Qualified Effect Types

Taming Control-Flow through Linear Effect Handlers

Jonathan Immanuel Brachthäuser
University of Tübingen
Germany
jonathan.brachthaeuser@uni-tuebingen.de

Daan Leijen
Microsoft Research
USA
daan@microsoft.com

Abstract

Combining control effects with global side-effects and external resources can lead to code which is hard to reason about. We introduce the concept of control-flow linearity (in contrast to data-flow linearity) as a tool for the programmer to reason about control-flow in the presence of control effects. Through a novel combination of an effect system with qualified types we track the control-flow linearity of each effect as part of the effect type of a function. We formalize control-flow linearity and prove soundness of our linearity analysis.

Editorial Note

This technical report is the result of an internship of Jonathan Brachthäuser at Microsoft Research, Redmond in 2018. While the report is published in 2023, the paper reflects the work at the time of writing.

1 Introduction

Control effects don't go well with global side-effects and external resources. Suppose we have a function that encapsulates opening and closing a file by giving an action that works directly on a file handle `h`:

```
fun with-file( path, action ) {
  val h = fopen(path)
  val x = action(h)
  fclose(h)
  x
}
```

Unfortunately, in the presence of control effects, this program is not correct. In particular, `action` may throw an exception in which case it never returns normally and therefore the file handle might not be closed. To deal with the resource safety, many solutions have been proposed ranging from `finally` statements [1, 23], automatic destructors [50], `defer` statements [15], finalizers [7], all the way to linear type systems [2, 6, 21, 51]. Most of these solutions focus on one particular effect: exceptions.

Algebraic effects [45] and handlers [47] are a a novel technique to structured programming with user defined effects where handlers define the semantics of an operation. Instead of extending a compiler with builtin effects, like exceptions, generators, or async/await, each of those can be library defined in terms of effect handlers. However, algebraic effect handlers exacerbate our trouble with the `with-file` function: before, only the `throw` operation of the exception effect could cause `action` to not return. Now, depending on its handler definition, *any* effect operation might cause it to not return! Even worse:

> *You can enter a room once, yet leave it twice.* – Landin, 1965

Landin wrote this in his seminal paper [32, 33] to highlight that functions that use first-class continuations might return more than once. This also holds for effect handlers. In our example, this might result in closing the file multiple times. To the best of our knowledge, only ad-hoc solutions exist to deal with multiple resumptions and side-effects exist, like `dynamic-wind` [19] or `initially` handlers [40].

The problem sketched above is in fact very general and always occurs when control operators to manipulate the control-flow are combined with external resources and global side-effects. In this paper we propose a new perspective on this old problem. The main idea is to annotate effects and handlers with their linearity constraints. For example, the `with-file` function may get a type like:

```
with-file: (string, (file-handle) → e a) → e a
             with (e ≤ linear)
```

allowing us to *locally* reason about the correctness of `with-file`: since the polymorphic effect type `e` restricted to be linear, the `action` won't throw exceptions or use other effects that result in resuming more than once. More generally:

- We introduce *control-flow linearity* (in contrast to data-flow linearity, that is, linear usage of resources) as a tool for the programmer, Section 2. It states whether a function will never return (abort), return exactly once (linear), at most once (affine), or multiple times (wild). Together these form a join semi-lattice as shown in Figure 1. We formalize control-flow linearity operationally in Section 4.
- We introduce a general framework of *qualified effect types* which propagates constraints through effect types (Section 3). This is a general theory and can be seen as a first step "to make algebraic effects

TR, November, 2018,
2018.

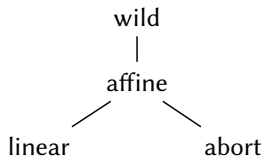

Fig. 1. The control-flow linearity lattice.

algebraic again" since we can propagate arbitrary predicates (like algebraic equations) to constrain handler implementations and usage contexts. In this paper we instantiate it to track control-flow linearity constraints specifically.

