Type Directed Compilation of Row-Typed Algebraic Effects

Daan Leijen
Microsoft Research, USA
daan@microsoft.com

Abstract

Algebraic effect handlers, are recently gaining in popularity as a purely functional approach to modeling effects. In this article, we give an end-to-end overview of practical algebraic effects in the context of a compiled implementation in the Koka language. In particular, we show how algebraic effects generalize over common constructs like exception handling, state, iterators and async-await. We give an effective type inference algorithm based on extensible effect rows using scoped labels, and a direct operational semantics. Finally, we show an efficient compilation scheme to common runtime platforms (such as JavaScript, the JVM, or .NET) using a type directed selective CPS translation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Languages, Design

Keywords Algebraic Effect Handlers, Type Inference, CPS transformation

1. Introduction

Algebraic effects (introduced by Plotkin and Power in 2002 [36]) and algebraic effect handlers (introduced by Plotkin and Pretnar in 2009 [35]), are recently gaining in popularity as a purely functional approach to modeling effects. As a restriction on general monads, algebraic effects come with various advantages: they can be freely composed, and there is a natural separation between their interface (as a set of operations) and their semantics (as a handler).

At this time, implementations are usually based on libraries [18, 20, 21, 51], or interpreted run-times [3, 16]. This is unfortunate, because we believe that algebraic effect handlers have wide applicability and should be considered as a basic mechanism for handling effects and control-flow in a wide range of languages – including languages like JavaScript and C#, which have added various specialized constructs over the years for concepts that are naturally expressed using algebraic effects.

In this article, we give an end-to-end overview of practical algebraic effects in the context of a compiled implementation in the Koka language. In particular:

- In Section 2 we present a language design for algebraic effects, and show how algebraic effects subsume many control-flow constructs that are specialized in other languages, e.g. exceptions, state, iterators, async-await, etc. In particular, iterators and async-await are complex constructs that can lead to subtle interactions with other features and require complex compilation mechanisms [4]. Being able to generalize over them using a single well-founded abstraction is a huge win.
- Typing algebraic effects, such that we can check what effects functions can have, and can check that all effects are handled in a program, is a challenge. We show in Section 3 how we can leverage the row-types of Koka, based on scoped labels [24], to implement sound and complete type inference for algebraic effects. A problem with many effect systems is that the inferred types become large or difficult to understand – we have extensive experience with large effectful programs that suggests that the row-type approach of Koka works well in practice.
- In Section 3.2 we show a novel approach to simplify polymorphic effect types. This turns out to be beneficial for an efficient CPS translation too.
- Operational semantics for algebraic effects in the literature are usually given in a continuation style calculus [3, 18] – this simplifies semantics and is convenient when reasoning about effects. In contrast, we give a more direct operational semantics to algebraic effects using syntactical contexts [50] in Section 4 (recent work by Lindley et al. [16, 30], and Kammar and Pretnar [19] use a similar style of operational semantics). This style of semantics does not use continuations but captures the execution context explicitly. For compilation purposes we believe this approach is more convenient. We also prove that well-typed programs cannot go 'wrong' under these semantics.
- Section 5 describes efficient compilation of algebraic effects to common runtime platforms (like JavaScript) where we do not have full control over the runtime stack. We show how we can use a type directed selective CPS translation to compile effect handlers efficiently. It turns out that the standard CPS translation does not work for functions with polymorphic effect variables, and in Section 5.3.2 we show a novel technique where we use polymorphic code duplication to dynamically pick the correct runtime representation of such functions.

There is a full implementation of algebraic effects in Koka, see [27] for detailed instructions on how to download it and program with algebraic effects.

2. Overview

We are going to demonstrate algebraic effects in the context of Koka – a call-by-value programming language with a type inference system 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. 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 [32] 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 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. The 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 [26]. In this article we extend the original system [25] to use algebraic effects and handlers to define new effect types.

In the following sections we give various examples of programming with algebraic effects, where we give particular attention to cases where algebraic effects subsume control-flow constructs that are specialized in many other languages, such as exceptions, iterators, and async-await. The interested reader may take a quick look ahead at Figure 4 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.

There are various ways to understand algebraic effects and handlers. As described originally [35, 36], the signature of the effect operations forms a free algebra which gives rise to a free monad. Free monads provide a natural way to give semantics to effects, where handlers describe a *fold* over the algebra of operations [44]. Using a more operational perspective, we can also view algebraic effects 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

The exception effect *exc* can be defined in Koka as:

```
effect exc {
  raise( s : string ) : a
}
```

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

```
fun safediv( x, y ) {
  if (y==0) then raise("divide by zero") else x / y
}
```

Type inference will infer the type $(int,int)\ \rightarrow\ exc\ int$ propagating the 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) {
    raise(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:

$$catch : (\ action\colon ()\ \rightarrow\ \langle exc \mid e\rangle\ a,\ h : string\ \rightarrow\ e\ a)\ \rightarrow\ e\ a$$

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

```
fun zerodiv(x,y) {
  catch( { safediv(x,y) }, fun(s){ 0 } )
}
```

has type $(int,int)\ \rightarrow\ \langle\rangle\ int$ and is a total function. Note that the Koka syntax $\{\ safediv(x,y)\ \}$ denotes an anonymous function that takes no arguments.

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)
    raise(s)  → Nothing
}}
```

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

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

$$\mathsf{handle}(action)\ \{\ \ldots\ \}\ \equiv\ (\mathsf{handler}\{\ \ldots\ \})(action)$$

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)
  raise(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.

2.2. State: Resuming Operations

The exception effect is somewhat special as it never resumes: any operations following the *raise* are never executed. Usually, operations will *resume* with a specific result instead of cutting the computation short. For example, we can have 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 {
  return x → x
  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
```

As another example, 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 () → ⟨*state*⟨*int*⟩,*console*,*div* | *e*⟩ () with the state 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. The function returned by the handler construct now 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
```

2.3. Iterators

Many contemporary languages, like JavaScript or C#, have special syntax and compilation rules for iterators and the *yield* statement [45]. Algebraic effects generalize over this where the *yield* effect can be defined as:

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

The *yield* effect generalizes over 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.

2.4. Ambiguity: Multiple Resumptions

You can enter a room once, yet leave it twice.
— Peter Landin [22, 23]

In the previous examples we looked at abstractions that never resume (e.g. exceptions), and abstractions that resume once (e.g. state and iterators). 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 that implement some variant of *callcc* [47]. 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 $(action : () \rightarrow \langle amb, ndet \mid e\rangle\ a) \rightarrow \langle ndet \mid e\rangle\ a$ 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 $amb : (action : () \rightarrow \langle amb \mid e\rangle\ a) \rightarrow e\ list\langle a\rangle$, 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 handler, 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 how, 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 [43, 49] where composing different monads required implementing a combined monad by hand.

2.5. Asynchronous Programming

Similarly to iterators, many programming languages are adding support for async-await style asynchronous programming [46]. For example, web servers written in JavaScript using NodeJS are highly asynchronous and without language support the resulting programs are difficult to write and debug due to excessive callbacks (e.g. the so-called "pyramid of doom"). However, extending a language with async-await is non-trivial, 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 [4].

Again, algebraic effect handlers generalize naturally over this pattern. In contrast to the earlier examples we can generally not implement this directly in our language but need to use primitives of the host system. For concreteness, we assume NodeJS as our host with a primitive to call readline:

```
prim-readline : (oninput : string → ()) → io ()
```

which calls its argument call back on successful input. We can now define an asynchronous effect as:

```
effect async {
  readline() : string
}
```

The handler for the asynchronous effect must effectively surround the entire program as it relies on the outer NodeJS event loop to re-invoke our callback when input is ready:

```
val outer-async = handler {
  readline() → prim-readline( resume )
}
```

We see that the *readline* clause just returns and exits the program to the outer NodeJS event loop. However, it registered *resume* as the callback – effectively resuming with the result input when available. In the Koka implementation the core library defines *async* as an abstract effect with a predefined handler around the *main* function. The handled operations are more generic such that library writers can easily wrap any asynchronous primitives provided by the host system. Moreover, since it is just another effect, it composes naturally with any other algebraic effects the user defines, such as state and exceptions.

Using asynchronous operations is straightforward now:

```
fun ask-age() {
  println("what is your name?")
  val name = readline()      // asynchronous!
  println("hello " + name)
}
```

Note that even though the previous example is now asynchronous, the program is written in an entirely straightforward manner where the type of the program signifies asynchronicity. In async-await style programming an async call site is signaled by an *await* keyword and each asynchronous method with an *async* keyword. This can be helpful for understanding the code. With our effect typing, the *type* signifies the effects code can have and the asynchronicity is immediately apparent through the inferred types of any expression.

