Structured Asynchrony with Algebraic Effects

Microsoft Technical report, MSR-TR-2017-21

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

Algebraic effect handlers generalize many control-flow abstractions that are implemented specially in most languages, like exception handling, iterators, or backtracking. In this article, we show how we can implement full support for asynchronous programming *as a library* using just algebraic effect handlers. The consistent type driven approach also leads naturally to powerful abstractions like block-scoped interleaving, cancellation, and timeout's that are lacking in other major asynchronous frameworks. We also introduce the concept of *ambient state* to reason about state that is local to the current strand of asynchronous execution.

Keywords Algebraic effects, Koka, effect types, asynchronous programming

1. Introduction

Suppose I design a programming language that should support complex control flow statements like exception handling, iterators (*yield*), and even asynchrony (*async-await*). I could take the C#, C++, or JavaScript way and implement each of these specially: first I change the compiler to add special keywords and syntax to the language that are checked with specific type rules. Then I extend the runtime with exception handling stack frames for exceptions. For iterators and *async-await* it is more complicated and I also need to implement special compiler transformations to turn regular code into stack restoring state machines etc. I also need to take special care that all of these features interact as expected, e.g. ensure that *finally* blocks from my exception handling code are not forgotten in the state machines for the iterator code. It is possible to do this, but it is a difficult road to travel. Moreover, I need to do all of this again for the *next* control flow abstraction that comes along.

Or I could have used *algebraic effect handlers* instead!

Algebraic effects (Plotkin and Power, 2003) and their extension with handlers (Plotkin and Pretnar, 2013, 2009), come from category theory as a way to reason about semantics of effects. An algebraic effect has interface in terms of a set of *operations* (and laws governing those operations), and we can give those operations a semantics through an effect *handler*. This single mechanism can describe any (algebraic) free monad and as a consequence generalizes most control-flow abstractions that need to be handled specially in many other languages. We argue that it is better if a language just implements support for algebraic effects and implements all other abstractions on top of that. This has several advantages:

- Algebraic effects have a solid semantic foundation in category theory and are well understood. They can be *composed* freely and their composition is always well defined.
- It can lead to simpler and more efficient runtimes and compilers since there is just one mechanism that needs to be supported well. As shown in Section 4.1, the operational semantics is simple and offers many opportunities for optimization. Moreover, it is an *untyped dynamic* semantics – meaning that algebraic effects can be applied in many settings and that static types (as in Koka) are not essential.
- With general algebraic effects library writers are empowered to implement various high-level abstractions without needing special compiler support or language extensions.

This paper does not give a final answer to the above argument. However, we answer part of it by giving an overview of how we can implement the asynchronous *async-await* (and exceptions and iterators) abstraction *as a library* using just algebraic effects. In particular:

- We implement full support for asynchronous programming in the style of *async-await* but as a library using just plain algebraic effect handlers without adding special primitives to our language. We demonstrate this in Koka, a language with algebraic effects with static type inference, which compiles to JavaScript that can run fully asynchronous on either Node.js or the browser.
- The consistent type driven development leads naturally to powerful abstractions that are (currently) lacking in mainstream asynchronous platforms like Node.js and .NET, like *cancelable*, *timeout*, and *interleaved*. These are all non-trivial to implement and we spend a significant part of the paper to discuss them in detail (Section 3). It's our hope this work may lead to these abstractions being implemented on other platforms in the future.
- The implementation of asynchronous operations on top of real world platforms is also a validation of the expressiveness of Koka's effect types based on row-polymorphism (Leijen, 2017, 2016b, 2005). We extensively use rank-2 polymorphic types (Leijen, 2009, 2008) to safely encapsulate local mutable state inside effect handlers (Leijen, 2014; Launchbury and Sabry, 1997). This technique has not found wide-spread use before, but safe state encapsulation works surprisingly well in combination with algebraic effect handlers.

There is a full implementation of *async-await* as a library in Koka and all examples can be run in either the browser or on Node.js. See (Leijen, 2016a) for detailed instructions to download Koka and program with algebraic effects.

We start by giving a general overview of Koka and algebraic effects in Section 2. The main part of the paper is the description of the implementation of asynchronous effects in Section 3. Section 4 defines the formal semantics of our system and we finish with the conclusion in 5. There is no separate related work section – instead we try to reference and discuss related work at each topic inline.

2. Overview

In this section we give an overview of programming with algebraic effects. The interested reader may take a quick look ahead at

Figure 4 in Section 4.1 to see the precise operational semantics of algebraic effect handlers. For the sake of concreteness, we show all examples in the current Koka implementation but we stress that the techniques shown here apply generally and can be applied in many other languages.

Koka is a call-by-value programming language with type inference that tracks effects. The type of a function has the form $\tau \rightarrow\ \epsilon\ \ \tau'$ signifying a function that takes an argument of type τ, returns a result of type τ' and may have a *side effect* ϵ. Note that, unlike Haskell, there are *three* arguments to the function arrow and we should *not* parse $\epsilon\ \ \tau$ as the type application $\epsilon\langle\tau\rangle$! We can leave out the effect and write $\tau \rightarrow\ \ \tau'$ as a shorthand for the total function without any side effect: $\tau \rightarrow\ \ \langle\rangle\ \ \tau'$. A key observation on Moggi's early work on monads (Moggi, 1991) was that values and computations should be assigned a different type. Koka applies that principle where effect types only occur on function types; and any other (value) type, like *int*, truly designates an evaluated value that cannot have any effect.

Koka has many features found in languages like ML and Haskell, such as type inference, algebraic data types and pattern matching, higher-order functions, impredicative polymorphism, open data types, etc. A pioneering feature of Koka is the use of row types with scoped labels to track effects in the type system, striking a balance between conciseness and simplicity. The system works well in practice and has been used to write significant programs (Leijen, 2015). Recently, the effect system was extended with full algebraic effects and handlers (Leijen, 2017).

There are various ways to understand algebraic effects and handlers. As described originally (Plotkin and Power, 2003; Plotkin and Pretnar, 2013), the signature of the effect operations forms a free algebra which gives rise to a free monad (Awodey, 2006). Free monads provide a natural way to give semantics to effects, where handlers describe a *fold* over the algebra of operations (Swierstra, 2008; Kiselyov and Ishii, 2015; Wu and Schrijvers, 2015). The original work on algebraic effects gives a solid semantic foundation and works well for proofs and semantic exploration.

However, for working with algebraic effects as a programming construct, it is can be more intuitive to use a more operational perspective. It turns out we can view algebraic effects operationally as *resumable exceptions* (or perhaps as a more structured form of delimited continuations). We therefore start our overview by modeling exceptional control flow.

2.1. Exceptions as Algebraic Effects

Suppose we have a programming language (like Koka) that has algebraic effects, but no builtin notion of exception handling. In such language you can define exception handling yourself as a library – no need for special compiler support! The exception effect *exn* can be defined in Koka as:

```
effect exn {
  fun throw( s : string ) : a
}
```

This defines a new effect type *exn* with a single primitive operation, *throw* with type *string* → *exn a* for any *a* (Koka uses single letters for polymorphic type variables). The *throw* operation can be used just like any other function:

```
fun exn-div( x, y ) {
  if (y == 0) then throw("divide by zero") else x / y
}
```

Note that in Koka we can use identifiers with dashes (as in *exn-div*) and end identifiers with question marks (as in *done?*). Type inference will infer the type *exn-div* : (*int*,*int*) → *exn int* propagating our new exception effect. Up to this point we have introduced the new effect type and the operation interface, but we have not yet defined what these operations mean. The semantics of an operation is given through an algebraic effect *handler* which allows us to *discharge* the effect type. The standard way to discharge exceptions is by catching them, and we can write this using effect handlers as:

```
fun catch(action,h) {
  handle(action) {
    throw(s) → h(s)
  }
}
```

The handle construct for an effect takes an *action* to evaluate and a set of operation clauses. The inferred type of *catch* is:

catch : (*action* : () → ⟨*exn* | *e*⟩ *a*, *h* : *string* → *e a*) → *e a*

The type is polymorphic in the result type *a* and its final effects *e* , where the action argument can have the *exn* effect and possibly more effects *e*. As we can see, the handle construct discharged the *exn* effect and the final result effect is just *e*. For example,

```
fun zero-div(x,y) {
  catch( {
    exn-div(x,y)
  }, fun(s){ 0 } )
}
```

has type (*int*,*int*) → ⟨⟩ *int* and is a total function. Note that the Koka syntax { *exn-div*(*x*,*y*) } denotes an anonymous function that takes no arguments. Koka also allows trailing function arguments to be applied Haskell-style without parenthesis, and we can write the *catch* application more concisely as:

```
fun zero-div(x,y) {
  catch {
    exn-div(x,y)
  }
  fun(s){ 0 }
}
```

We use this syntax extensively in Section 3. Generally in Koka, expressions between parenthesis are eagerly evaluated while expressions between curly braces are generally part of a function body whose evaluation is delayed. These syntax conventions are very convenient for defining new control-flow abstractions.