- The strength of effect handlers is that they define the semantics of effect operations completely within a single handler. We use this to locally analyze a handler and statically determine whether it satisfies the promised control-flow linearity constraints. This analysis is independent of the general framework of qualified effect types and can be instantiated differently for other constraints. In this paper we use a simple syntactic analysis (Sections 2.4 and 4.1).
- We prove that our analysis is *sound*: If the type of an expressions states that it only uses linear effects, then the control-flow will be linear at runtime (Section 4.2). In particular, any handler for a linear effect preserves the control-flow linearity of any expression using its operations. Combining the type system guarantees with effect handlers, we can now locally reason about our programs to ensure that the usage constraints of external resources are satisfied.
- The type system is built using three well established components, namely Hindley-Milner style type rules [26, 43], row-based effect types [20, 25, 34, 41], and qualified types [27, 29], and thus naturally supports type inference and fits well into existing implementations. We believe this has a high "power to weight" ratio where we can reason about control-flow linearity without needing complex type system extensions.

2 Effect Handlers and Control-Flow Linearity

The problems of combining control-effects with external resources and the idea of control-flow linearity are both general and apply to any system with advanced control effects. For concreteness though, we give all our examples in the Koka language, a strict language with effect inference where every function has a corresponding effect type [39].

2.1 Control-Flow Linearity

Combining control effects like exceptions or (delimited) continuations with global side effects like IO or external resources makes it difficult to reason about the correctness of a program. Consider the following function:

```
fun div(body : () → ⟨out|e⟩ ()) : ⟨out|e⟩ () {
  print("<div>")
  body()
  print("</div>")
}
```

In Koka, all function types have three components: the input types, the output type, *and* their effect type. Here, the type of the `body` parameter, `() → ⟨out|e⟩ ()`, shows that `body` is a function that takes no arguments, returns a unit value (of type `()`), and it can have any side-effects `e` including the `out` effect for printing to the console. The full type of `div` shows that it uses no other effects besides the `out` effect (and `e`).

However, even with precise effect types, reasoning about the correctness of the higher-order function `div` remains difficult since it is *polymorphic* in the effects `e`. What if the effect row `e` includes exception-like effects? This can lead to a missing closing tag `"</div>"`. Similarly, effects for probabilistic choice or backtracking search that resume more than once would result in printing multiple closing tags.

We identify the concept of *control-flow linearity* to reason about this precisely. Our notion of control-flow linearity is formalized in Section 4, but informally we say,

> *A function is control-flow linear if it returns exactly once*

Similarly, a function is control-flow *abortive* if it never returns, control-flow *affine* if it returns at most once and *wild* if it may never return or return many times. We use "linearity" for the general concept subsuming linear, abortive, affine and wild expressions.

In languages with algebraic effects and handlers, like Koka, the *only* way to alter the control-flow is through the use of effect operations. Also, if a function only uses effect operations that are handled by linear handlers, then the function itself will be control-flow linear. That is, if we know the effects of a function and their linearity, we can reason about its control-flow linearity! Koka already tracks effects in the type system, we just need to slightly change the effect system to also statically track the control-flow linearity.

1. The main idea is to declare control-flow linearity as part of an effect type, as in the following effect declarations:

   ```
   effect abort exn { throw(e: exception): a }
   effect linear out { println(x : string): () }
   effect wild amb { flip(): bool }
   ```

   We also define the control-flow linearity of an entire row of effects as the join of their individual effects.

2. We use the general framework of qualified types [27–29] to propagate control-flow linearity constraints on effect types. For example, we can annotate `div` to guarantee that the body is well-behaved:

```
div : (body : () → ⟨out|e⟩ ()) → ⟨out|e⟩ ()
        with e ≤ linear
```

   This constrains all instantiations of `div` to satisfy the `e ≤ linear` predicate. Inside `div` we can now reason *locally* that body returns exactly once and that we print exactly one closing tag. The framework of qualified types is well understood and straightforward to implement as part of type inference.
3. Finally, we statically check that any particular handler implementation for an effect actually satisfies the control-flow linearity that the corresponding effect declares. We use a simple syntactic check to restrict the way in which a handler uses the captured continuation. While this check is simple, it is independent of the rest of our approach and can very easily be replaced by a more involved variant in the future.