```
ask-age: () → ⟨async,console⟩ ()
```

The previous example does not use asynchronicity in any essential way but in general it is used to serve multiple requests interleaved where no request handler will block on I/O operations. Moreover, the Koka core library provides primitives like *async-all* to start multiple asynchronous operations that are interleaved with each other.

In future work we are planning to write highly robust asynchronous web servers using algebraic effect handlers. A similar technique as shown here is used in multi-core OCaml where one-shot algebraic effects are used to implement concurrency [11].

2.6. Domain Specific Effects: Parsing

All the previous examples are well-known effects and are available in various forms in other languages too. However, we can also implement *domain-specific effects*. For example, in a compiler we may have a *name-supply* effect that generates fresh names, a *warning* effect for logging warnings, or an *inference* effect that maintains a typing environment. Like monads, encapsulating effects allows for abstraction over a particular implementation and removes the need to explicitly deal with environments and result values.

As an example of such domain-specific effect, we show how to implement a *parse* effect to implement parser combinators [17, 28]. In particular, the effect will abstract over the current input state, handling failure, and combining multiple parse results. Following the original example of Wu et al. [51], we first extend the *amb* effect of Section 2.4 to describe multiple parse results and failures:

```
effect many {
  flip() : bool
  fail() : a
}
```

Using *flip* we can already describe choice between two parsers:

```
fun choice(p1,p2) { if (flip()) then p1() else p2() }
```

The *choice* combinator seems somewhat magical since it uses the *flip* operation as an oracle but we will see that this allows for multiple evaluation strategies. Using *choice*, we can define the *many* combinator for parsing a sequence of zero or more *p* parsers:

```
fun many(p)   { choice( { many1(p) }, { Nil } ) }
fun many1(p)  { Cons(p(), many(p)) }
```

where *many* has the inferred type

```
many: (p : () → ⟨many,div | e⟩ a) → ⟨many,div | e⟩ list⟨a⟩
```

A possible handler for the *many* effect returns all possible results:

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

Another handler is *eager* which returns the first successful result:

```
val eager = handler {
  return x → Just(x)
  fail() → Nothing
  flip() → match(resume(False)) {
    Nothing → resume(True)
    Just(x) → Just(x)
}}
```

Here, the *False* branch is taken first and the result examined to determine whether we should explore the *True* branch or not. The types of these handlers are:

```
  fun solutions : (() → ⟨many | e⟩ a) → e list⟨a⟩
  fun eager : (() → ⟨many | e⟩ a) → e maybe⟨a⟩
```

To do actual parsing, we are defining the *parse* effect with just one operation to test if the current input satisfies a predicate:

```
effect parse { satisfy⟨a⟩(string → maybe⟨(a,string)⟩) : a }
```

Note that the result type a is locally quantified, e.g. the type of *satisfy* is

```
satisfy: forall⟨a⟩ (string → maybe⟨(a,string)⟩) → parse a
```

A handler for the *parse* effect can be defined as:

```
val parse = handler(input) {
  return x    → (x,input)
  satisfy(p)  → match(p(input)) {
    Nothing          → fail()
    Just((x,rest))   → resume(rest,x)
}}
```

In this handler, we use *fail* operation from the *many* effect to handle failure. The handler is parameterized with the current *input* string and the final result is tupled with any remaining input:

```
parse : (string,() → ⟨parse,many | e⟩ a) → ⟨many | e⟩ (a,string)
```

With the *satisfy* combinator we can create basic parsers to recognize symbols and digits:

```
fun symbol(c : char) : parse char {
  satisfy( fun(input) { match(input.first) {
    Just((d,rest)) | d == c → Just((c,rest))
    _ → Nothing
  })
}

fun digit(c : char) : parse int {
  satisfy( fun(input) { match(input.first) {
    Just((d,rest)) | d.digit? → Just((d - '0').int,rest))
    _ → Nothing
  })
}
```

We combine the *digit* parser with $many_1$ in order to parse numbers:

```
fun number() {
  many1(digit).foldl(0, fun(n,d) { 10 * n + d })
}
```

With these building blocks in place, parsing simple expressions becomes straightforward:

```
fun expr() : ⟨div,parse,many⟩ int {
  choice {
    val i : int = term()
    symbol('+')
    i + term()
  }
  { term() }
}