Besides clauses for each operation, each handler can have a return clause too: this is applied to the final result of the handled action. In the previous example, we just passed the result unchanged, but in general we may want to apply some transformation. For example, transforming exceptional computations into *maybe* values:

```
fun to-maybe(action) {
  handle(action) {
    return x→ Just(x)
    throw(s)→ Nothing
}}
```

with the inferred type $(() \rightarrow \langle exn \mid e\rangle\ a) \rightarrow e\ maybe\langle a\rangle$.

The handle construct is actually syntactic sugar over the more primitive handler construct:

```
handle(action) { … }   ≡   (handler{ … })(action)
```

A handler just takes a set of operation clauses for an effect, and returns a function that discharges the effect over a given action. This allows us to express *to-maybe* more concisely as a (function) value:

```
val to-maybe = handler {
  return x→ Just(x)
  throw(s)→ Nothing
}
```

with the same type as before.

Just like monadic programming, algebraic effects allows us to conveniently program with exceptions without having to explicitly plumb *maybe* values around. When using monads though we have to provide a *Monad* instance with a *bind* and return, and we need to create a separate discharge function. In contrast, with algebraic effects we only define the *operation* interface and the discharge is implicit in the handler definition.

In Koka, we exceptions are implemented not just over strings but using an open *exception* type that can be extended with user defined constructors. The operations *try* and *untry* convert between explicit exceptions and the exception effect:

```
val try = handler {
  return x    → Right(x)
  throw(exn)→ Left(exn) }

fun untry(ex) {
  match(ex) {
    Left(exn)→ throw(exn)
    Right(x) → x
}}
```

where *try* has type $(() \rightarrow \langle exn \mid e\rangle\ a) \rightarrow e\ either\langle exception,a\rangle$. We will use these functions again when implementing asynchronous effects in Section 3

2.2. Resuming Operations

The exception effect is somewhat special as it never resumes: any instructions following the *throw* are never executed. Usually, operations will *resume* with a specific result. An example of a resumable effect is a *reader* effect, where we dynamically bind a readable value. This can for example be used in Node.js servers to expose the current request object as ambient state (Section 3.6). Here we illustrate this with an *input* effect:

```
effect input { getstr() : string }
```

where the operation *getstr* returns some input. We can use this as:

```
fun hello() {
  val name = getstr()
  println("hello " + name)
}
```

An obvious implementation of *getstr* gets the input from the user, but we can just as well create a handler that takes a set of strings to provide as input, or always returns the same string:

```
val always-there = handler {
  getstr()→ resume("there")
}
```

Every operation clause in a handler brings an identifier *resume* in scope which takes as an argument the result of the operation and resumes the program at the invocation of the operation – if the *resume* occurs at the tail position (as in our example) it is much like a regular function call. Executing *always-there*(*hello*) will output:

```
> always-there(hello)
hello there
```

The *resume* function is very powerful as it resumes the program at the operation's invocation site – in the implementation this entails saving the current execution context, including the stack up to the handler, such that it can be restored when invoking *resume*. The *resume* function is a first-class function and can be passed around, stored in data structures etc. Due to the structure of algebraic effects, we can generally implement this quite efficiently and optimize for common scenarios. For example, multi-core OCaml (Dolan et al., 2015) supports a very efficient *resume* implementation by restricting it to one-shot resumes only.

2.3. State

As another example of resuming, we can define a stateful effect:

```
effect state⟨s⟩ {
  get() : s
  put( x : s ) : ()
}
```

The *state* effect is polymorphic over the values *s* it stores. For example, in

```
fun counter() {
  val i = get()
  if (i ≤ 0) then () else {
    println("hi")
    put(i - 1)
    counter()
}}
```

the type becomes $() \rightarrow \langle state\langle int\rangle,console,div \mid e\rangle\ ()$ with the state *s* instantiated to *int*. To define the *state* effect we could use the built-in state effect of Koka, but a cleaner way is to use *parameterized* handlers. Such handlers take a parameter that is updated at every *resume*. Here is a possible definition for handling state:

```
val state = handler(s) {
  return x→ (x, s)
  get()     → resume(s, s)
  put(s')  → resume(s', ())
}
```

We see that the handler binds a parameter *s* (of the polymorphic type *s*), the current state. The return clause returns the final result tupled with the final state. The *resume* function in a parameterized handler takes now multiple arguments: the first argument is the handler parameter used *when handling the resumption*, while the last argument is the result of the operation. The *get* operation leaves the current state unchanged, while the *put* operation resumes with its passed-in state argument. Just like *resume*, the function returned by the parameterized handler also takes the initial state as an extra argument:

```
state : (x: s, action: () → ⟨state⟨s⟩ | e⟩ a) → e (a,s)
```

and we can use it as:

```
> state(2,counter)
hi
hi
```

2.4. Iterators

Many contemporary languages, like JavaScript or C#, have special syntax and compilation rules for iterators and the *yield* statement (The EcmaScript committee, 2015). Algebraic effects generalize over this where the *yield* effect can be defined as:

```
effect yield⟨a⟩ {
  yield( item : a ) : ()
}
```

This effect is polymorphic in the values *a* that are yielded. For example, we can define an "iterator" over lists as:

```
fun iterate( xs : list⟨a⟩ ) : yield⟨a⟩ () {
  match(xs) {
    Nil        → ()
    Cons(x,xx)→ { yield(x); iterate(xx) }
}}
```

and similarly for many data structures. Orthogonal to the iterators, we can define handlers that handle the yielded elements. For example, here is a generic *foreach* function that applies a function f to each element that is yielded and breaks the iteration when f returns *False*:

```
fun foreach(f : a → e bool, act : () → ⟨yield⟨a⟩ | e⟩ ()): e () {
  handle(action) {
    return x→ ()
    yield(x) → if (f(x)) then resume(()) else ()
}}
```

Note how we can stop the iteration simply by not calling *resume* – and that we can define this behavior orthogonal to the definition of any particular iterator.

2.5. Multiple Resumptions

> You can enter a room once, yet leave it twice.
> — Peter Landin (1965, 1998)

In the previous examples we looked at abstractions that never resume (e.g. exceptions), and abstractions that resume once (e.g. reading and state). Such abstractions are common in most programming languages. Less common are abstractions that can resume more than once. Examples of this behavior can usually only be found in languages like Lisp and Scheme, that implement some variant of *callcc* (Thielecke, 1999). A nice example to illustrate multiple resumptions is the ambiguity effect:

```
effect amb {
  flip() : bool
}
```

where we have a *flip* operation that returns a boolean. As an example, we take the exclusive or of two flip operations:

```
fun xor() : amb bool {
  val p = flip()
  val q = flip()
  ((p || q) && !(p && q))
}
```

There are many ways we may assign semantics to *flip*. One handler just flips randomly:

```
val coinflip = handler {
  flip() → resume(random-bool())
}
```

with type $(action: () \to \langle amb, ndet \mid e\rangle\ a) \to \langle ndet \mid e\rangle\ a$ where *random-bool* induced the (built-in) non-deterministic effect *ndet*. A more interesting implementation though is to return *all* possible results, resuming twice for each *flip*: once with a *False* result, and once with a *True* result:

```
val amb = handler {
  return x→ [x]
  flip()   → resume(False) + resume(True)
}
```

with type $amb : (action : () \to \langle amb \mid e\rangle\ a) \to e\ list\langle a\rangle$, discharging the *amb* effect and lifting the result type *a* to a $list\langle a\rangle$ of all possible results. The return clause wraps the final result of the action in a list, while in the *flip* clause we append the results of both resumptions (using +). Since each resume is handled by the same handler, the results of each resumption will indeed be of type $list\langle a\rangle$. For example, executing *amb*(*xor*) leads to:

```
> amb(xor)
[False,True,True,False]
```

Multiple resumptions should be used with care though as the composition with other effects can sometimes be surprising. As an example, consider a program that uses both *state* and ambiguity:

```
fun surprising() : ⟨state⟨int⟩,amb⟩ bool {
  val p = flip()
  val i = get()
  put(i + 1)
  if (i ≥ 1 && p) then xor() else False
}
```

We can use our earlier handlers to handle the state and ambiguity effects, but we can compose them in two ways, giving rise to two different semantics. First, we can handle the state outside the ambiguity handler, giving rise to a "global" state that is shared between each ambiguous assumption.

```
> state(0, { amb(surprising) })
([False,False,True,True,False],2)
```

The final result is a tuple of a list of booleans and the final state. Since the state is shared, only the first time ($i \geq 1$ && p) is evaluated the result will be *False* (the first element of the result list). On the second resumption, *xor*() will be evaluated leading to the other 4 elements.