2.2 Linear Effect Handlers

The following example illustrates the powerful combination of effect handlers and linear effect types. Suppose we have the primitive file operations:

```
fun fopen( path : string ) : ⟨fileio,exn⟩ fhandle
fun fread( h : fhandle )   : ⟨fileio⟩ string
fun fclose( h : fhandle )  : ⟨fileio,exn⟩ ()
```

To provide a safe abstraction over these operations, using linearity annotations, we can declare the following effect

```
effect linear file { read() : string }
```

and an effect handler that encapsulates the unsafe primitives:

```
fun with-file(path, action) {
  val fh = fopen(path)
  val result = handle(action) {
    read() → resume(fread(fh))
  }
  fclose(fh)
  result
}
```

The function first opens a file, executes `action` under a handler that allows reads from that file, and finally closes the file to then return the result of executing `action`. Our type system supports full type inference, and it infers the following type for the function `with-file`:

```
(string, () → ⟨file|e⟩ a) → ⟨fileio,exn|e⟩ a
  with e ≤ linear
```

The `e ≤ linear` constraint is automatically inferred because we define a handler for the `file` effect that was declared as linear. The type checker guarantees that any `action` passed to `with-file` only uses linear effects and thus is control-flow linear. Our implementation of `with-file` is safe: by construction, `action` always returns exactly once with a regular value; no exceptions or multiple returns are allowed!

The following instantiations are all rejected by the type checker since `e` will instantiate to either `⟨exn,file⟩` or to `⟨amb,file⟩`. Both do not satisfy the predicate since they contain non-linear effects.

```
with-file("foo.txt") {
  throw("ouch!") // rejected
  read().length
}
```

```
with-file("foo.txt") {
  flip()  // rejected
  read().length
}
```

The strength of effect handlers is that the full semantics of the effect operations are defined all together in the `handle` construct. That means we can *locally* reason inside `with-file` whether we use our external `fileio` resources correctly. For example, it is immediately apparent that the file handle `fh` does not escape its context. Additionally, the type system now guarantees that `action` is control-flow linear, which allows us to conclude that the file is closed exactly once.

2.3 Example: Heaps and Multiple Resumptions

Another example, which was the original motivation of this work, is to use heap allocated state and control-effects together. Let's assume the following two effect declarations, for now without linearity annotations:

```
effect amb { flip() : bool }
effect state⟨s⟩ { get() : s; set(v : s) : () }
```

The canonical handler implementation for ambiguity collects all results in a list and we can use it to produce boolean tables, where `amb { flip() && flip() }` returns the list `[True,False,False,False]`.

```
fun amb(action: () → ⟨amb|e⟩ a ): e list⟨a⟩ {
  handle(action) {
    return x → [x]
    flip()   → resume(True) + resume(False)
  }
}
```

As an efficient alternative to the usual state-passing implementation of the `state` effect [39], we might want to use a heap-allocated, mutable reference cell. The signature of such a handler looks like:

```
fun state(init: a, f: () → ⟨state⟨a⟩|e⟩ r ): e r
```

Now, consider the following program that uses both the state effect and the ambiguity effect.

```
amb {
  state(0) {
    flip()
    set( get() + 1 )
    get()
  }
}
```

According to the semantics of the handlers, the `state` effect is locally handled under `amb` and it should not be shared among the resumptions. Thus, the expected result is `[1,1]`. Unfortunately, the efficient implementation of `state` using a real heap means, we can now observe the previous effects of our earlier strand that already exited the `state` scope, and the final result is `[1,2]` instead!

Again, annotating the effect declaration for ambiguity with `wild` and changing the signature of `state` to

```
fun state(init: a, f: () → ⟨state⟨a⟩|e⟩ r ): e r
   where e ≤ affine
```

solves this problem. Since `flip` has the wild `amb` effect, our problematic example is statically rejected.

Note, that reordering the handlers to

```
state(0) {
  amb { ... }
}
```

is neither operationally problematic, nor rejected by our type system. The non-linear ambiguity effect is handled locally and does not violate the linearity assumptions of our state handler.

2.4 Checking Handler Linearity

The linearity annotations on effect declarations not only lead to a stricter type when inferring the type of handlers, but also require that any effect handler implementation needs to respect the declared control-flow linearity. That is, the declaration is a *promise*, and any handler needs to fulfill it. In particular,

1. handlers may only use effects that have a linearity lower or equal to the declared linearity of the handled effect; and
2. handlers have to use `resume` (which represents the continuation after the call to the effect operation) according to the declared linearity.