fun term() {                     fun factor() {
  choice {                         choice(number) {
    val i : int = factor()           symbol('(')
    symbol('*')                      val i = expr()
    val j = factor()                 symbol(')')
    i * j                            i
  }                                }
  { factor() }                   }
}
```

Using the expression parser *expr*, we can now use the the *solutions* and *parse* handlers to apply the parsers to simple expressions:

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

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

Total functions	$\tau_1 \rightarrow \tau_2$	$\doteq$	$\tau_1 \rightarrow \langle\rangle\; \tau_2$
Effects	ϵ	$\doteq$	τ^{e}
Effect variables	μ	$\doteq$	α^{e}
Effect labels	l	$\doteq$	$c^k\langle \tau_1, ..., \tau_n\rangle \quad k\;=\;... \rightarrow \mathsf{k}$
Closed effects	$\langle l_1, ..., l_n\rangle$	$\doteq$	$\langle l_1, ..., l_n \mid \langle\rangle\rangle$
Effect extension	$\langle l_1, ..., l_n \mid \epsilon\rangle$	$\doteq$	$\langle l_1 \mid ... \langle l_n \mid \epsilon\rangle ... \rangle$

Figure 1. Syntax of expressions, types, and kinds

$$\epsilon \cong \epsilon \quad [\text{EQ-REFL}] \qquad\qquad \frac{\epsilon_1 \cong \epsilon_2 \quad \epsilon_2 \cong \epsilon_3}{\epsilon_1 \cong \epsilon_3}\;[\text{EQ-TRANS}]$$

$$\frac{\epsilon_1 \cong \epsilon_2}{\langle l \mid \epsilon_1\rangle \cong \langle l \mid \epsilon_2\rangle}\;[\text{EQ-HEAD}] \qquad \frac{l_1 \ncong l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon\rangle\,\rangle \cong \langle l_2 \mid \langle l_1 \mid \epsilon\rangle\,\rangle}\;[\text{EQ-SWAP}]$$

$$\frac{c \neq c'}{c\langle \tau_1, ..., \tau_n\rangle \ncong c'\langle \tau_1', ..., \tau_n'\rangle}\;[\text{UNEQ-LABEL}]$$

Figure 2. Row equivalence

$>$ *solutions*{ *parse*("1+2*3", *expr*) }
[(7,""),(3,"*3"),(1,"+2*3")]

Changing the parsing strategy it orthogonal to the implementation of the parsers, and we can apply the *eager* handler to return the first successful parse result:

$>$ *eager*{ *parse*("1+2*3", *expr*) }
Just((7,""))

We hope this section gives a taste of the power of abstraction offered by algebraic effect handlers. In general, any free monad can be readily expressed with algebraic effects.

3. Type Rules

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 [24]. Originally, user-defined effects were described using a monadic approach [49] but it turns out that algebraic effects fit the original type system well with almost no changes. The new system based on algebraic effects is much simpler and allows for free composition of user defined effects.

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 $\langle l_1, ..., l_n\rangle$, or an *open effect* of the form $\langle l_1, ..., l_n \mid \mu\rangle$.

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 [24] 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 [14, 29, 38, 42], 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 different from $\langle exc\rangle$. There are some use-cases for this but in practice we have not found many uses for duplicate effects (nor any drawbacks).

3.1. Type Inference

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.

The type rules are quite standard. The rule VAR derives the type of a variable x with an arbitrary effect ϵ. We may have expected to derive only the total effect $\langle\rangle$ since the evaluation of a variable has no effect at all. However, there is no rule that lets one upgrade the final effect and instead we need to pick the final effect right away. Another way to look at this is that since the variable evaluation has

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

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

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

Figure 3. Type rules.

no effect, we are free to assume any arbitrary effect. We use the VAR rule too for operations *op* and constants *c* where we assume the types of those are part of the initial environment.

The LAM rule is similar in that it assumes any effect ϵ for the result since the evaluation of a lambda is a value. At this rule, we also see how the effect derived for the body of a lambda ϵ' shifts to the derived function type $\tau_1 \to \epsilon'\ \tau_2$. Rule APP is standard and derives an effect ϵ requiring that its premises derive the same effect as the function effect.

Rules INST and GEN instantiate and generalize types. The generalization rule has an interesting twist as it requires the derived effect to be total. When combining our calculus with polymorphic mutable reference cells, this is required to ensure a sound semantics. This is the semantic equivalent to the syntactic value restriction in ML. In our core calculus we cannot define polymorphic reference cells directly so the restriction is not necessary persé but it seems good taste to leave it in as it is required for the full Koka language.

Finally, the HANDLE rule types effect handlers. We assume that effect declarations populate an initial environment Γ_0 with the types of declared operations, and also a *signature environment* Σ that maps declared effect labels to the set of operations that belong to it. We also assume that all operations have unique names, such that given the operation names, we can uniquely determine to which effect *l* they belong.

The rule HANDLE requires that all operations in the signature $\Sigma(l)$ are part of the handler, and we reject handlers that do not handle all operations that are part of the effect *l*. The `return` clause is typed with $x : \tau$ where τ is the result type of the handled action. All clauses must have the same result type τ_r and effect ϵ. For each operation clause $op_i(x_i) \to e_i$ we first look up the type of op_i in the environment as $\tau_i \to \langle l\rangle\,\tau_i'$, and bind x_i to the argument type of the operations, and bind *resume* to the function $\tau_i' \to \tau_r$ where the argument type of resume is the result type of the operation. The derived type of the handler is a function that discharges the effect type *l*.

3.2. Simplifying Types

The rule APP is a little surprising since it requires both the effects of the function and the arguments to match. This only works because we set things up to always be able to infer the effects of functions that are 'open' – i.e. have a polymorphic μ in their tail. For example, consider the identity function:

$$id = \lambda x.\, x$$

If we assign the valid type $\forall\alpha.\, \alpha \to \langle\rangle\ \alpha$ to the *id* function, we get into trouble quickly. For example, the application $id(raise(\text{"hi"}))$ would not type check since the effect of *id* is total while the effect of the argument contains *exc*. Of course, the type inference algorithm always infers a most general type for *id*, namely $\forall\alpha\,\mu.\, \alpha \to \mu\ \alpha$ which has no such problems.

In practice though we wish to simplify the types more and leave out 'obvious' polymorphism. In Koka we adopted two extra type rules to achieve this. The first rule opens closed effects of function types:

$$\frac{\Gamma \vdash e : \tau_1 \to \langle l_1, ..., l_n\rangle\ \tau_2 \mid \epsilon}{\Gamma \vdash e : \tau_1 \to \langle l_1, ..., l_n \mid \epsilon'\rangle\ \tau_2 \mid \epsilon}\ [\textsc{open}]$$

With this rule, we can type the application $id(raise(\text{"hi"}))$ even with the simpler type assigned to *id* as we can open the effect type of *id* using the OPEN rule to match the effect of $raise(\text{"hi"})$. We combine this with a closing rule (which is just an instance of INST/GEN):

$$\frac{\begin{array}{l}\Gamma \vdash e : \forall\mu\overline{\alpha}.\, \tau_1 \to \langle l_1, ..., l_n \mid \mu\rangle\ \tau_2 \mid \epsilon \\ \mu \notin \mathsf{ftv}(\tau_1, \tau_2, l_1, ..., l_n)\end{array}}{\Gamma \vdash e : \forall\overline{\alpha}.\, \tau_1 \to \langle l_1, ..., l_n\rangle\ \tau_2 \mid \epsilon}\ [\textsc{close}]$$

During inference, the rule CLOSE is applied (when possible) before assigning a type to a let-bound variable. In general, such technique would lead to incompleteness where some programs that were well-typed before, may now be rejected since CLOSE assigns a less general type. However, due to OPEN this is not the case: at every occurrence of such let-bound variable the rule OPEN can always be applied (possibly surrounded by INST/GEN) to lead to the original most general type – i.e. even though the types are simplified, the set of typeable programs is unchanged.

We have significant experience with the Koka type system in practice with several large programs (up to 14.000 loc) and type simplification works well in practice. As we will see later in Section 5, the OPEN rule actually proves essential for an efficient CPS translation of algebraic effects.

3.3. Type Inference

The type system as defined is closely related to the core type system originally presented for Koka [25], with the main differences that this paper uses a simpler presentation without treating isolated state, and we added the rule for handlers. Since the handler rule is straightforward and can be encoded using regular applications and lambdas, the results in [25] carry over directly.

In particular, we can define syntax directed type rules that are sound and complete with respect to the rules in Figure 3, and there exists a sound and complete type inference algorithm. Due to this, it was straightforward to change the Koka compiler to type check algebraic effects and handlers as there were no changes to the core type inference algorithm. The main difficulties that needed to be overcome were found in an efficient compilation to common runtime platforms as described in Section 5.

4. Operational Semantics

In this section we define a precise semantics for our core language with algebraic effect handlers, and show that well-typed programs

Evaluation contexts:

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

Reduction rules:

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

Figure 4. Reduction rules and evaluation contexts

cannot go 'wrong'. Even though algebraic effects are originally conceived with a semantics in category theory, we will give a more regular operational semantics. The main reason to do so is that a direct operational semantics is more useful as a guidance for efficient compilation as described in Section 5. Moreover, using a direct operational semantics may help exposing algebraic effects to a wider audience and allows us to use a traditional style proof of soundness.

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. In particular, it 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:

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

and

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

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 [35]. 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* [18, 30].

For this article we do not formalize parameterized handlers as shown in Section 2.2. However the reduction rule is straightforward. For example, a handler with a single parameter p is reduced as:

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

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 $\longmapsto$.

4.1. Optimizing Tail-Resumptions

From the reduction rules, we can already see some possible optimizations that can be used to compiler handlers efficiently. For example, if a handler never resumes, we can treat it similarly to how exceptions are handled and do not need to capture the execution context. An important other optimization can apply to *tail resumptions*, i.e. a *resume* that occurs in the tail position of an operation clause. Suppose we have an operations clause h with $op(x) \rightarrow resume(e)\ \in h$ and $resume\ \notin \mathsf{fv}(e)$. In that case, we can derive:

$$\begin{array}{l} \mathsf{handle}\{h\}(X_{op}[op(v)]) \\ \quad \longrightarrow \\ resume(e)[x \mapsto v,\ resume \mapsto \lambda y.\ \mathsf{handle}\{h\}(X_{op}[y])] \\ \quad \longrightarrow \\ (\lambda y.\ \mathsf{handle}\{h\}(X_{op}[y]))(e[x \mapsto v]) \\ \quad \longrightarrow^*\ \{\ e[x \mapsto v] \longrightarrow^*\ v'\ \} \\ (\lambda y.\ \mathsf{handle}\{h\}(X_{op}[y]))(v') \\ \quad \longrightarrow \\ \mathsf{handle}\{h\}(X_{op}[v']) \end{array}$$