If we change the order of the handlers, we effectively make the state "local" to each ambiguous resumption:

```
> amb( { state(0,surprising) } )
[(False,1),(False,1)]
```

and the result is now a list of tuples and in both resumptions of the first *flip* the *i* will be the initial state leading to two *False* elements in the result list.

Note that, in contrast to general monads, algebraic effects can be composed freely (since they are restricted to the free monad). This is quite an improvement over previous work (Swamy et al., 2011; Vazou and Leijen, 2016) where composing different monads required implementing a combined monad by hand.

Generally we need to program careful with effects that can resume more than once since, as shown, those can interact in unusual ways with stateful computations – and similarly for never resuming effects like exceptions. Nevertheless, it turns out that both resuming multiple times and never resuming are natural for many effects, like backtracking, and parsers (Wu et al., 2014; Leijen, 2017), and are also needed to implement asynchronous primitives as we will see in the next section.

3. Asynchronous Programming

Similarly to iterators, many programming languages are adding support for async-await style asynchronous programming (The EcmaScript committee, 2016). For example, web servers written in JavaScript using Node.js are highly asynchronous and without language support the resulting programs are difficult to write and debug due to excessive callbacks (i.e. the so-called "pyramid of doom"). Extending a language with *async-await* is non-trivial though, both in terms of semantics, as well as compilation complexity where async methods need to be translated into state-machines to simulate co-routine behavior (Bierman et al., 2012). The interaction with iterators and exceptions is also complex and not always well understood. In this section we look at implementing asynchronous abstraction using just algebraic effects.

3.1. An Asynchronous Effect

We begin by defining a type alias *result⟨a⟩* type that captures that an asynchronous operation may either return a result or an exception:

```
alias result⟨a⟩ = either⟨exception,a⟩
```

The asynchronous effect just consists of a single *await* operation:

```
effect async {
  fun await( initiate : (result⟨a⟩ → io ()) → io ()) : result⟨a⟩
}
```

The *await* operation takes a single argument *initiate* that initiates a primitive asynchronous operation. The *initiate* function gets as an argument itself a callback function of type *result⟨a⟩ → io ()* which takes the results value that will be returned from *await*.

Usually we immediately translate a *result⟨a⟩* into a thrown exception or a plain value. The $await_1$ abstraction does just that:

```
fun await1( initiate : (a → io ()) → io ()) : ⟨async,exn⟩ a {
  untry( await( fun(cb){
    initiate( fun(x){ cb(Right(x)) } )
  }))
}
```

We can also define $await_0$ which is convenient for operations that return a unit result:

```
fun await0( initiate : (() → io ()) → io ()) : ⟨async,exn⟩ () {
  await1( fun(cb) { initiate( { cb(()) } )})
}
```

Using our new *await* operation, it becomes easy to expose primitive asynchronous operations in the *async* effect. For example, we can define a *wait* function that waits for a specified duration:

```
fun wait( secs : duration ) : ⟨async,exn⟩ () {
  await0 fun(cb) {
    set-timeout( cb, secs.milli-seconds.int32 )
    ()
}}
external set-timeout( cb : () → io (), ms : int32 ) : io timeout-id {
  js "setTimeout(#1,#2)"
}
```

The external declaration is part of the foreign function interface of Koka – here we call the JavaScript *setTimeout* function with the given callback *cb* and duration *ms* in milliseconds. What is of essence here is just that our new *async* effect declaration gives us an *await* operation that allows us to capture the execution context, and pass the continuation as a first-class callback function *cb* to the primitive asynchronous operations of the host platform – and this can all be done without special compiler support for asynchrony!

We can now use our new *wait* function as any other function:

```
fun hello-world() : ⟨async,exn,console⟩ () {
  println("hello")
  wait(2.seconds)
  println("world")
}
```

Et voilà – a true asynchronous program build on top of plain algebraic effects. But of course, we need to still define an actual handler for *async*!

3.2. Implementing an asynchronous handler

The implementation of the *async* handler is surprisingly straightforward – we simply pass the *resume* function directly as the actual callback to *initiate*:

```
val async-handle = handler {
  await(initiate) → initiate( resume )
}
```

How beautifully concise! Moreover, it corresponds *exactly* to the algebraic definition of *shift* for delimited continuations (as shown in Section 4.2) – just instantiated to a particular type instead of being fully generic:

```
async-handle : (() → ⟨async,io⟩ ()) → ⟨async,io⟩ ()
```

Of course, the *async-handle* function is meant to be used on the most outer level of the program (i.e. around *main*) since after *initiate* the host platform expects a program to exit *main* and return to the host event loop which will call the registered callbacks when primitive asynchronous operations are completed. This is the case for all major asynchronous environments, in particular the browser, Node.js and the .NET environment. Nevertheless, since *async* is just a regular effect, we can always declare other handlers: for example to *mock* certain functionality or to use a special event loop.

Actually, another point in the design space that we explored is to describe all available asynchronous operations as a generalized algebraic data type (GADT) (Johann and Ghani, 2008). This way, an *async* handler can introspect all asynchronous requests and choose various ways to implement them. This would allow for example a testing framework with various interleaving strategies. The main drawback of using the GADT approach is that it does not extend well: we need to define the entire asynchronous API in the request type. For that reason we currently do not use this approach.

3.3. Interleaving

Even though its type is sound, the basic *await* operation is perhaps a bit too powerful as it allows embedding any *io* operation in the *async* effect. As such we envision the use of *await* mostly for library writers to encapsulate primitive asynchronous operations.

As an example of the power of *await*, we can write a function that exits the program without ever returning:

```
fun exit() : ⟨async,exn⟩ a {
  await( fun(cb) { } ).untry
}
```

It does this by simply ignoring the callback *cb*. This seems a rather useless function but as we see later, it is essential to implement more higher-level primitives.

Dually, we can also implement operations that return multiple times from *await*. This is used to implement primitive forking.

```
fun fork() : ⟨async,exn,ndet⟩ bool {
  await1 fun(cb) {
    set-timeout({cb(True)}, 0.int32)
    cb(False)
  }
}
```

The *fork* function returns twice: first with *False*, and later with *True* (using *set-timeout*). Note how we immediately return with *False* by calling the callback directly – which in turn calls *resume* and resumes at the point where *fork* was called. Often, the effects *async* , *exn*, and *ndet* occur together so we define a convenient type alias:

```
alias asyncx = ⟨async,exn,ndet⟩
```

Using the new *fork* and *exit* operations, we are now in the position to define *interleaved*:

```
fun interleaved(a : () → ⟨asyncx | e⟩ a,
                b : () → ⟨asyncx | e⟩ b) : ⟨asyncx | e⟩ (a,b) {
  val (ar,br) = interleavedx(a,b)
  (ar.untry,br.untry)
}
```

The function interleaves two actions *a* and *b* and is defined over the *interleavedx* function which returns a *result* for each component:

```
fun interleavedx(a : () → ⟨asyncx | e⟩ a,
                 b : () → ⟨asyncx | e⟩ b ) : ⟨asyncx | e⟩ (result⟨a⟩,result⟨b⟩)
{
  handle-shared {
    var ares := Nothing
    var bres := Nothing
    if (fork()) {
      val br = try( inject-st(b) )
      bres := Just(br)
      match(ares) { Nothing → exit()
                    Just(ar)→ (ar,br)
    }}
    else {
      val ar = try( inject-st(a) )
      ares := Just(ar)
      match(bres) {Nothing → exit()
                   Just(br)→ (ar,br)
}}}}
```

For now we ignore the *handle-shared* function which is discussed later. The function starts by declaring to mutable variables *ares* and *bres* that will store the result of either action. The *fork* function will return twice – once with *True* and once with *False* – and depending on its result we execute either action *a* or *b*. The actions are executed under a *try* operation that catches any exceptions and wraps the result in an *either* type (see Section 2.1). We then store the result in our mutable variable and then match on the mutable variable of the other action: if it is *Nothing* that action did not complete yet and we use *exit*() to exit from our asynchronous strand. Otherwise, both actions did complete and we return a tuple of the results.