While the first property is very easy to check in a type system, for the latter there are many approaches with varying complexity. In this paper we use a simple *syntactic check* on each operation clause in a handler – in particular:

- linear: each operation clause must `resume` exactly once and in tail-call position (as in our `with-file` example);
- abort: each operation clause should never invoke resume, i.e. not refer to `resume` at all;
- affine: each operation clause must either resume once in a tail-call position or directly return a value;
- wild: no restrictions on the use of `resume`.

This poses a direct connection from syntactic restrictions on the use of `resume` in handlers to the control-flow linearity of the effect types. Together with the static type check, it guarantees that the declared linearity of an effect type is always respected by any specific handler implementation (see our proof in Section 4.2).

```
firstLine :: FilePath → IO_L ω ByteString
firstLine fp =
  do { f ← openFile fp
     ; (f, Unrestricted bs) ← readLine f
     ; closeFile f
     ; resturn bs }
```

Fig. 2. Example program in Linear Haskell.

The syntactic check is quite simple but in our experience already covers many practical instances. Importantly, our system of qualified effect types to check control-flow linearity is independent of the concrete check for linear resumption usage. Developing more sophisticated implementations is left for future research.

2.5 Linear Control-Flow vs. Linear Data-Flow

Contrast the previous `with-file` example with linear type systems based on linear logic [21, 22, 51]: such systems focus on the linear *usage* of specific resources (like the file handle). Control-flow linearity by itself does *not* guarantee that the file handle `fh` is used linearly. Instead, through the use of effect handlers, we encapsulate the access to the file handle inside the scope of the `with-file` handler. Then, in a second step, by means of control-flow linearity we can locally reason about the correct usage of external resources. In contrast, linear types guarantee the linear usage of resources, but do *not* express linear control-flow, per se. This is illustrated by an example of Bernardy et al. [6] in Linear Haskell (Figure 2). However, even though it guarantees that the file resource `f` is consumed linearly, the file might still not be closed if `readLine` throws an exception.

We are not proposing control-flow linearity as a replacement for linear types. Instead, control-flow linearity offers a different perspective on problems where traditionally linear type systems would be considered as a solution. Instead of focusing on how a function uses its arguments and restricting which function can be called, control-flow linearity focuses on the context requirements of a function and restricts the contexts in which the function can be called.

3 A Calculus of Qualified Effect Types

In this section, we give a formal definition of our effect system with qualified effects. We build on a polymorphic row-based effect system underlying the Koka language [39]. Extending the effect system with qualified types [27] only affects the type system and thus the operational semantics carries over unchanged. The type system and its properties has been presented before in a similar form [39] but we include it again here to make this article self contained – necessarily keeping the description short.

Our effect system with qualified effects is parametric in a join semi-lattice of annotations $(A, \leqslant)$ where elements $a \in A$ are called *effect annotations*. Without loss of generality,

we specialize our presentation to the control-flow linearity lattice $A = \{ \text{abort}, \text{linear}, \text{affine}, \text{wild} \}$ of Figure 1 with

$$\text{linear} < \text{affine} < \text{wild}, \quad \text{abort} < \text{affine}$$

and often refer to $a \in A$ as *linearity annotations*. The lattice structure is only necessary to achieve a form of subtyping relationship between different control-flow linearities. In particular, we require it to be a join-lattice to be able to compute the linearity of an arbitrary effect row by taking the join of its components. Our calculus is a conservative extension of the effect system presented by Leijen [39] in the sense that instantiating $A = \{ () \}$ yields the original calculus since all predicates are trivially satisfied.

3.1 Syntax

Figure 3a describes the syntax of expressions, types and kinds. For the formal presentation we require that not only the effect declarations but also the handlers are annotated with their linearity. In practice these linearity annotations will be inferred, but the source of these annotations and their semantic implications are external to the type system itself. We often omit the handler implementation h and for instance write a linear state handler as $\mathsf{handle}^{state}_{\mathsf{linear}}$. Most of the syntax immediately carries over from earlier presentations [39]. We assume that all effect operations take just one argument and use membership notation $op(x) \to e \in h$ to denote that h contains a particular operation clause.

Well-formed types are guaranteed through kinds k which we denote using a superscript, as in τ^k. Besides the usual kinds for value types $*$ and type constructors $\to$ we also have kinds for effect constants x and effect rows e. We omit kinds when immediately apparent or not relevant.