That means that in an implementation we do not need to capture and restore the context X_{op} at all but can directly evaluate the operation expression as if it was a regular function call. Of course, special precautions must be taken that any operations yielded in the evaluation of $e[x \mapsto v]$ are not handled by any handler in $\mathsf{handle}\{h\}(X_{op}[])$.

4.2. Comparison with Delimited Continuations

Shan [41] has shown that various variants of delimited continuations can be defined in terms of each other. Following Kammar et al. [18], we can define a variant of Danvy and Filinski's [7] shift and reset operators, called $shift_0$ and $reset_0$, as

$$reset_0(X_s[shift_0(\lambda k.\ e)]) \longrightarrow e[k \mapsto \lambda x.\ reset_0(X_s[x])]$$

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 $shift_0(f) \rightarrow f(resume)$:

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

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 [1, 6]. Kammar et al. [18] also show that it is possible to go the other direction and implement handlers using delimited continuations but that solution requires mutable reference cells to implement a global handler stack.

4.3. Soundness: Well Typed Effect Handlers Cannot Go Wrong

Under our semantics, well-typed programs cannot go *wrong*:

Theorem 1. (*Semantic soundness*)
If $\cdot \vdash e : \tau \mid \epsilon$ then either $e \Uparrow$ or $e \longmapsto\!\!\!\!\rightarrow v$ where $\cdot \vdash v : \tau \mid \epsilon$.

where we use the notation $e \Uparrow$ for a never-ending reduction. The proof of this theorem consists of showing two main lemmas:

- Show that reduction in the operational semantics preserves well-typing, i.e. subject reduction, and,
- Show that faulty expressions are not typeable.

If programs are closed and well-typed we know from subject reduction that we can only reduce to well-typed terms, which are either faulty, a value, or an expression containing a further redex. Since faulty expressions are not typeable, it must be that evaluation either produces a well-type value or diverges. (Often a soundness proof is done using *progress* instead of *faulty* expressions but we use the latter technique since it turns out that for proving state isolation [25] this technique works better).

Subject reduction is stated more precisely as:

Lemma 1. (*Subject reduction*)
If $\Gamma \vdash e_1 : \tau \mid \epsilon$ and $e_1 \longmapsto e_2$ then $\Gamma \vdash e_2 : \tau \mid \epsilon$.

To show that subject reduction holds we need to establish various other lemmas. Two particularly important ones are the substitution and replacement lemmas:

Lemma 2. (*Substitution*)
If $\Gamma,\ x : \forall\overline{\alpha}.\ \tau \vdash e : \tau' \mid \epsilon$ where $x \notin \mathsf{dom}(\Gamma)$, $\Gamma \vdash v : \tau \mid \epsilon$, and $\overline{\alpha} \not\cap \mathsf{ftv}(\Gamma)$, then $\Gamma \vdash e[x \mapsto v] : \tau' \mid \epsilon$.

$$\begin{array}{rcll}
e & ::= & x^{\sigma} \mid c^{\sigma} \mid e(e) & \\
& \mid & \lambda^{\epsilon} x : \sigma.\ e & \\
& \mid & \mathsf{val}\ x = e;\ e & \text{binding} \\
& \mid & e\langle\sigma\rangle \mid \Lambda\alpha.\ e & \text{type application/abstraction} \\
& \mid & e\langle\!\langle\epsilon\rangle\!\rangle & \text{opening an effect} \\
& \mid & \mathsf{handle}\langle l\rangle\{h\}(e) &
\end{array}$$

Figure 5. Syntax of explicitly typed Koka

Lemma 3. (*Replacement*)
If D is a deduction ending in $\Gamma \vdash E[e] : \tau \mid \epsilon$, and D' is a sub-deduction of D ending in $\Gamma' \vdash e : \tau' \mid \epsilon'$ and occurs at the hole of E, and $\Gamma \vdash e' : \tau' \mid \epsilon'$, then we have that $\Gamma \vdash E[e'] : \tau \mid \epsilon$.

The proofs of these lemmas carry over directly from [50]. Using these lemmas we can prove subject reduction. We focus on the interesting cases for (*let*), (*return*) and (*handle*):

Proof. (*Subject reduction*)
We prove by induction over the reduction rules of $\longrightarrow$.
case $\mathsf{val}\ x = v;\ e \longrightarrow e[x \mapsto v]$: From LET we have $\Gamma \vdash v : \sigma \mid \epsilon$, and $\Gamma, x : \sigma \vdash e : \tau \mid \epsilon$, and by Lemma 2, we can derive that $\Gamma \vdash e[x \mapsto v] : \tau \mid \epsilon$.
case $\mathsf{handle}\{h\}(v) \longrightarrow e[x \mapsto v]$ with $\mathsf{return}\ x \rightarrow e \in h$ (**0**): From HANDLE we have $\Gamma \vdash v : \tau \mid \_$ (**1**), and $\Gamma, x : \tau \vdash e : \tau_r \mid \epsilon$. Using (1) and lemma 2 we can derive $\Gamma \vdash e[x \mapsto v] : \tau_r \mid \epsilon$.
case $\mathsf{handle}\{h\}(X_{op}[op(v)]) \longrightarrow e[x \mapsto v,\ resume \mapsto \lambda y.\ \mathsf{handle}\ \{h\}(X_{op}[y])]$ with $op(x) \rightarrow \epsilon \in h$ (**0**): Assume $op \in \Sigma(l)$ (**1**). From HANDLE we have $\Gamma \vdash X_{op}[op(v)] : \tau \mid \langle l | \epsilon\rangle$. By (1) we can derive $G_0 \vdash op : \tau_1 \rightarrow \langle l\rangle\ \tau_2 \mid \langle\rangle$, and thus from APP and $op(v)$, we derive $\Gamma \vdash v : \tau_1 \mid \_$ (**2**), and $\Gamma \vdash op(v) : \tau_2 \mid \_$ (**3**). Using (3) and lemma 3, and assuming $\cdot \vdash y : \tau_2 \mid \_$, it follows $\Gamma \vdash X_{op}[y] : \tau \mid \langle l | \epsilon\rangle$. Using HANDLE and (1), we also can derive $\Gamma \vdash \mathsf{handle}\{h\}(X_{op}[y]) : \tau_r \mid \epsilon$, and through the rule LAM, we have $\Gamma \vdash \lambda y.\ \mathsf{handle}\{h\}(X_{op}[y]) : \tau_2 \rightarrow \epsilon\ \tau_r \mid \_$ (**4**). Again from HANDLE and (0) we have $\Gamma, resume : \tau_2 \rightarrow \epsilon\ \tau_r,\ x : \tau_1 \vdash e : \tau_r \mid \epsilon$, and using lemma 1 in combination with (2) and (4) we conclude $\Gamma \vdash e[x \mapsto v,\ resume \mapsto \lambda y.\ \mathsf{handle}\{h\}(X_{op}[y])]$ □

4.4. Faulty Expressions

The main purpose of type checking is of course to guarantee that wrong expressions cannot occur. Apart from the usual errors, like adding a number to a string, we have one more kind of error in our system that we would like to avoid, namely using operations without a corresponding handler to give semantics:

Lemma 4. (*Faulty expressions are not typeable*)
a. If $\Gamma \vdash c(v) : \tau | \epsilon$ then $\delta(c, v)$ is defined.
b. If $\Gamma \vdash X_{op}[op(v)] : \tau | \epsilon$, with $op \in \Sigma(l)$, then $l \in \epsilon$.

The second statement is somewhat unusual since it concerns itself with effects only. It is a powerful lemma though as it states that effect types cannot be discarded (except through handlers). This lemma also implies effect types are *meaningful*, e.g. if a function does not have an *exc* effect, it will never throw an exception.

Proof. (*Lemma 4.b*)
Suppose $\Gamma \vdash X_{op}[op(v)] : \tau | \epsilon$ with $op \in \Sigma(l)$ (**1**). To be well typed, we must have $\Gamma \vdash op(v) : \_ \mid \epsilon'$ with $l \in \epsilon'$ (due to (1)). We use induction on the structure of X_{op} to show that $l \in \epsilon'$ for any $\Gamma \vdash X_{op}[op(v)] : \_ \mid \epsilon'$.
case $X_{op}(e')$: Due to the induction hypothesis, the premise in rule APP is typed with an effect ϵ' with $l \in \epsilon'$ and therefore also in the result effect $l \in \epsilon'$.
case $v(X_{op})$: as the previous case.
case $\mathsf{val}\ x = X_{op};\ e$: Similarly to the previous cases, the premise

$$\frac{}{\vdash\ c^{\sigma}\ :\ \sigma \mid \epsilon}\ [\textsc{Con}] \qquad \frac{\vdash\ e\ :\ \sigma \mid \epsilon}{\vdash\ \Lambda\alpha.\, e\ :\ \forall\alpha.\,\sigma \mid \epsilon}\ [\textsc{TLam}] \qquad \frac{\vdash\ e\ :\ \forall\alpha.\,\sigma \mid \epsilon}{\vdash\ e\langle\sigma_1\rangle\ :\ \sigma[\alpha \mapsto \sigma_1] \mid \epsilon}\ [\textsc{TApp}]$$

$$\frac{}{\vdash\ x^{\sigma}\ :\ \sigma \mid \epsilon}\ [\textsc{Var}] \qquad \frac{\vdash\ e\ :\ \sigma \mid \epsilon}{\vdash\ \lambda^{\epsilon} x : \sigma_1.\, e\ :\ \sigma_1 \to \epsilon\,\sigma \mid \epsilon'}\ [\textsc{Lam}] \qquad \frac{\vdash\ e_1\ :\ \sigma_2 \to \epsilon\,\sigma \mid \epsilon \quad \vdash\ e_2\ :\ \sigma_2 \mid \epsilon}{\vdash\ e_1(e_2)\ :\ \sigma \mid \epsilon}\ [\textsc{App}]$$

$$\frac{\vdash\ e\ :\ \sigma_1 \to \langle l_1, ..., l_n\rangle\, \sigma_2 \mid \epsilon'}{\vdash\ e\langle\!\langle\epsilon\rangle\!\rangle\ :\ \sigma_1 \to \langle l_1, ..., l_n | \epsilon\rangle\, \sigma_2 \mid \epsilon'}\ [\textsc{Open}] \qquad \frac{\begin{array}{c}\Sigma(l)\ =\ \{op_1, ..., op_n\} \quad \vdash\ op_i\ :\ \sigma_i \to \langle l\rangle\, \sigma'_i \mid \langle\rangle \\ \vdash\ e\ :\ \sigma_r \mid \langle l|\epsilon\rangle \quad \vdash\ e_i\ :\ \sigma \mid \epsilon \quad \vdash\ e_r\ :\ \sigma \mid \epsilon\end{array}}{\vdash\ \mathsf{handle}\langle l\rangle\{\ op_1(x : \sigma_1) \to e_1;\ ...;\ op_n(x : \sigma_n) \to e_n;\ \mathsf{return}\, x : \sigma_r \to e_r\,\}(e)\ :\ \sigma \mid \epsilon}\ [\textsc{handle}]$$

Figure 6. Type rules for explicitly typed Koka

is typed with an effect ϵ' with $l \in \epsilon'$ and therefore due to rule LET the result effect also has $l \in \epsilon'$
case $\mathsf{handle}\{h\}(X_{op})$ with $op \notin h$: since $op \notin h$ the handler discharges some effect l' with $l \neq l'$. Using rule HANDLE and APP we have a premise $\Gamma\ \vdash\ X_{op}\ :\ \_ \mid \langle l', l|\_\rangle$ and a result effect $\langle l|\_\rangle$ and thus $l \in \epsilon'$ □

5. Compilation

Compiling algebraic effects efficiently is not straightforward. In particular, as can be seen in the operational semantics of Figure 4, the rule for handlers captures a delimited execution context $X_{op}[op(v)]$ which in practice means we need to capture the call stack up to the handler.

If we have full control over the runtime system, this can be done in a straightforward manner similarly how many compilers for Scheme and ML implement *callcc*. This approach does not work though when targeting a common runtime platform, like the JVM or the .NET environment. For Koka in particular, we compile to JavaScript to take advantage of the rich libraries and runtime environments (like high performance asynchronous web services in NodeJS).

In these environments we cannot capture the call stack and need to use other mechanisms to implement effect handlers. One way to avoid capturing the stack, is to translate the program into continuation passing style (CPS) [10]. This makes the evaluation context explicit in the current continuation. For example, Scheme implementations usually use this in order to implement both proper tail-calls, as well as *callcc*, when targeting JavaScript [31, 48, 52]. This was also used in Scala to provide first-class delimited continuations on the JVM platform [40].

With a CPS translation, the evaluation context X_{op} essentially disappears since all constructs take an explicit continuation function k as a last argument. For example, the xor function from the Section 2.4 would get CPS translated into:

$$\mathsf{val}\ xor\ =\ \lambda k.\, flip(\lambda p.\, flip(\lambda q.\ k(\ (p||q)\ \&\&\ !(p\&\&q)\ )\ ))$$

We can see that an operation call $op(v)$ becomes $op(v, k)$ where k is a function taking the result of the operation call. The reduction rule for handlers essentially becomes:

$$\begin{array}{l}\mathsf{handle}\{h\}(H_{op}[op(v, k)]) \\ \ \ \longrightarrow\ \{\ op(x) \to e \in h\ \} \\ e[x \mapsto v,\ resume \mapsto \lambda y.\, \mathsf{handle}\{h\}(H_{op}[k(y)])]\end{array}$$

where the context H_{op} is now strictly a stack of handlers. In an implementation it is straightforward to maintain such a 'shadow' stack explicitly.

At first, we tried a full CPS translation in Koka but it turned out to slow down the code significantly. One of the larger programs written in Koka is a markdown processor called Madoko [26] (and this article is entirely written in Madoko!). This program runs usually client-side in the browser and with a full CPS translation it started to use too much resources to run reliably. A better approach was needed.