3.3.1. Safe Encapsulation of State

The reader may wonder why the stateful mutation is not reflected in the effect type of *interleavedx*. The Koka type system does infer that the body of the function has a stateful effect. In particular, the *ares* variable has type *ref⟨h,maybe⟨result⟨a⟩⟩⟩* for some heap *h*. Assigning and reading from such variable leads to a stateful effect *st⟨h⟩*. When generalizing over the body of the function though, it can be determined that *h* can be generalized and does not escape its scope. As such, the effect *st⟨h⟩* is not observable from the outside and can be safely discarded. This mechanism is proven sound by Leijen (2014) and is similar to the safe encapsulation of state using *runST* in Haskell (Launchbury and Sabry, 1997).

This is also the reason for the use of *inject-st* which injects the *st⟨h⟩* effect into a function effect *e*:

```
inject-st : (() → e a) → total (() → ⟨st⟨h⟩ | e⟩ a)
```

If we would not have applied this function over the actions *a* and *b*, the effects of those functions would have directly unified with the ambient effect which contains *st⟨h⟩*; in that case the type *h* would escape into the types of *a* and *b* preventing Koka from discarding the *st⟨h⟩* effect at generalization time. By injecting an *st ⟨h⟩* effect manually into the types of *a* and *b*, we prevent this from happening.

3.3.2. Semantics of interleaving

The current definition of *interleaved* may not yet quite be what we expect. In particular, since *async-handle* is the outermost handler of effects any effect handlers under it are 'isolated' per asynchronous strand – just like our previous example of a state handler under an ambiguous effect (Section 2.3). For example, suppose we define the following 'append' state effect:

```
effect astate { append( s : string ) : () }

val astate-handle = handler(acc = "") {
  return x→ (x,acc)
  append(s) → resume(acc + s + " ", ())
}
```

and use that state inside different interleaved actions:

```
val (_,st) = astate-handle {
  interleaved
    { wait(1.seconds); append("1") }
    { wait(2.seconds); append("2") }
}
println("final state: " + st)
```

Unfortunately, the final state is not "1 2" but rather just "2". This is because the *async* handler is the outermost handler, and each asynchronous strand gets its own isolated copy of the append state. One way around this is to use the builtin state effect *st⟨h⟩* since the builtin effects are handled even outside *async*. However, generally we would like the user to be able to define various stateful effects that *are* shared between the various asynchronous strands. For example, when defining Express Node.js servers, one typically threads an explicit request object *req* to all functions – it would be much nicer to define a *request* effect instead that gives access to the current request without plumbing around an explicit object everywhere.

3.3.3. Sharing the Handler Context

It turns out that in combination with mutable state we can redefine the sharing of the handler stack using a regular effect handler! Here is the definition of *handle-shared*[1]:

```
fun handle-shared(action : () → ⟨async,exn | e⟩ a ) : ⟨async,exn | e⟩ a
{
  var latest := fun(_) { () }
  handle(inject-st(action)) {
    await(initiate) → {
      val r : either⟨exception,() → ⟨async | e⟩ a⟩
        = await fun(cb) {
          latest := cb
          initiate( fun(x : either⟨exception,a⟩ ) {
            latest( Right( { resume(x) } ) )
          })
        }
      match(r.fst) {   Right(f) → f()
                       Left(exn) → resume( Left(exn) )
}}}}
```

In order to share we are going to share part of the callback function among the different strands. The handler captures all *await* operations. It immediate calls the *await* itself but with a modified *initiate* function: the final callback *cb* that is passed is stored in the local variable *latest*. The original *initiate* function is now called but with a modified callback: it calls the *latest* callback (instead of *cb*) with a result function that calls the local *resume* function.

The *cb* (and thus *latest*) functions will always return exactly to the *await* in our handler (as shown by the arrow). Even though all callbacks use *latest* to return to the handler, the result *r* contains the anonymous function that calls the *resume* that returns to the *original* asynchronous strand. This is exactly the behavior we want: all the encapsulated asynchronous strands share the latest callback from our handler, but below that each strand uses a regular *resume*.

The local state mutation of *latest* can again be safely hidden because it is not observable from outside; we need to use *inject-st* again on the *action* to prevent the local heap parameter from escaping into the type of the *action*. Rerunning our previous example with a *handle-shared* handler in the definition of *interleavedx* gives now the expected final state of "1 2".

[1]We assume scoped type variables here to concisely annotate the binders but Koka does currently not support this feature and you need to use the some quantifier for those.

3.4. Cancellation

A very important building block for further abstraction is cancellation. Our main primitive is the *cancelable* handler:

```
fun cancelable( action : () → ⟨asyncx | e⟩ a) : asyncx a
```

and a new operation *cancel* that is added to the *async* effect:

```
effect async {
  fun await( initiate : (result⟨a⟩ → io ()) → io ()) : result⟨a⟩
  fun cancel() : ()
}
```

The *cancel* operation cancels any outstanding asynchronous operation under its enclosing *cancelable* handler. Canceling an asynchronous operation results in the *Cancel* exception. Implementing *cancelable* and *cancel* is a bit involved and we delay describing it until Section 3.4.2.

Using *cancelable* we can build an interleaved function *firstof* that returns the result of the first action that completes:

```
fun firstof(a : () → ⟨asyncx | e⟩ a,
            b : () → ⟨asyncx | e⟩ a ) : ⟨asyncx | e⟩ a {
  val (ra,rb) = cancelable {
    interleavedx
      { val x = a(); cancel(); x }
      { val y = b(); cancel(); y }
  }
  (if (ra.canceled?) then rb else ra).untry
}
```

where *canceled?* is defined as:

```
fun canceled?( x : either⟨exception,a⟩ ): bool {
  match(x) {
    Left(exn)→ exn.info.cancel?
    Right    → False
}}
```

We use our *interleavedx* abstraction to execute the two actions. This is done under a *cancelable* handler. Each interleaved action now calls *cancel* upon completion which causes the other action to throw a *Cancel* exception for any outstanding asynchronous operations (on the next tick). Once both are finished, either with a *Cancel* exception or normally, the first result that was not a *Cancel* exception is returned. Note that if both actions are canceled (by a *cancel* under a *cancelable* higher up), the function itself re-throws that *Cancel* exception as expected. Also, it is important that the operation *cancel itself* does not throw a *Cancel* exception – as shown here, we actually continue after *cancel* with a valid result.

The *firstof* is useful by itself, for example for issuing a download request to multiple servers concurrently and using the first request that completes. Our main use for *firstof* though is to create an even more interesting abstraction, namely a block scoped *timeout* operation:

```
fun timeout(secs : duration,
            action : () → ⟨asynx | e⟩ a ) : ⟨asynx | e⟩ maybe⟨a⟩ {
  firstof
    { wait(secs); Nothing }
    { Just(action()) }
}
```

This is a general *timeout* function that executes *action* but if it is not completed within *secs* duration, it cancels it and returns *Nothing* instead. This is a powerful abstraction as it is not tied to a particular operation but instead *block scoped over any composition of asynchronous operations*. For example, frameworks like Node.js or .NET usually provide a timeout on some particular operations, like a download request, but for any composition of operations you need to implement custom solutions – usually checking flags everywhere. Such custom solutions are usually not very robust and since cancellation is not well supported it is very hard to provide the kind of robustness and performance that is provided by the *timeout* function as shown here.

3.4.1. Releasing Resources

There is still a problem with the *wait* implementation though: even though it will be canceled when the *action* completes first, it will still hold on to its registered callback in *set-timeout*; when the primitive timeout expires, this callback will be called and immediately terminate (because it was canceled) but that is still a resource leak. In general, some primitive operations need to release their resources when canceled. We re-implement *wait* to clear its timeout on cancellation:

```
fun wait( secs : duration ) : asyncx () {
  var vtid := Nothing
  on-cancel {
    match(vtid) {
      Nothing → ()
      Just(tid) → clear-timeout(tid)
    }
  }
  { await0 fun(cb) {
      vtid := set-timeout(cb,secs.milli-seconds.int32)
    }
}}
```

where the first argument of *on-cancel* is run whenever a *Cancel* exception is raised in its second argument:

```
fun on-cancel( caction, action ) {
  catch(action) fun(exn) {
    if (exn.info.cancel?) then caction()
    throw(exn)// re-throw
}}
```

In *wait* we now use a mutable variable to keep the *timeout-id* of the timeout. Using *on-cancel* we clear the timeout if a *Cancel* exception was raised. Unfortunately, the above definition does not type check! The assignment *vtid* := ... happens as part of the *io* typed *initiate* and the stateful effect *st*⟨*h*⟩ will unify with the global *io* state effect (*st*⟨*global*⟩).