We use meta variables l for effect labels, a for linearity annotations, ϵ for effect rows, μ for effect variables and α for regular type variables. For the purposes of this paper effect labels can just be single constants but in general Koka allows type arguments too [39].

Effect types are defined as a row of effect labels l. A 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 an effect constant and are never polymorphic. By construction, effect types are either a *closed effect* of the form $\langle l_1, \ldots, l_n \rangle$, or an *open effect* of the form $\langle l_1, \ldots, l_n \mid \mu \rangle$. Effect rows are considered equivalent up to permutation of unequal labels as defined in Figure 3c. There exists a principal unification algorithm for such effect rows with possible duplicate labels and enables type inference [34, 39].

Qualified Effect Types To account for linearity annotations, we extend type schemes σ to include predicates on effect rows. The predicate $\epsilon \sqsubseteq a$ states that the effect row ϵ should have *at most* linearity a. Some examples of type schemes are:

$$\begin{array}{l} \mathit{int} \\ \forall \mu.\ (\mu \sqsubseteq \text{linear}) \Rightarrow () \to \mu\ \mathit{int} \\ \forall \mu.\ (\langle \mathit{out}, \mathit{exn} \mid \mu \rangle \sqsubseteq \text{affine}) \Rightarrow () \to \langle \mathit{out} \mid \mu \rangle\ \mathit{int} \end{array}$$

That is, a pure expression of type *int*, a function that produces an *int* potentially using linear effects μ and a function using affine effects μ and *out* and *exn*. As another example, we can write the type scheme of our example function `with-file` from the introduction as:

$$\forall \mu \sqsubseteq \text{linear}.\ (\mathit{string}, () \to \langle \mathit{file} \mid \mu \rangle\ \alpha) \to \langle \mathit{fileio} \mid \mu \rangle\ \alpha$$

Since all effects are statically annotated with their linearity, we can always simplify satisfiable predicates of the form $\epsilon \sqsubseteq a$ to the form $\mu \sqsubseteq a$ using the rules for linearity resolving in Figure 3d. For instance, we can resolve the predicate $\langle \mathit{out}, \mathit{exn} \mid \mu \rangle \sqsubseteq \text{affine}$ to $\mu \sqsubseteq \text{affine}$, since we know that $\mathsf{lin}(\mathit{out}) = \text{linear}$ and $\mathsf{lin}(\mathit{exn}) = \text{abort}$.

3.2 Operational Semantics

The dynamic semantics of algebraic effects and handlers immediately carries over unchanged from [39] but is included in Figure 3b for easier reference 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 H^l context is used for handlers. In particular, it evaluates down through any handlers that do *not* handle the effect l. This is used to express concisely that the *innermost handler* handles a particular operation.

Dot notation: Most of the definitions, lemmas and proofs in the remainder of this paper involve arguments about the evaluation context. For notational convenience we establish the following convention for evaluation contexts (representing the runtime stack): We use the right-associative operator $\cdot$ for context substitution where $\mathsf{E} \cdot e \doteq \mathsf{E}[e]$ and $e \cdot e' \doteq e(e')$. That is, we write $\mathsf{E}_1 \cdot \mathsf{E}_2 \cdot e \cdot v$ instead of the more common notation $\mathsf{E}_1[\mathsf{E}_2[e(v)]\,]$.

The first three reduction rules, (δ), (β), and (let) are standard rules of call-by-value evaluation. The remaining two rules evaluate handlers. Rule (return) applies the return clause of a handler when the argument is fully evaluated. The (handle) rule uses a H^l context to ensure by construction that an operation $op^l(v)$ is handled by the innermost handler for l. Evaluation continues with the expression e but besides binding the parameter x to v, also the *resume* variable is bound to the continuation: $\lambda y.\ \mathsf{handle}^l_h \cdot \mathsf{H}^l \cdot y$. Applying *resume* results in continuing evaluation at H^l with the supplied argument as the result. Moreover, the continued evaluation occurs again under the same handler h, effectively implementing *deep handlers* – as opposed to *shallow handlers* where the handler can change for each handled operation [30, 42].

3.3 Type Rules

Figure 4 presents the type system of Koka [39] extended with qualified effect types. A type environment Γ maps variables.