5.1. A Type-Directed Selective CPS Translation.

It has long been recognized that one can selectively CPS transform only parts of the program that need it [8, 9, 39]. In our case, we only have to use CPS translation on those parts that may issue effectful operations. Moreover, since effects are tracked in the type system, we can use a type-directed selective CPS translation (as used by Scala [40] for example). We built on the translation by Nielsen [33] who introduces a sound selective CPS translation for the simply typed lambda calculus extended with *callcc* and *throw*. However, the translation by Nielsen applies to monomorphic effects only and we will see that in the presence of polymorphic effect variables the translation becomes more complex.

We define the CPS translation over an explicitly typed core calculus, defined in Figure 5. This is the internal core calculus of Koka generated by type inference. It is essentially System F [15] extended with the effect annotations. In particular, lambda's carry the effect of the body as ϵ. Similarly, handlers are annotated with the handled effect type l. Finally, there is a special construct $e\langle\!\langle\epsilon\rangle\!\rangle$ that *opens* the effect of e with effect ϵ – this is generated whenever the OPEN rule is applied as discussed in Section 3.2 on simplifying types.

The type checking rules for the explicitly typed core calculus are given in Figure 6. A rule $\vdash\ e\ :\ \sigma \mid \epsilon$ states that a given expression e has type σ under a given ϵ, where the effect ϵ is inherited and not synthesized. We can see this in rule LAM where the annotated effect determines the effect of the body. Moreover, we avoid having to pass around an explicit type environment in the type rules by annotating all variables and constants with their type – which should match the type of the binder by convention.

5.1.1. Selective Translation

For a selective translation we need a function $\mathcal{H}$ (for *handled* effects) that determines based on the type if a CPS translation is needed. The handler function has the form $\mathcal{H}(\theta, \epsilon)$ where θ is a set of 'unhandled' effect variables (μ):

$$\begin{array}{ll}\mathcal{H}(\theta, \langle l|\epsilon\rangle) = & \mathcal{H}(l) \vee \mathcal{H}(\theta, \epsilon) \\ \mathcal{H}(\theta, \langle\rangle) & = false \\ \mathcal{H}(\theta, \mu) & = \mu \notin \theta\end{array}$$

We also overload $\mathcal{H}(l)$ to determine if a particular effect l may need CPS translation. For now, we assume $\mathcal{H}(l)$ is always *true*. In the Koka implementation though, we distinguish built-in effects from user defined effects through the kind system. The built-in effects consist of exceptions (exn), non-termination (div), non-determinism ($ndet$), polymorphic state ($st\langle h\rangle$), and general I/O operations (io). All of these are usually provided directly by the target system (like JavaScript) and can thus be directly compiled without

$$
\begin{array}{llll}
\text{Static expression} & \mathsf{e} ::= & e & \text{(expression)} \\
 & \quad\mid & \boldsymbol{\lambda}\mathsf{x}.\mathsf{e} & \text{(abstraction)} \\
 & \quad\mid & \mathsf{e}[\mathsf{e}] & \text{(application)}
\end{array}
$$

$$\frac{}{\theta \vdash c^{\sigma} \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[c]}\;[\textsc{con}]$$

$$\frac{}{\theta \vdash x^{\sigma} \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[x]}\;[\textsc{var}]$$

$$\frac{\theta \vdash e \rightsquigarrow \mathsf{e}}{\theta \vdash e\langle\tau\rangle \rightsquigarrow \mathsf{e}}\;[\textsc{tapp}]$$

$$\frac{\theta \vdash e \rightsquigarrow \mathsf{e}}{\theta \vdash \Lambda\alpha.\, e \rightsquigarrow \mathsf{e}}\;[\textsc{tlam}]$$

$$\frac{\begin{array}{c}\theta \vdash e \rightsquigarrow \mathsf{e} \quad \vdash e : \tau_1 \rightarrow \epsilon'\, \tau_2 \\ \mathcal{H}(\theta,\epsilon') \vee \neg\mathcal{H}(\theta,\epsilon)\end{array}}{\theta \vdash e\langle\!\langle\epsilon\rangle\!\rangle \rightsquigarrow \mathsf{e}}\;[\textsc{open}]$$

$$\frac{\begin{array}{c}\theta \vdash e \rightsquigarrow \mathsf{e} \quad \vdash e : \tau_1 \rightarrow \epsilon'\, \tau_2 \\ \neg\mathcal{H}(\theta,\epsilon') \wedge \mathcal{H}(\theta,\epsilon)\end{array}}{\theta \vdash e\langle\!\langle\epsilon\rangle\!\rangle \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[\lambda x\, k.\, \mathsf{e}[\boldsymbol{\lambda}\mathsf{f}.k(\mathsf{f}(x))]\!]}\;[\textsc{open-cps}]$$

$$\frac{\theta \vdash e_1 \rightsquigarrow \mathsf{e}_1 \quad \theta \vdash e_2 \rightsquigarrow \mathsf{e}_2}{\theta \vdash \mathsf{val}\, x = e_1;\ e_2 \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{e}_1[\boldsymbol{\lambda}\mathsf{x}.\mathsf{val}\, x = \mathsf{x};\ \mathsf{e}_2[\mathsf{k}]]}\;[\textsc{val}]$$

$$\frac{\theta \vdash e \rightsquigarrow \mathsf{e} \quad \theta \vdash e_r \rightsquigarrow \mathsf{e}_r \quad \theta \vdash e_i \rightsquigarrow \mathsf{e}_i}{\begin{array}{l}\theta \vdash \mathsf{handle}\langle l\rangle\{\, op_i(x:\sigma_i) \rightarrow e_i;\ \mathsf{return}\, x:\sigma_r \rightarrow e_r\,\}(e) \rightsquigarrow \\ \boldsymbol{\lambda}\mathsf{k}.\mathsf{handle}_l\{\, op_i(x) \rightarrow \mathsf{e}_i[\mathsf{k}];\ \mathsf{return}\, x \rightarrow \mathsf{e}_r[\mathsf{k}]\,\}(\mathsf{e}[\boldsymbol{\lambda}\mathsf{x}.\mathsf{x}])\end{array}}\;[\textsc{handle}]$$

$$\frac{\theta \vdash e_1 \rightsquigarrow \mathsf{e}_1 \quad \theta \vdash e_2 \rightsquigarrow \mathsf{e}_2 \quad \vdash e_1 : \tau_1 \rightarrow \epsilon\, \tau_2 \quad \neg\mathcal{H}(\theta,\epsilon)}{\theta \vdash e_1(e_2) \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{e}_1[\boldsymbol{\lambda}\mathsf{f}.\mathsf{e}_2[\boldsymbol{\lambda}\mathsf{x}.\mathsf{k}[\mathsf{f}(\mathsf{x})]]]}\;[\textsc{app}]$$

$$\frac{\theta \vdash e_1 \rightsquigarrow \mathsf{e}_1 \quad \theta \vdash e_2 \rightsquigarrow \mathsf{e}_2 \quad \vdash e_1 : \tau_1 \rightarrow \epsilon\, \tau_2 \quad \mathcal{H}(\theta,\epsilon)}{\theta \vdash e_1(e_2) \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{e}_1[\boldsymbol{\lambda}\mathsf{f}.\mathsf{e}_2[\boldsymbol{\lambda}\mathsf{x}.\mathsf{f}(\mathsf{x}, \lambda y.\, \mathsf{k}[y])]]}\;[\textsc{app-cps}]$$

$$\frac{\theta \vdash e \rightsquigarrow \mathsf{e} \quad \neg\mathcal{H}(\theta,\epsilon) \quad \epsilon \neq \langle l_1, \ldots, l_n \mid \mu\rangle}{\theta \vdash \lambda^{\epsilon} x : \tau.\, e \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[\lambda^{\epsilon} x.\, \mathsf{e}[\boldsymbol{\lambda}\mathsf{x}.\mathsf{x}]]}\;[\textsc{lam}]$$

$$\frac{\theta \vdash e \rightsquigarrow \mathsf{e} \quad \mathcal{H}(\theta,\epsilon) \quad \epsilon \neq \langle l_1, \ldots, l_n \mid \mu\rangle}{\theta \vdash \lambda^{\epsilon} x : \tau.\, e \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[\lambda x\, k.\, \mathsf{e}[\boldsymbol{\lambda}\mathsf{x}.k(\mathsf{x})]\!]}\;[\textsc{lam-cps}]$$

$$\frac{\epsilon = \langle l_1, \ldots, l_n \mid \mu\rangle \quad \theta/\{\mu\} \vdash e \rightsquigarrow \mathsf{e}_{cps} \quad \theta \cup \{\mu\} \vdash e \rightsquigarrow \mathsf{e}_{plain}}{\theta \vdash \lambda^{\epsilon} x : \tau.\, e \rightsquigarrow \boldsymbol{\lambda}\mathsf{k}.\mathsf{k}[\lambda x\, k.\ \mathsf{if}\ k?\ \mathsf{then}\ (\mathsf{e}_{cps}[\boldsymbol{\lambda}\mathsf{x}.k(\mathsf{x})])\ \mathsf{else}\ (\mathsf{e}_{plain}[\boldsymbol{\lambda}\mathsf{x}.\mathsf{x}])]}\;[\textsc{lam-dup}]$$

Figure 7. Type directed selective CPS translation.

needing CPS translation. This turns out to be an important optimization as many (leaf) functions do not use any user defined effects and can thus be compiled directly – and implies there is no cost for effect handlers for any code that does not use them.

For any function f with a function type with effect ϵ where $\mathcal{H}(\theta, \epsilon)$ is *true*, f is translated in CPS style and will have an extra continuation parameter k at runtime. When $\mathcal{H}(\theta, \epsilon)$ is *false*, the function is compiled as usual without a continuation parameter.

The last case of $\mathcal{H}$ for a polymorphic effect variable μ is only *false* if μ is an element of the set θ. In general we always need to assume a CPS translation is needed for any μ as such a variable may get instantiated later on with a user-defined effect. However, as we will see, for polymorphic functions we need to generate two translations and the θ set is used to force certain effect variables to be treated as needing no CPS translation.

In the case of polymorphic functions, the simplified types of Section 3.2 turn out to have a performance impact as well: many functions that would otherwise get a type with an open polymorphic effect, are now closed and thus do not need a CPS translation as $\mathcal{H}$ will be *false*. In the Koka core library, this reduced the set of CPS translated functions by over 80%.

5.1.2. Translation Rules

Using the $\mathcal{H}$ function, Figure 7 defines a type directed selective CPS translation for our explicitly typed core calculus. Each rule of the form $\theta \vdash e \rightsquigarrow \mathsf{e}$ states that expression e gets translated into a *static expression* e assuming a set of unhandled effect variables θ (initially empty). Following Nielsen [33], we write the translation itself in a continuation passing style; to distinguish the translation lambda's and applications from the ones in the program we write static applications as $\mathsf{e}[\mathsf{e}]$ and static lambda's as $\boldsymbol{\lambda}$. In a rule $e \rightsquigarrow \mathsf{e}$ the e expression is a function that takes itself a continuation function k, which takes expressions to cps expressions.

The rules translate explicitly typed core into a untyped lambda calculus. We would have liked the target calculus to be explicitly typed as well, but as we will see, that would require further type rules to deal with variadic functions. In the actual Koka implementation we do translate to an explicitly typed core though.

Most rules are standard, except for the OPEN and LAM rules. Rules CON and VAR pass the value on to the continuation. The TAPP and TLAM rules just pass on the translation of their bodies. In the VAL rule we can see why a continuation based translation works well, as we can pass the binding of x as a static continuation itself. The APP rule is specialized depending on whether effect is CPS translated or not. The HANDLE rule is straightforward and translates all its subexpressions. In practice we would need to provide a shared binding for k though since the rule as state might duplicate code. The result is an application $handle_l$ which is a handler for effect l. On every target platform this must be implemented as a primitive.

That leaves the OPEN and LAM rules which need more explanation.

5.2. Opening: Non-CPS to CPS

The OPEN and OPEN-CPS rule open an effect type. However, opening an effect type $\langle l_1, \ldots, l_n\rangle$ to $\langle l_1, \ldots, l_n | \epsilon\rangle$ may change the runtime representation: in particular, if l_1 to l_n are all built-in effects, but ϵ is a handled effect, the function type changes from being non-CPS to CPS translated! – the CPS translated result should now take a continuation k as its last argument. The rule OPEN-CPS defines this case and wraps the non-CPS translated function in a lambda that takes a continuation k and applies that k directly to the result of applying the translated non-CPS function. This is effectively the point where non-CPS functions are lifted into the CPS world at runtime.

5.3. Closing: CPS to Non-CPS

With the OPEN rules, the runtime representation can be changed directly with a small and constant cost. There is one other place where the runtime representation can change and that is due to type instantiation. Unfortunately, in this case we cannot so easily change the term at runtime. In particular, a function type may be hidden inside some other type. For example, if we define a data type as:

