynchronous C#. In *26th European Conf. on Object-Oriented Programming*, 2012.

D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM on Programming Languages*, 2(POPL), Jan. 2018.

D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Abstracting algebraic effects. *Proc. ACM on Programming Languages*, 3(POPL), Jan. 2019.

D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM on Programming Languages*, 4(POPL), Jan. 2020.

J. I. Brachthäuser, P. Schuster, and K. Ostermann. Algebraic effects for the masses. *Proc. ACM on Programming Languages*, 2 (OOPSLA), Oct. 2018.

J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala. *J. Functional Programming*, 30, Mar. 2020.

O. Bračevac, N. Amin, G. Salvaneschi, S. Erdweg, P. Eugster, and M. Mezini. Versatile event correlation with algebraic effects. *Proc. ACM on Programming Languages*, 2(ICFP), Aug. 2018.

P. A. Buhr. μC++ annotated reference manual, version 7.0.0. Technical report, School of Computer Science, University of Waterloo, 2019.

B. Cabral and P. Marques. Hidden truth behind .NET's exception handling today. *IET Software*, 1(6), 2007.

L. Convent, S. Lindley, C. McBride, and C. McLaughlin. Doo bee doo bee doo. *J. Functional Programming*, 30, Mar. 2020.

O. Danvy and A. Filinski. Abstracting control. In *ACM Conf. on LISP and Functional Programming*, pages 151–160, 1990.

S. Dolan, S. Eliopoulos, D. Hillerström, A. Madhavapeddy, K. C. Sivaramakrishnan, and L. White. Concurrent system programming with effect handlers. In *Trends in Functional Programming*, 2017.

D. Dreyer. Milner award lecture: The type soundness theorem that you really want to prove (and now you can). In *45th ACM Symp. on Principles of Programming Languages (POPL)*, 2018.

D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *24th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 2009.

ECMA International. ECMAScript 2018 language specification. Standard-ECMA 262, June 2018.

M. Felleisen. The theory and practice of first-class prompts. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 180–190, 1988.

M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1), 1991.

Y. Forster, O. Kammar, S. Lindley, and M. Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM on Programming Languages*, 1(ICFP), Aug. 2017.

S. Fowler, S. Lindley, J. G. Morris, and S. Decova. Exceptional asynchronous session types. In *46th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 2019.

J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. *The Java Language Specification*. Oracle America, se 11 edition, Aug 2018.

R. E. Griswold, D. R. Hanson, and J. T. Korb. Generators in Icon. *ACM Trans. on Programming Languages and Systems*, 3(2): 144–161, Apr. 1981.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 282–293. ACM Press, 2002.

C. A. Gunter, D. Rémy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *7th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

C. T. Haynes, D. P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages*, 11 (3–4):143–153, 1986.

A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language.* Addison-Wesley, 1st edition, Oct. 2003. ISBN 0321154916.

D. Hillerström and S. Lindley. Shallow effect handlers. In *Asian Symp. on Programming Languages and Systems*, 2018.

K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, 1999.

O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *18th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2013.

S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

C. Lattner and J. Groff. Async/await for Swift. https://gist.github.com/lattner/429b9070918248274f25b714dcfc7619, 2019.

D. Leijen. Koka: Programming with row polymorphic effect types. In *5th Workshop on Mathematically Structured Functional Programming*. EPTCS, 2014.

D. Leijen. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, 2017a.

D. Leijen. Type directed compilation of row-typed algebraic effects. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017b.

D. Leijen. Implementing algebraic effects in C. In *15th Asian Symposium on Programming Languages and Systems (APLAS)*, 2017c.

S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1988.

B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Comm. of the ACM*, 20(8):564–576, Aug. 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems.*

J. Liu, A. Kimball, and A. C. Myers. Interruptible iterators. In *33rd ACM Symp. on Principles of Programming Languages (POPL)*, POPL '06, pages 283–294, Jan. 2006.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, POPL '88, pages 47–57, 1988.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1991.

J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual version 5.0. Technical Report CSL-79-3, Xerox Research Center, Palo Alto, Ca., 1979.

J. H. Morris, Jr. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Trans. on Programming Languages and Systems*, 18(1):1–15, Jan. 1996.

Node. Node.js. https://nodejs.org.

S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in C#. In *36th Int'l Conf. on Software Engineering (ICSE)*, 2014.

M. Piróg, P. Polesiuk, and F. Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD)*, 2019.

A. M. Pitts and I. Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–273, 1998.

G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, Feb 2003.

G. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, Dec. 2013.

P. Polesiuk. IxFree: Step-indexed logical relations in Coq. In *3rd International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

Rust language team. Async & await in Rust: a full proposal. https://boats.gitlab.io/blog/post/2018-04-06-async-await-final, 2018.

M. Shaw, W. Wulf, and R. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Comm. of the ACM*, 20(8), Aug. 1977.

B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, 2nd edition, 2004. ISBN 0-974-51405-5.

M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

G. van Rossum. *The Python Language Reference Manual*. Network Theory, Ltd., Sept. 2003.

M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1), 1991.

Y. Zhang and A. C. Myers. Abstraction-safe effect handlers via tunneling. *Proc. ACM on Programming Languages*, 3(POPL), Jan. 2019.

Y. Zhang, G. Salvaneschi, Q. Beightol, B. Liskov, and A. C. Myers. Accepting blame for safe tunneled exceptions. In *37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 281–295, June 2016.

Y. Zhang, G. Salvaneschi, and A. C. Myers. Handling bidirectional control flow: Technical report. https://arxiv.org/abs/2010.09073, 2020.