Syntax of Expressions:

Expressions	e	::=	$e(e)$	application
			$\mathsf{val}\ x\ =\ e;\ e$	binding
			$\mathsf{handle}^{l}_{a}\ \{\ h\ \}(e)$	handler
			v	value
Values	v	::=	$x \mid c \mid op \mid \lambda x.\ e$	
Clauses	h	::=	$\mathsf{return}\ x \to e$	
		|	$op(x) \to e;\ h$	$op \notin h$

Syntax of Types:

Types	τ^{k}	::=	α^{k}	type variable
		|	$c^{k}\langle \tau_1^{k_1}, \ldots, \tau_n^{k_n} \rangle$	$c\ ::\ (k_1, \ldots, k_n) \to k$
Kinds	k	::=	$*$	values
		|	e	effect rows
		|	x	effect constants
		|	$(k_1, \ldots, k_n) \to k$	type constructor
Predicates	p	::=	$\epsilon \sqsubseteq a$	single predicate
	P, Q	::=	$(p_1, \ldots, p_n)$	predicate set
Type scheme	σ	::=	$\forall \alpha^{k}.\ \sigma \mid P \Rightarrow \tau^{*}$	
Effect Annotations	a	::=	linear | abort	
		|	affine | wild	

Type Constants:

(), *bool*		::	$*$	unit, booleans
$(\_ \to \_\ \_)$		::	$(*, \mathsf{e}, *) \to *$	functions
$\langle\rangle$		::	e	empty effect
$\langle \_ \mid \_ \rangle$		::	$(\mathsf{x}, \mathsf{e}) \to \mathsf{e}$	effect extension
exn, *div*, ...		::	x	effect constants

Shorthands:

Effect labels	l	$\doteq$	c^{x}
Effect rows	ϵ	$\doteq$	τ^{e}
Effect row variables	μ	$\doteq$	α^{e}

(a) Syntax of expressions, types and kinds. Control-flow linearity related syntax is highlighted in *grey*.

Evaluation Contexts:

$$\begin{array}{lll} \mathsf{E} & ::= \square \mid \mathsf{E}(e) \mid v(\mathsf{E}) \mid \mathsf{val}\ x\ =\ \mathsf{E};\ e & \\ & \mid\ \mathsf{handle}^{l'}_{a}\ \{\ h\ \}(\mathsf{E}) & \\ \mathsf{H}^{l} & ::= \square \mid \mathsf{H}^{l}(e) \mid v(\mathsf{H}^{l}) \mid \mathsf{val}\ x\ =\ \mathsf{H}^{l};\ e & \\ & \mid\ \mathsf{handle}^{l'}_{a}\ \{\ h\ \}\ (\mathsf{H}^{l}) & \text{if } l \neq l' \end{array}$$

Reduction Rules:

$$\begin{array}{lll} (\delta) & c(v) \longmapsto \delta(c, v) & \\ & \quad \text{if } \delta(c, v) \text{ is defined} & \\ (\beta) & (\lambda x.\ e)(v) \longmapsto e[x \mapsto v] & \\ (let) & \mathsf{val}\ x\ =\ v;\ e \longmapsto e[x \mapsto v] & \\ (return) & \mathsf{handle}^{l}_{a}\ \{\ h\ \}(v) & \\ & \qquad \longmapsto e[x \mapsto v] & \\ & \quad \text{where } (\mathsf{return}\ x \to e) \in h & \\ (handle) & \mathsf{handle}^{l}_{a}\ \{\ h\ \} \cdot \mathsf{H}^{l} \cdot op^{l}(v) & \\ & \qquad \longmapsto e[x \mapsto v,\ resume \mapsto r)] & \\ & \quad \text{where } (op(x) \to e) \in h & \\ & \quad \text{and } r\ =\ \lambda y.\ \mathsf{handle}^{l}_{a}\ \{\ h\ \} \cdot \mathsf{H}^{l} \cdot y & \end{array}$$

(b) Operational Semantics.

$$\epsilon \cong \epsilon \quad [\textsc{eq-refl}] \qquad \frac{\epsilon_1 \cong \epsilon_2}{\langle l \mid \epsilon_1 \rangle \cong \langle l \mid \epsilon_2 \rangle}\ [\textsc{eq-head}]$$

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

$$\frac{l_1 \neq l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon \rangle \rangle \cong \langle l_2 \mid \langle l_1 \mid \epsilon \rangle \rangle}\ [\textsc{eq-comm}]$$

(c) Row equivalence

$$\frac{}{\Vdash \langle\rangle \sqsubseteq \phi}\ [\textsc{res-empty}] \qquad \frac{}{\Vdash \epsilon \sqsubseteq \top}\ [\textsc{res-top}]$$

$$\frac{\Vdash \mathsf{lin}(l) \leqslant \phi \qquad \Vdash \epsilon \sqsubseteq \phi}{\Vdash \langle l \mid \epsilon \rangle \sqsubseteq \phi}\ [\textsc{res-hd}]$$

(d) Linearity resolving.