```
type hide⟨e⟩ { Hide( f : int → e int ) }
```

then we can have terms of the form $Hide(id) : \forall\mu.\ hide\langle\mu\rangle$. When this is instantiated it is not clear how to convert such a term at runtime back to a non-CPS form. This is a deep problem and in other work on monadic effects [49] led to restrictions on the amount of polymorphism allowed in the effect system to avoid this situation.

Here we take another approach: we are going to try to *not* change the runtime representation of functions that are CPS translated, even if they are called from a non-CPS context and the continuation argument is lacking. We assume that our target environment supports some form of variadic functions and that we can check at runtime if the k argument is present or not. This is well supported in JavaScript but it works well in typed environments too. For example, for the .NET target, first-class functions are represented by function objects and we can modify the $Apply$ methods to check if the k parameter was present.

5.3.1. Assume Identity?

At first, we thought it was sufficient to default the continuation parameter to the identity function and assume $k = \lambda x.x$ if the argument was not present. That approach does not work though in the presence of effect polymorphic higher-order functions. Consider the map function:

```
fun map( xs : list⟨a⟩, f : a → e b) : e list⟨b⟩ {
  match(xs) {
    Nil → Nil
    Cons(x,xx) → Cons(f(x),map(xx,f))
  }
}
```

which is effect polymorphic. A naïve CPS translation leads to (assuming a match construct and tupling):