To make this pass the type checker we need to do the assignment locally and lift it outside the *io* action. We can use a similar trick as with *fork* where we return twice: once directly with the registered *timeout-id* and once with *Nothing* when the timeout triggers:

```
fun wait( secs : duration ) : asyncx () {
  var vtid := Nothing
  on-cancel {
    match(vtid) {
      Nothing → ()
      Just(tid)→ clear-timeout(tid)
    }
  }
  { val mbtid = await1 fun(cb) {
      val tid = set-timeout({cb(Nothing)},secs.milli-seconds.int32 )
      cb(Just(tid))
    }
    match(mbtid) {
      Just(tid) → { vtid := Just(tid); exit() }
      Nothing→ ()
}}}
```

3.4.2. Implementing Cancellation

To implement cancellation we need to add more bookkeeping to our current implementation and keep track of all outstanding asynchronous requests to be able to cancel them. Since we cannot directly compare callback functions for equality we are going to assign each callback a unique *waiting identifier* (*wid*) that can only be compared for equality.

```
abstract struct wid( id : int )
fun (==)( wid1 : wid, wid2 : wid ) { wid1.id == wid2.id }
```

Moreover, we need to know when we can remove a callback from the outstanding requests – since certain callbacks may resume more than once, we require that callbacks return whether this is their final resumption or not. The new *result* alias becomes a triple now:

```
alias result⟨a⟩ = (either⟨exception,a⟩, bool, wid)
```

and includes a boolean that flags whether this is the last resumption and the waiting identifier of the callback. Moreover, the *await* operation needs to get the assigned identifier of the callback too, and we need a way to create an initial unique callback identifier. We are going to replace the *await* operation in the *async* effect with two new operations to do this:

```
effect async {
  fun await-on( initiate : result⟨a⟩ → io (), wid : wid ) : result⟨a⟩
  fun await-id() : wid
  fun cancel( ids : list⟨wid⟩ = [] ) : ()
}
```

The *await* function now abstracts over *await-on* and *await-id* to hide the internal bookkeeping of the waiting identifiers:

```
fun await(initiate : (either⟨exception,a⟩,bool) → io () )
              : async either⟨exception,a⟩ {
  val wid = await-id()
  val res = await-on( fun(r) { initiate(r.fst,r.snd) }, wid )
  res.fst
}
```

The outer *async* handler keeps track of all outstanding request in a *awaits* list that maps waiting identifiers to callbacks:

```
fun async-handle(action : () → ⟨async,io⟩ () ) : io ()
{
  var awaits := []
  fun callback( resume, wid : wid) { ... }
  fun cancel-awaits( wids : list⟨wid⟩ ) { ... }

  handle(action) {
    await-id() → resume(Wid(unique()))
    await-on( initiate, wid ) → initiate( callback(resume,wid) )
    cancel( wids ) → resume(wids || awaits.map(fst))
  }
}
```

The *await-id* handler simply returns a unique identifier. The *await-on* handler now uses *callback* to create a wrapper callback around *resume* that checks for cancellation:

```
fun callback( resume : result⟨a⟩ → io (), wid : wid ) {
  fun cb(res) {
    val (_, done?, wid1 : wid) = res
    if (awaits.contains(wid1)) {
      if (done?) then awaits := awaits.remove(wid1)
      resume(res)
    }
  }
  if (wid != wid-exit) awaits := Cons((wid,cb),awaits)
  cb
}
```

The new callback *cb* first checks if the waiting id is still in the outstanding *awaits* list: this will only be the case if this was not yet canceled or already returned with *done?* being *True*. This is an essential check: even if an operation is canceled it may still happen that a callback is called later on if the resources were not properly released. For example, if we would not call *clear-timeout* on a registered timeout handler. The check in *callback* prevents resuming in such case and ensures cancellation will work over all operations whether they are cancellation aware or not.

After the check, the new callback removes itself from the outstanding *awaits* list if it is the last resumption, i.e. when *done?* is *True*, and finally resume with the result. We add the new callback *cb* to the *awaits* list. There is a check here for the special *wid-exit* id; this identifier is used by the *exit* operation that never resumes – just for that particular case we don't want to store the callback in the *awaits* list at all.

Finally, the handler for *cancel* calls *cancel-awaits* to cancel all supplied waiting identifiers by directly invoking their callback with a *Cancel* exception:

```
fun cancel-awaits( wids : list⟨wid⟩ ) {
  wids.foreach fun(wid) {
    match( awaits.lookup(wid) ) {
      Nothing→ ()
      Just(cb) → cb((Left(cancel-exn),True,wid))
}}}
```

Note that the invocation of *cb* will also remove the callback from the *awaits* list as it will have been created by the *callback* function.

3.4.3. Implementing Cancelable

A *cancelable* handler is now straightforward to construct since we only have to keep track of the waiting identifiers of the asynchronous requests in our scope:

```
fun cancelable( action : () → ⟨async | e⟩ a ) : ⟨async | e⟩ a {
  var awaits := []
  var canceled? := False;
  handle(inject-st(action)) {
    await-on(initiate,wid) → {
      if (canceled?) then resume((Left(cancel-exn),True,wid))
      else {
        if (wid != wid-exit) awaits := Cons(wid,awaits)
        val res = await(initiate,wid)
        if (res.snd) awaits := awaits.remove( fun(i) { i == res.thd })
        resume(res)
      }
    }
    cancel(wids) → {
      canceled? := True
      if (wids.nil? && awaits.nil?)
        then resume(())
        else resume(cancel(wids || awaits))
    }
    await-id() → resume(await-id())
  }
}
```

The handler intercepts all *await-on* operations and maintains its own local list of outstanding request. If *cancel* is invoked with an empty list, the handler calls the outer *cancel* with its own *awaits* list canceling all outstanding requests in its scope. A canceled strand could still catch the *Cancel* exception and perform further asynchronous operations. The *canceled?* flag is set to to *True* on cancellation and checked on each *await*: if the strand is already canceled.

3.5. Promises

There is an important difference in how the *async* effect works using algebraic effects versus how languages like JavaScript and C# handle asynchrony: in the latter, asynchronous operations always return a *Promise* (or *Task*) which is then awaited on. When we compose some *async* methods it will create a new promise that we need to await separately, i.e. we get nested call chains of *async*-*await*. A common problem with promises is 'losing' exceptions or forgetting to await a promise (Rauschmayer, 2016, ch. 25). In contrast, with algebraic effects we just have certain functions with an *async* effect and there is no intermediate promise object – just as all other effects the asynchrony is purely lexically scoped.

However, we still need to add the concept of a promise to our framework too because not all dataflow in a program is lexically scoped. For example, we may want to cancel a computation when the user presses a button but *cancel* can only cancel asynchronous operations up to its enclosing *cancelable* handler. Similarly, we may want initiate asynchronous operations but only await them in another part of the program.

We implement a first-class promise as an abstract type that carries a mutable reference to either a list of *await*ers, or its final resolved value.

```
abstract struct promise⟨a⟩(
  state : ref⟨global,either⟨list⟨(a → io ())⟩,a⟩⟩
)
```

When we *await* a promise, we either add ourselves to the awaiters or immediately return if the promise was already resolved:

```
public fun await( p : promise⟨a⟩ ) : ⟨async,exn,ndet⟩ a {
  await1 fun(cb) {
    match ( ! p.state) {
      Left(listeners) → p.state := Left(Cons(cb,listeners))
      Right(value)→ cb(value)
}}}
```

Resolving a promise updates the promise value and invokes all the current awaiters.

```
fun resolve( p : promise⟨a⟩, value : a ) : ⟨async,exn⟩ () {
  await fun(cb) {
    match( ! p.state) {
      Left(listeners) → {
        p.state := Right(value)
        listeners.foreach fun(cbawait) { cbawait(value) }
        cb(Right(()))
      }
      Right → cb(Left(exception("promise was already resolved.",Error)))
}}}
```

The new promise abstraction lets us communicate values across separate computations, for example:

```
val p = promise()
interleaved {
  println("what is your name?")
  p.resolve( readline() )
}
{ println("your name was: " + p.await ) }
```

Note that our new abstraction differs from .NET *Task*s. When the action of a *Task* throws an exception, it is re-thrown at the *await* . This is because the action of a *Task* executes outside its lexical context and cannot use the enclosing exception handlers (Leijen et al., 2009). We can mimic this behavior using a promise of type *promise⟨either⟨exception,a⟩⟩* returning either an exception or successful value when awaited.

3.6. Ambient Programming

By adhering to the typed discipline of algebraic effects we maintain a logical *strand* of execution – all operations are defined within their lexical scope. This naturally leads to *ambient programming*: a style of programming where we can declare variables and functions that *ambient*, i.e. dynamically bound to your current strand of execution and are neither local- nor global. The closest we have in other languages are exception handlers, where the exception that a function throws is handled by its nearest enclosing handler.