Fig. 3. Syntax, operational semantics, row equivalence and linearity resolving.

Thus, if Γ' equals $\Gamma, x : \sigma$, then $\Gamma'(x) = \sigma$ and $\Gamma'(y) = \Gamma(y)$ for any $x \neq y$. The judgment form $P \mid \Gamma \vdash e : \tau \mid \epsilon$ states that under environment Γ and given the predicates in P the expression e has type τ with possible effects ϵ. Informally speaking, the type system is a Hindley/Milner-style polymorphic type system [26, 43] extended with (i) a polymorphic row-based effect system [35, 39] and (ii) qualified types [27, 29]. Interestingly, the predicates in our qualified type system range over effect rows, not types. Even though it is not syntax directed, we can think of Γ as being inherited and P, τ and ϵ as synthesized components of the typing judgment.

The first four type rules are quite standard and just extend the rules by Jones [29] with effect rows. The rule Var derives the type of a variable x with an arbitrary effect ϵ and under arbitrary predicates P. Operations op and constants c are also looked up using the Var-rule assuming the types of those are part of the initial environment Γ_0. The Lam rule is similar

Standard Rules:

$$\frac{P \mid \Gamma \vdash e_1 : \sigma \mid \epsilon \quad Q \mid \Gamma, x : \sigma \vdash e_2 : \tau \mid \epsilon}{P,\ Q \mid \Gamma \vdash \mathsf{val}\ x = e_1;\ e_2 : \tau \mid \epsilon}\ [\textsc{Let}] \qquad \frac{\Gamma(x) = \sigma}{P \mid \Gamma \vdash x : \sigma \mid \epsilon}\ [\textsc{Var}]$$

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

Qualified Effect Types and Polymorphism:

$$\frac{P,\ Q \mid \Gamma \vdash e : \tau \mid \langle\rangle \quad \overline{\alpha} \notin \mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(P)}{P \mid \Gamma \vdash e : \forall \overline{\alpha}.\, Q \Rightarrow \tau \mid \epsilon}\ [\textsc{Gen}] \qquad \frac{P \mid \Gamma \vdash e : \forall \overline{\alpha}.\, Q \Rightarrow \tau \mid \epsilon \quad P \Vdash Q}{P \mid \Gamma \vdash e : \tau[\overline{\alpha} \mapsto \overline{\tau}] \mid \epsilon}\ [\textsc{Inst}]$$

Effect Handling:

$$\frac{\begin{array}{l} S(l) = (a,\ \{op_1,\ \ldots,\ op_n\}) \cdot \mid \Gamma \vdash op_i : \tau_i \to \langle l \rangle\ \tau_i' \mid \langle\rangle \\ P \mid \Gamma, x : \tau \vdash e_r : \tau_r \mid \epsilon \qquad P \mid \Gamma, \mathit{resume} : \tau_i' \to \epsilon\ \tau_r,\ x_i : \tau_i \vdash e_i : \tau_r \mid \epsilon \\ P \Vdash \epsilon \sqsubseteq a' \qquad a' \leqslant a \qquad P \mid \Gamma \vdash e : \tau \mid \langle l \mid \epsilon \rangle \end{array}}{P \mid \Gamma \vdash \mathsf{handle}^{l}_{a'}\{\ op_1(x_1) \to e_1;\ \ldots;\ op_n(x_n) \to e_n;\ \mathsf{return}\ x \to e_r\ \}(e) : \tau_r \mid \epsilon}\ [\textsc{Handle}]$$

Fig. 4. Type rules - handling an effect adds a linearity predicate on the remaining effects.

to the Var rule in that it can freely assume any effect ϵ for the result since the evaluation of a lambda is a value. The predicates P which are assumed for type checking the body propagate up to the lambda abstraction. This rule also shows how the effect derived for the body of a lambda ϵ' shifts to the effectful function type $\tau_1 \to \epsilon'\ \tau_2$. Rule App derives an effect ϵ requiring that its premises derive the same effect as the function effect. Additionally, all involved predicates for type-checking the function and the argument are required to unify. As usual in a polymorphic Hindley/Milner style type system, the rule Let type checks the bound expression against a type scheme σ. The derived effects ϵ are required to be the same for bound expression e_1 and body e_2.