```
val map_cps = λ(xs, f, k).
  match(xs) {
    Nil → k(Nil)
    Cons(x, xx) → f(x, λy. map(xx, f, λyy. k( Cons(y, yy)) ))
  }
```

Note in particular that f itself is called as a CPS function with a continuation argument. If we now have a call to map where the effect type is immediately instantiated to the empty effect, the continuation argument k will not be present. For example, the explicitly typed expression,

$map\langle\langle\rangle, int, int\rangle([1], inc)$

where we assume $inc : int \rightarrow int$, would get translated to:

$map([1], inc)$

Even if map would detect that the k parameter is not present and substitute $k = \lambda x.x$, this would still go wrong at runtime as inc is called inside map with a continuation argument!

5.3.2. Polymorphic Duplication

The solution is to generate *two* translations of every function polymorphic in some effect μ – one is the CPS translation (e.g. map_{cps}), and one is a plain translation (e.g. map_{plain}). We then use a wrapper that chooses either implementation at runtime based on whether the continuation argument is present. For example, the wrapper for map becomes:

```
val map = λ(xs, f, k). if k? then map_cps(xs, f, k) else map_plain(xs, f)
```

where we assume $k?$ tests if k was present. The implementation of the question mark operator is dependent on the particular target environment. For example, in our JavaScript backend we can simply use $k\ !==\ undefined$.

The duplicate translation rule is LAM-DUP – here we finally use the θ set to generate a CPS translated version of the body (using $\theta/\{\mu\}$), and non-CPS version too (using $\theta \cup \{\mu\}$). Finally we generate a wrapper to choose the correct version at runtime. The other two rules, LAM and LAM-CPS are used for non effect polymorphic functions.

There is a performance advantage too to this translation – similarly to the worker-wrapper transformation [34] a target platform can usually inline most call sites. Since many of these effect polymorphic functions abstract over iteration patterns (like map) this gives the target platform more opportunities to optimize loops since the $plain$ versions will be more amenable to common loop optimizations.

Since only the tail of an effect row can be polymorphic, there is no risk of exponential code duplication. Even for the Koka core library which contains many of these higher-order effect polymorphic functions, the code size increased by a modest 20%.

5.4. The Runtime System

The runtime system needs to implement the $handle_l$ primitive. In general, the $handle_l$ function registers the operation and return clauses and pushes a handler frame for effect l on the handler stack. When an operation is performed, it searches along the handler stack for a handler frame and calls into the appropriate operation clause with the current continuation as an argument.

In our JavaScript implementation we have refined this where we have a generic handler that implements a trampoline: operations just return to this handler loop where they are handled and resumed. This way we ensure the stack always gets unrolled for code that uses operations – this essentially implements proper tail calls (except that the programmer needs to call an operation every once in a while).

6. Related Work

Algebraic effects were described by Plotkin and Power [36] as an algebraic model of computational effects. Later Plotkin and Pretnar [35] added algebraic effect handlers to describe exceptions. Various implementations of algebraic effects exist. Kammar et al. [18] show an efficient implementation as a library in Template Haskell where they use a continuation monad to implement handlers. Using first-class patterns, Wu et al. [51] also embed algebraic effects as a library in Haskell. Brady [5] implements algebraic effects as a DSL library in the dependently typed Idris language. Kiselyov et al. [20, 21] implement a extensible effects in Haskell using the ‘freer’ monad.

The language Eff [3] is an ML-like language designed around algebraic effect handlers. It also provides support for dynamic effect resources which can be used to model polymorphic mutable references. There are designs for an effect type system with type inference for Eff [2, 37] based subtyping and a region system.

The Links language [29] has recently been extended with support for algebraic effect handlers. Just like we reused the original effect system of Koka, Hillerström and Lindley [16] describe how the original Links type system can be naturally extended to handle algebraic effects too. Their system is also based on row-polymorphism but they use instead Remy style rows [38] where the kind system is extended to record presence or absence of effects in the row. We believe our approach based on scoped labels [24] is simpler in practice, but the Remy style can be more expressive as it can describe the absence of effects.

Frank [30] is an experimental language based solely on effect handlers where there is no primitive notion of a function: its handlers all the way down. Just like scoped labels, effects may occur multiple times in a row.

In contrast to our work, all of the previous implementations have full control over the runtime stack being implemented as an interpreter, a library, or by using specialized runtimes like OCaml. Also, many other type systems use *structural* types for the effects where each effect operation (and its type) occurs in the inferred types. In this paper we use a *nominal* system where a single effect type implies the available operations (also used by Frank [30]). We believe this is better in practice to keep inferred types small and understandable. For example, compare the type of the *state* handler in Section 2.2 with the type inferred in Links [16] (where _ denotes absence):