For example, in Node.js the Express framework (Brown, 2014) is used to write server programs where each request is handled asynchronously (i.e. interleaved with all other outstanding requests). Each strand of execution now needs access to the current request object (*req*) which needs to be passed around manually through every function call. With algebraic effects, we just declare a handler that returns 'the current' request object – which now becomes an *ambient* read-only variable:

```
effect req {get-request() : request }


fun with-request( req : request, action : () → ⟨req | e⟩ a ) : e a {
  handle(action) {
    get-request() → resume(req)
}}
```

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & e(e) & \text{application} \\
 & & \mid & \mathsf{val}\ x\ =\ e;\ e & \text{binding} \\
 & & \mid & \mathsf{handle}\{h\}(e) & \text{handler} \\
 & & \mid & v & \text{value} \\
\\
\text{Values} & v & ::= & x \mid c \mid op \ \mid \lambda x.\, e \\
\\
\text{Clauses} & h & ::= & \mathsf{return}\ x \ \rightarrow\ e \\
 & & \mid & op(x)\ \rightarrow\ e;\ \ h & op \notin h \\
\\
\text{Types} & \tau^k & ::= & \alpha^k & \text{type variable} \\
 & & \mid & c^{k_0}\langle \tau_1^{k_1}, \ldots, \tau_n^{k_n} \rangle & k_0\ =\ (k_1, \ldots, k_n)\ \rightarrow\ k \\
\\
\text{Kinds} & k & ::= & * \ \mid\ \mathsf{e} & \text{values, effects} \\
 & & \mid & \mathsf{k} & \text{effect constants} \\
 & & \mid & (k_1, \ldots, k_n)\ \rightarrow\ k & \text{type constructor} \\
\\
\text{Type scheme} & \sigma & ::= & \forall \alpha^k.\, \sigma \ \mid\ \tau^*
\end{array}
$$

$$
\begin{array}{llll}
\text{Constants} & (),\ \ bool & ::\ \ * & \text{unit, booleans} \\
 & (\_\ \rightarrow\ \_\ \_) & ::\ \ (*, \mathsf{e}, *)\ \rightarrow\ * & \text{functions} \\
 & \langle\rangle & ::\ \ \mathsf{e} & \text{empty effect} \\
 & \langle \_\ \mid\ \_ \rangle & ::\ \ (\mathsf{k}, \mathsf{e})\ \rightarrow\ \mathsf{e} & \text{effect extension}
\end{array}
$$

$$
\begin{array}{llll}
\text{Total functions} & \tau_1\ \rightarrow\ \tau_2 & \doteq & \tau_1\ \rightarrow\ \langle\rangle\ \ \tau_2 \\
\text{Effects} & \epsilon & \doteq & \tau^{\mathsf{e}} \\
\text{Effect variables} & \mu & \doteq & \alpha^{\mathsf{e}} \\
\text{Effect labels} & l & \doteq & c^k\langle \tau_1, \ldots, \tau_n \rangle \quad k\ =\ \ldots\ \rightarrow\ \mathsf{k} \\
\text{Closed effects} & \langle l_1, \ldots, l_n \rangle & \doteq & \langle l_1, \ldots, l_n\ \mid\ \langle\rangle\ \rangle \\
\text{Effect extension} & \langle l_1, \ldots, l_n \mid \epsilon \rangle & \doteq & \langle l_1\ \mid\ \ldots\ \langle l_n\ \mid\ \epsilon \rangle\ \ldots\ \rangle
\end{array}
$$

Fig. 1. Syntax of expressions, types, and kinds

This is much like *implicit parameters* (Lewis et al., 2000; Odersky, 2016). The approach here is more expressive though since we can define general operations besides just passing a value. For example, in Express, we also need to pass around the *response* object explicitly. With algebraic effects we can declare a *response* effect with various operations like *set-status-code* to manipulate the final server response as ambient mutable state.

Having all operations lexically scoped, and having a clear notion of the current strand of execution is perhaps the most important contribution that algebraic effects bring when implementing *async-await*. In other languages where *async-await* is based on promises (or *Task*s) there is no clear notion of the current strand of execution and a loss of lexically scoped operations. This leads to many problems in practice: debuggers have trouble showing the current variables in scope or 'call stack', profilers have trouble attributing resource usage to a particular strand of execution, in C# one needs to manually pass around cancellation tokens to enable cancelable operations, in Node.js there is the soft-deprecated *domains* abstraction to capture 'lost' exceptions etc, etc.

4. Semantics

In this section we give a formal definition of our polymorphic row-based effect system for the core calculus of Koka. The calculus and its type system has been in use for many years now and has been developed from the start using effect types based on rows with scoped labels (Leijen, 2005). Originally, user-defined effects were described using a monadic approach (Vazou and Leijen, 2016) but it turns out that algebraic effects fit the original type system well with almost no changes. Row based effect types are also used by Links (Lindley and Cheney, 2012) and Frank (Lindley et al., 2017), while the Eff language uses subtype constraints instead (Pretnar, 2014).

$$
\epsilon \cong \epsilon \quad [\textsc{eq-refl}] \qquad \frac{\epsilon_1 \cong \epsilon_2 \quad \epsilon_2 \cong \epsilon_3}{\epsilon_1 \cong \epsilon_3}\ [\textsc{eq-trans}]
$$

$$
\frac{\epsilon_1 \cong \epsilon_2}{\langle l \mid \epsilon_1 \rangle \cong \langle l \mid \epsilon_2 \rangle}\ [\textsc{eq-head}] \qquad \frac{l_1 \not\cong l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon \rangle\ \rangle \cong \langle l_2 \mid \langle l_1 \mid \epsilon \rangle\ \rangle}\ [\textsc{eq-swap}]
$$

$$
\frac{c\ \neq\ c'}{c\langle \tau_1, \ldots, \tau_n \rangle \not\cong c'\langle \tau_1', \ldots, \tau_n' \rangle}\ [\textsc{uneq-label}]
$$

Fig. 2. Row equivalence

Figure 1 defines the syntax of types and expressions. The expression grammar is straightforward but we distinguish values v from expressions e that can have effects. Values consist of variables x, constants c, operations op, and lambda's. Expression include handler expressions $\mathsf{handle}\{h\}(e)$ where h is a set of operation clauses. The handler construct of the previous section can be seen as syntactic sugar, where:

$$\mathsf{handler}\{h\} \equiv \lambda f.\ \mathsf{handle}\{h\}(f())$$

For simplicity we assume that all operations take just one argument. We also use membership notation $op(x)\ \rightarrow\ e\ \ \in h$ to denote that h contains a particular operation clause. Sometimes we shorten this to $op \in h$.

Well-formed types are guaranteed through kinds k which we denote using a superscript, as in τ^k. We have the usual kinds for value types $*$ and type constructors $\rightarrow$, but because we use a row based effect system, we also have kinds for effect rows ϵ, and effect constants (or effect labels) k. When the kind of a type is immediately apparent or not relevant, we usually leave it out. For clarity, we use α for regular type variables, and μ for effect type variables. Similarly, we use ϵ for effect row types, and l for effect constants/labels.

Effect types are defined as a row of effect labels l. Such row is either empty $\langle\rangle$, a polymorphic effect variable μ, or an extension of an effect ϵ with a label l, written as $\langle l\ \mid\ \epsilon \rangle$. Effect labels must start with a constant and are never polymorphic. By construction, effect type are either a *closed effect* of the form $\langle l_1, \ldots, l_n \rangle$, or an *open effect* of the form $\langle l_1, \ldots, l_n\ \mid\ \mu \rangle$.