```
sig state : (s) -> (Comp({Get:s,Put:(s) {}-> ()|e},a))
                        -> Comp({Get{_},Put{_}|e},a)
```

where the operations of the state effect are explicit in the type. A drawback of our approach is that an effect handler must handle all operations of a particular effect type, and cannot pick and choose arbitrarily.

There are also approaches to handling effects using monads [43, 49]. A significant drawback of these approaches is that monads do not naturally compose and combining different effects is difficult in these systems. An exception is Filinski's work on layered monads which do support a similar style of composing effects [12, 13].

7. Conclusion

Algebraic effect handlers concisely describe many complex control-flow constructs in various programming languages. We hope that the language design, the direct operational semantics, and compilation scheme described in this article will contribute to wider adoption of algebraic effects. In the future we plan use algebraic effects to implement strongly typed asynchronous web services in NodeJS.

References

[1] Kenichi Asai, and Yukiyoshi Kameyama. "Polymorphic Delimited Continuations." In *APLAS'07*, 239–254. 2007. doi:10.1007/978-3-540-76637-7_16.

[2] Andrej Bauer, and Matija Pretnar. "An Effect System for Algebraic Effects and Handlers." *Logical Methods in Computer Science* 10 (4). 2014.

[3] Andrej Bauer, and Matija Pretnar. "Programming with Algebraic Effects and Handlers." *J. Log. Algebr. Meth. Program.* 84 (1): 108–123. 2015. doi:10.1016/j.jlamp.2014.02.001.

[4] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. "Pause 'n' Play: Formalizing Asynchronous C#." In *ECOOP 2012 – Object-Oriented Programming: 26th European Conference, Beijing, China*, edited by James Noble, 233–257. Springer. 2012. doi:10.1007/978-3-642-31057-7_12.

[5] Edwin Brady. "Programming and Reasoning with Algebraic Effects and Dependent Types." In *Proc. of ICFP'13*, 133–144. 2013. doi:10.1145/2500365.2500581.

[6] Olivier Danvy, and Andrzej Filinski. *A Functional Abstraction of Typed Contexts*. 1989.

[7] 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.

[8] Olivier Danvy, and John Hatcliff. "CPS-Transformation After Strictness Analysis." *ACM Lett. Program. Lang. Syst.* 1 (3). ACM: 195–212. Sep. 1992. doi:10.1145/151640.151641.

[9] Olivier Danvy, Jung-taek Kim, and Kwangkeun Yi. "Assessing the Overhead of ML Exceptions by Selective CPS Transform." In *In Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 103–114. 1998.

[10] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. "On One-Pass CPS Transformations." *J. Funct. Program.* 17 (6): 793–812. Nov. 2007. doi:10.1017/S0956796807006387.

[11] S Dolan, L White, Sivaramakrishnan K, Yallop J, and A Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*. Sep. 2015.

[12] Andrzej Filinski. "Representing Layered Monads." In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, 175–188. ACM Press. 1999.

[13] Andrzej Filinski. "Monads in Action." In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 483–494. 2010. doi:10.1145/1706299.1706354.

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

[15] Jean-Yves Girard. "The System F of Variable Types, Fifteen Years Later." *TCS*. 1986.

[16] Daniel Hillerström, and Sam Lindley. "Liberating Effects with Rows and Handlers." TyDe 2016. Nara, Japan. 2016. doi:10.1145/2976022.2976033.

[17] Graham Hutton, and Erik Meijer. *Monadic Parser Combinators*. NOTTCS-TR-96-4. Dept. of Computer Science, University of Nottingham. 1996. `http://www.cs.nott.ac.uk/Dept.{}/Staff/gmh/monparsing.ps`.

[18] 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.

[19] Ohad Kammar, and Matija Pretnar. "No Value Restriction Is Needed for Algebraic Effects and Handlers." *CoRR* abs/1605.06938. 2016.

[20] 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.

[21] Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible Effects: An Alternative to Monad Transformers." In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, 59–70. Haskell '13. Boston, Massachusetts, USA. 2013. doi:10.1145/2503778.2503791.

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

[23] 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 [22].

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

[25] 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.

[26] 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.

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

[28] Daan Leijen, and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. UU-CS-2001-27. Dept. of Computer Science, Universiteit Utrecht. 2001.

[29] Sam Lindley, and James Cheney. "Row-Based Effect Types for Database Integration." In *TLDI'12*, 91–102. 2012. doi:10.1145/2103786.2103798.

[30] Sam Lindley, Connor McBride, and Craig McLaughlin. "Do Be Do Be Do." In *POPL 2017*. Paris, France. 2016.

[31] Florian Loitsch, Manuel Serrano, and Inria Sophia Antipolis. "Hop Client-Side Compilation." In *Proceedings of the 8th Symposium on Trends on Functional Languages*. 2007.

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

[33] Lasse R. Nielsen. "A Selective CPS Transformation." *Electronic Notes in Theoretical Comp. Sc.* 45: 311–331. 2001. doi:10.1016/S1571-0661(04)80969-1. MFPS 2001, 17th Conf. on the Mathematical Foundations of Prog. Semantics.

[34] Simon L. Peyton Jones, and André L. M. Santos. "A Transformation-Based Optimiser for Haskell." *Science of Computer Programming* 32 (1): 3–47. 1998. doi:10.1016/S0167-6423(97)00029-4.

[35] 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.

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

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

[38] Didier Rémy. "Type Inference for Records in Natural Extension of ML." In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994.

[39] John Reppy. "Optimizing Nested Loops Using Local CPS Conversion." *Higher-Order and Symbolic Computation* 15 (2): 161–180. 2002. doi:10.1023/A:1020839128338.

[40] Tiark Rompf, Ingo Maier, and Martin Odersky. "Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform." In . ICFP. 2009.

[41] 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.

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

[43] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. "Lightweight Monadic Programming in ML." In *ICFP*. 2011. doi:10.1145/2034773.2034778.

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

[45] The EcmaScript committee. "ES6: The EcmaScript 2015 Language Specification." 2015. `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`.

[46] The EcmaScript committee. "ES7: The Draft EcmaScript 2017 Language Specification." 2016. `https://tc39.github.io/ecma262`.

[47] Hayo Thielecke. "Using a Continuation Twice and Its Implications for the Expressive Power of Call/Cc." *Higher Order Symbol. Comput.* 12 (1). Kluwer Academic Publishers, Hingham, MA, USA: 47–73. Apr. 1999. doi:10.1023/A:1010068800499.

[48] Eric Thivierge, and Marc Feeley. "Efficient Compilation of Tail Calls and Continuations to JavaScript." In *Proc. of the 2012 Annual Workshop on Scheme and Funct. Prog.*, 47–57. 2012. doi:10.1145/2661103.2661108.

[49] Niki Vazou, and Daan Leijen. "From Monads to Effects and Back." In *18th Int. Symp. on the Practical Aspects of Declarative Languages*, 169–186. 2016. doi:10.1007/978-3-319-28228-2_11.

[50] Andrew K. Wright, and Matthias Felleisen. "A Syntactic Approach to Type Soundness." *Inf. Comput.* 115 (1): 38–94. Nov. 1994. doi:10.1006/inco.1994.1093.

[51] 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. ACM, New York, NY, USA. 2014. doi:10.1145/2633357.2633358.

[52] Danny Yoo, and Shriram Krishnamurthi. "Whalesong: Running Racket in the Browser." In *Proceedings of the 9th Symposium on Dynamic Languages*, 97–108. DLS '13. ACM, Indianapolis, Indiana, USA. 2013. doi:10.1145/2508168.2508172.