We cannot use direct equality on types since we would like to regard effect rows equivalent up to the order of their effect constants. Figure 2 defines an equivalence relation ($\cong$) between effect rows. This relation is essentially the same as for the *scoped labels* record system (Leijen, 2005) with the difference that we ignore the type arguments when comparing labels. By reusing the *scoped labels* approach, we also get a deterministic and terminating unification algorithm which is essential for type inference. Moreover, in contrast to other record calculi (Rémy, 1994; Lindley and Cheney, 2012; Gaster and Jones, 1996; Sulzmann, 1997), our approach does not require extra constraints, like *lacks* or *absence* constraints, on the types which simplifies the type system significantly. The system also allows *duplicate* labels, where an effect $\langle exc, exc \rangle$ is legal and

$$\frac{\Gamma(x) \;=\; \sigma}{\Gamma \;\vdash\; x \;:\; \sigma \mid \epsilon}\;[\textsc{Var}] \qquad \frac{\Gamma \;\vdash\; e_1 \;:\; \sigma \mid \epsilon \quad \Gamma, x : \sigma \;\vdash\; e_2 \;:\; \tau \mid \epsilon}{\Gamma \;\vdash\; \mathsf{val}\; x \;=\; e_1;\; e_2 \;:\; \tau \mid \epsilon}\;[\textsc{Let}] \qquad \frac{\Gamma \;\vdash\; e \;:\; \tau \mid \langle\rangle \quad \overline{\alpha} \notin \mathrm{ftv}(\Gamma)}{\Gamma \;\vdash\; e \;:\; \forall\overline{\alpha}.\, \tau \mid \epsilon}\;[\textsc{Gen}]$$

$$\frac{\Gamma, x : \tau_1 \;\vdash\; e \;:\; \tau_2 \mid \epsilon'}{\Gamma \;\vdash\; \lambda x.\, e \;:\; \tau_1 \to \epsilon'\; \tau_2 \mid \epsilon}\;[\textsc{Lam}] \qquad \frac{\Gamma \;\vdash\; e_1 \;:\; \tau_2 \to \epsilon\; \tau \mid \epsilon \quad \Gamma \vdash\; e_2 \;:\; \tau_2 \mid \epsilon}{\Gamma \;\vdash\; e_1(e_2) \;:\; \tau \mid \epsilon}\;[\textsc{App}] \qquad \frac{\Gamma \;\vdash\; e \;:\; \forall\overline{\alpha}.\, \tau \mid \epsilon}{\Gamma \;\vdash\; e \;:\; \tau[\overline{\alpha} \mapsto \overline{\tau}] \mid \epsilon}\;[\textsc{Inst}]$$

$$\frac{\begin{array}{c}\Gamma \;\vdash\; e \;:\; \tau \mid \langle l \mid \epsilon\rangle \quad \Gamma, x : \tau \vdash\; e_r \;:\; \tau_r \mid \epsilon \quad \Sigma(l) = \{op_1,\; \ldots,\; op_n\} \\ \Gamma, \mathit{resume} \;:\; \tau_i' \to \epsilon\; \tau_r,\; x_i \;:\; \tau_i \;\vdash\; e_i \;:\; \tau_r \mid \epsilon \quad \Gamma \;\vdash\; op_i \;:\; \tau_i \to \langle l\rangle\; \tau_i' \mid \langle\rangle\end{array}}{\Gamma \;\vdash\; \mathsf{handle}\{\; op_1(x_1) \to e_1;\; \ldots;\; op_n(x_n) \to e_n;\; \mathsf{return}\; x \to e_r\; \}(e) \;:\; \tau_r \mid \epsilon}\;[\textsc{Handle}]$$

Fig. 3. Type rules.

Evaluation contexts:

$$\begin{array}{lll} E & ::= [\,] \mid E(e) \mid v(E) \mid \mathsf{val}\; x \;=\; E;\; e \mid \mathsf{handle}\{h\}(E) & \\ X_{op} & ::= [\,] \mid X_{op}(e) \mid v(X_{op}) \mid \mathsf{val}\; x \;=\; X_{op};\; e & \\ & \;\;\mid\;\; \mathsf{handle}\{h\}(X_{op}) & \text{if } op \notin h \end{array}$$

Reduction rules:

$$\begin{array}{llll} (\delta) & c(v) & \longrightarrow & \delta(c, v) \quad \text{if } \delta(c, v) \text{ is defined} \\ (\beta) & (\lambda x.\;\; e)(v) & \longrightarrow & e[x \;\mapsto\; v] \\ (\mathit{let}) & \mathsf{val}\;\; x \;\;=\;\; v;\;\; e & \longrightarrow & e[x \;\mapsto\; v] \\ & & & \\ (\mathit{return}) & \mathsf{handle}\{h\}(v) & \longrightarrow & e[x \;\mapsto\; v] \\ & & & \text{where} \\ & & & \quad (\mathsf{return}\;\; x \;\to\; e) \;\; \in h \\ & & & \\ (\mathit{handle}) & \mathsf{handle}\{h\}(X_{op}[op(v)]) & \longrightarrow & e[x \;\mapsto\; v,\;\; \mathit{resume} \;\mapsto\; r] \\ & & & \text{where} \\ & & & \quad (op(x) \;\to\; e) \;\; \in h \\ & & & \quad r \;\;=\;\; \lambda y.\;\; \mathsf{handle}\{h\}(X_{op}[y]) \end{array}$$

Fig. 4. Reduction rules and evaluation contexts

different from $\langle \mathit{exc} \rangle$. There are some use-cases for this but in practice we have not found many uses for duplicate effects (nor any drawbacks).

The type rules for our calculus is given in Figure 3. A type environment Γ maps variables to types and can be extended using a comma: if Γ' equals $\Gamma, x : \sigma$, then $\Gamma'(x) \;=\; \sigma$ and $\Gamma'(y) \;=\; \Gamma(y)$ for any $x \neq y$. A type rule $\Gamma \;\vdash\; e \;:\; \tau \mid \epsilon$ states that under environment Γ, the expression e has type τ with possible effects ϵ. All the type rules are straightforward and support complete and principal type inference. We refer the reader to (Leijen, 2017) for further details.

4.1. Operational semantics

In this section we define a precise semantics for our core language with algebraic effect handlers. It has been shown that well-typed programs cannot go 'wrong' under these semantics (Leijen, 2017). The operational semantics of our calculus is given in Figure 4 and consists of just five evaluation rules. We use two evaluation contexts: the E context is the usual one for a call-by-value lambda calculus. The X_{op} context is used for handlers and evaluates down through any handlers that do *not* handle the operation *op*. This is used to express concisely that the 'nearest enclosing handler' handles particular operations.

The first three reduction rules, (δ), (β), and (let) are the standard rules of call-by-value evaluation. The final two rules evaluate handlers. Rule (return) applies the return clause of a handler when the argument is fully evaluated. Note that this evaluation rule subsumes both lambda- and let-bindings and we can define both as a reduction to a handler without any operations:

$$(\lambda x.\; e_1)(e_2) \;\equiv\; \mathsf{handle}\{\mathsf{return}\; x \to e_1\}(e_2)$$

and

$$\mathsf{val}\; x \;=\; e_1;\; e_2 \;\equiv\; \mathsf{handle}\{\mathsf{return}\; x \to e_2\}(e_1)$$

The next rule, (handle), is where all the action is. Here we see how algebraic effect handlers are closely related to delimited continuations as the evaluation rules captures a delimited 'stack' $X_{op}[op(v)]$ under the handler h. Using a X_{op} context ensures by construction that only the innermost handler containing a clause for *op*, can handle the operation $op(v)$. Evaluation continues with the expression ϵ but besides binding the parameter x to v, also the *resume* variable is bound to the continuation: $\lambda y.\; \mathsf{handle}\{h\}(X_{op}[y])$. Applying *resume* results in continuing evaluation at X_{op} with the supplied argument as the result. Moreover, the continued evaluation occurs again under the handler h.

Resuming under the same handler is important as it ensures that our semantics correspond to the original categorical interpretation of algebraic effect handlers as a *fold* over the effect algebra (Plotkin and Pretnar, 2013). If the continuation is not resumed under the same handler, it behaves more like a *case* statement doing only one level of the *fold*. Such handlers are sometimes called *shallow handlers* (Kammar et al., 2013; Lindley et al., 2017).

For this article we do not formalize parameterized handlers as shown in Section 2.3. However the reduction rule is straightforward. For example, a handler with a single parameter p is reduced as:

$$\begin{array}{l} \mathsf{handle}\{h\}(p \;=\; v_p)(X_{op}[op(v)]) \\ \;\;\longrightarrow \{\; op(v) \to e \;\in h\;\} \\ e[x \mapsto v,\; p \mapsto v_p,\; \mathit{resume} \mapsto \lambda q\; y.\, \mathsf{handle}\{h\}(p \;=\; q)(X_{op}[y])] \end{array}$$

Using the reduction rules of Figure 4 we can define the evaluation function ($\longmapsto$), where $E[e] \;\longmapsto\; E[e']$ iff $e \;\longrightarrow\; e'$. We also define the function $\longmapsto\!\!\!\!\rightarrow$ as the reflexive and transitive closure of $\longmapsto$.

4.2. Comparison with Delimited Continuations

Shan (2007) has shown that various variants of delimited continuations can be defined in terms of each other. Following Kammar et al. (2013), we can define a variant of Danvy and Filinski's shift and reset operators (1990) , called shift_0 and reset_0, as

$$\mathit{reset}_0(X_s[\mathit{shift}_0(\lambda k.\; e)]) \longrightarrow e[k \mapsto \lambda x.\; \mathit{reset}_0(X_s[x])]$$

where we write X_s for a context that does not contain a $reset_0$. Therefore, the $shift_0$ captures the continuation up to the nearest enclosing $reset_0$. Just like handlers, the captured continuation is itself also wrapped in a $reset_0$. Unlike handlers though, the handling is done by the $shift_0$ directly instead of being done by the delimiter $reset_0$. From the reduction rule, we can easily see that we can implement delimited continuations using algebraic effect handlers, where $shift_0$ is an operation and $X_s \equiv X_{shift_0}$:

$$reset_0(e) \doteq \mathsf{handle}\{\ shift_0(f) \rightarrow f(resume)\ \}(e)$$

Using this definition, we can show it is equivalent to the original reduction rule for delimited continuations, where we write h for the handler $shift_0(f) \rightarrow f(resume)$:

$$\begin{array}{l} reset_0(X_s[shift_0(\lambda k.\ e)]) \doteq \mathsf{handle}\{h\}(X_s[shift_0(\lambda k.\ e)]) \\ \longrightarrow \\ (f(resume))[f \mapsto \lambda k.\ e,\ resume \mapsto \lambda x.\ \mathsf{handle}\{h\}(X_s[x])] \\ \longrightarrow \\ (\lambda k.\ e)(\lambda x.\ \mathsf{handle}\{h\}(X_s[x])) \\ \longrightarrow \\ e[k \mapsto \lambda x.\ \mathsf{handle}\{h\}(X_s[x])] \doteq e[k \mapsto \lambda x.\ reset_0(X_s[x])] \end{array}$$

Even though we can define this equivalence in our untyped calculus, we cannot give a general type to the $shift_0$ operation in our system. To generally type shift and reset operations a more expressive type system with answer types is required (Danvy and Filinski, 1989; Asai and Kameyama, 2007). In recent work Forster, Kammar, Lindley, and Pretnar (2017) show that it is possible to go the other direction and implement handlers using delimited continuations, improving on an earlier result (Kammar et al., 2013) that used mutable state.

5. Conclusion

We have shown how we can implement full *async-await* style programming with algebraic effects. We hope that abstractions like *cancelable* and *timeout* will find their way into other asynchronous platforms as well. In the future we plan to apply ambient programming in the context of web services with algebraic effects.

References

Kenichi Asai, and Yukiyoshi Kameyama. "Polymorphic Delimited Continuations." In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, 239–254. APLAS'07. Singapore. 2007. doi:10.1007/978-3-540-76637-7_16.

Steve Awodey. *Category Theory*. Oxford Logic Guides 46. Oxford university press. 2006.

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. "Pause 'n' Play: Formalizing Asynchronous C#." In *Proceedings of the 26th European Conference on Object-Oriented Programming*, 233–257. ECOOP'12. Beijing, China. 2012. doi:10.1007/978-3-642-31057-7_12.

Ethan Brown. *Web Development with Node and Express*. O'Reilly. Jul. 2014.

Olivier Danvy, and Andrzej Filinski. *A Functional Abstraction of Typed Contexts*. DIKU, University of Copenhagen. 1989.

Olivier Danvy, and Andrzej Filinski. "Abstracting Control." In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 151–160. LFP '90. Nice, France. 1990. doi:10.1145/91556.91622.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*. Sep. 2015.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. "On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control." In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*. ICFP'17. 2017. arXiv:1610.09161.

Ben R. Gaster, and Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham. 1996.

Patricia Johann, and Neil Ghani. "Foundations for Structured Programming with GADTs." In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 297–308. POPL '08. San Francisco, California, USA. 2008. doi:10.1145/1328438.1328475.

Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in Action." In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 145–158. ICFP '13. ACM, New York, NY, USA. 2013. doi:10.1145/2500365.2500590.

Oleg Kiselyov, and Hiromi Ishii. "Freer Monads, More Extensible Effects." In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 94–105. Haskell '15. Vancouver, BC, Canada. 2015. doi:10.1145/2804302.2804319.

Peter J. Landin. *A Generalization of Jumps and Labels*. UNIVAC systems programming research. 1965.

Peter J. Landin. "A Generalization of Jumps and Labels." *Higher-Order and Symbolic Computation* 11 (2): 125–143. 1998. doi:10.1023/A:1010068630801. Reprint from (Landin, 1965).

John Launchbury, and Amr Sabry. "Monadic State: Axiomatization and Type Safety." In *In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, 227–238. ICFP'97. 1997. doi:10.1145/258948.258970.

Daan Leijen. "Extensible Records with Scoped Labels." In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312. 2005.

Daan Leijen. "HMF: Simple Type Inference for First-Class Polymorphism." In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. Sep. 2008. doi:10.1145/1411204.1411245. Extended version available as a technical report: MSR-TR-2007-118, Sep 2007, Microsoft Research.

Daan Leijen. "Flexible Types: Robust Type Inference for First-Class Polymorphism." In *POPL'09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 66–77. Feb. 2009. doi:10.1145/1594834.1480891.

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

Daan Leijen. "Madoko: Scholarly Documents for the Web." In *Proceedings of the 2015 ACM Symposium on Document Engineering*, 129–132. DocEng '15. ACM, Lausanne, Switzerland. 2015. doi:10.1145/2682571.2797097.

Daan Leijen. "Koka Overview and Reference." 2016. `http://bit.do/kokabook`.

Daan Leijen. *Algebraic Effects for Functional Programming*. MSR-TR-2016-29. Microsoft Research technical report. Aug. 2016. Extended version of (Leijen, 2017).

Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. "The Design of a Task Parallel Library." In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 227–242. OOPSLA'09. Orlando, Florida, USA. 2009. doi:10.1145/1640089.1640106.

Jeff Lewis, Mark Shields, Erik Meijer, and John Launchbury. "Implicit Parameters: Dynamic Scoping with Static Types." In *27th ACM Symp. on Principles of Programming Languages (POPL'00)*, 108–118. Boston, Massachussets. Jan. 2000. doi:10.1145/325694.325708.

Sam Lindley, and James Cheney. "Row-Based Effect Types for Database Integration." In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 91–102. TLDI'12. 2012. doi:10.1145/2103786.2103798.

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

Eugenio Moggi. "Notions of Computation and Monads." *Information and Computation* 93 (1): 55–92. 1991. doi:10.1016/0890-5401(91)90052-4.

Martin Odersky. “Implicit Function Types.” Dec. 2016. `https://www.scala-lang.org/blog/2016/12/07/implicit-function-types.html`. Blog post.

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

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

Gordon D. Plotkin, and Matija Pretnar. “Handling Algebraic Effects.” In *Logical Methods in Computer Science*, volume 9. 4. 2013. doi:10.2168/LMCS-9(4:23)2013.

Matija Pretnar. “Inferring Algebraic Effects.” *Logical Methods in Computer Science* 10 (3). 2014. doi:10.2168/LMCS-10(3:21)2014.

Axel Rauschmayer. *Exploring ES6: Upgrade to the next Version of JavaScript*. 2016. `http://exploringjs.com/es6`.

Didier Rémy. “Type Inference for Records in Natural Extension of ML.” In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994. doi:10.1.1.48.5873.

Chung-chieh Shan. “A Static Simulation of Dynamic Delimited Control.” *Higher-Order and Symbolic Computation* 20 (4): 371–401. 2007. doi:10.1007/s10990-007-9010-4.

Martin Sulzmann. *Designing Record Systems*. YALEU/DCS/RR-1128. Yale University. Apr. 1997.

Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. “Lightweight Monadic Programming in ML.” In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 15–27. ICFP'11. Tokyo, Japan. 2011. doi:10.1145/2034773.2034778.

Wouter Swierstra. “Data Types à La Carte.” *Journal of Functional Programming* 18 (4): 423–436. Jul. 2008. doi:10.1017/S0956796808006758.

The EcmaScript committee. “ES6: The EcmaScript 2015 Language Specification.” 2015. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`.

The EcmaScript committee. “ES7: The Draft EcmaScript 2017 Language Specification.” 2016. `https://tc39.github.io/ecma262`.

Hayo Thielecke. “Using a Continuation Twice and Its Implications for the Expressive Power of Call/CC.” *Higher Order Symbolic Computation* 12 (1): 47–73. Apr. 1999. doi:10.1023/A:1010068800499.

Niki Vazou, and Daan Leijen. “From Monads to Effects and Back.” In *Proceedings of the 18th International Symposium on the Practical Aspects of Declarative Languages*, 169–186. PADL'16'. 2016. doi:10.1007/978-3-319-28228-2_11.

Nicolas Wu, and Tom Schrijvers. “Fusion for Free: Efficient Algebraic Effect Handlers.” In *Proceedings of the International Conference on Mathematics of Program Construction*. MPC'15. 2015. doi:10.1.1.723.5577.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. “Effect Handlers in Scope.” In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Göthenburg, Sweden. 2014. doi:10.1145/2633357.2633358.