Rules Inst and gen instantiate and generalize types. Instead of having separate introduction and elimination rules for qualified types [27], we combine the standard rule for instantiation with elimination and the rule for generalization with introduction of qualified effect types. For instantiation, rule Inst requires Q to be entailed by P. That is, P should at least contain all linearity constraints guaranteed by Q. Generalization requires the derived effect to be total. This immediately corresponds to value restriction in ML [39].

The last and most interesting rule handle types effect handlers. We assume that all operations have unique names, such that given the operation names, we can uniquely determine to which effect l they belong. We also assume that every effect declaration is annotated with a linearity a. We require a *signature environment* S in which we can lookup the effect signature of some effect label l, giving us the effect annotation and the set of effect operations: $S(l) = (a,\ \{\ op_1,\ op_2,\ \ldots\ \})$. As an example, assuming the effect declaration

```
effect linear state⟨s⟩ { get(): s; put(v : s): () }
```

then $S(\mathit{state}) = (\text{linear},\ \{\ \mathit{get},\ \mathit{put}\ \})$. We use $\mathsf{lin}(l)$ to immediately access the linearity component in the signature environment. That is, $\mathsf{lin}(\mathit{state}) = \text{linear}$.

The rule handle requires that all operations in the signature $S(l)$ are part of the handler, and we reject handlers that do not handle all operations of the effect l. By means of

$$S(l) = (a,\ \{op_1,\ \ldots,\ op_n\}) \cdot \mid \Gamma \vdash op_i : \tau_i \to \langle l \rangle\ \tau_i' \mid \langle\rangle$$

we look up the type of every operation clause op_i in the environment as $\tau_i \to \langle l \rangle\ \tau_i'$. We then type check the return clause ϵ_r and all operation clauses $op_i(x_i) \to e_i$. All clauses are expected to type check under the same linearity predicates P using the same effects ϵ and resulting in the same return type τ_r. To type check the operation clauses, we extend the environment with the continuation $\mathit{resume} : \tau_i' \to \epsilon\ \tau_r$ and the operation argument $x_i : \tau_i$. Finally, we require that the linearity constraint $\epsilon \sqsubseteq a'$ is entailed by P and that the linearity declared on the handler a' is less than the linearity declared on the effect signature a. Again, the type system does not perform any checks here to assert that the handler conforms to linearity a'. Thus, annotating the handler with the correct linearity is external to our type and effect system.

Summarizing, a handler respects the linearity requirements a of the handled effects if itself is annotated with at most a and the remaining effects in ϵ satisfy the handler's requirement $\epsilon \sqsubseteq a'$. In particular, the handler itself might only use effects in ϵ to implement the operation clauses.

As described in earlier work [36], type inference is straightforward based on Hindley-Milner [26, 43], and is sound and complete with respect to the declarative type rules. Inference extends naturally with qualified types as shown by Jones [27].

3.4 General Properties of the Type System

It is shown in [39] using techniques from [53] that well-typed effectful programs cannot go *wrong*. From the theory of qualified types [27, 29] it follows directly that the original proof is naturally applicable in the presence of qualified types too.

Theorem 1. (*Semantic soundness*)
If $P \mid \ . \ \vdash \ e \ : \ \tau \mid \epsilon$ and P *satisfiable* then either $e \Uparrow$ or we have $e \ \longmapsto^* \ v$ where $P \mid \ . \ \vdash \ v \ : \ \tau \mid \epsilon$.

where we use the notation $e \Uparrow$ for divergence. An important property of our system is that effects are tracked faithfully [39]:

Lemma 1. (*Effects are meaningful*)
If $P \mid \Gamma \ \vdash \ \mathsf{H}^l \ \cdot \ op^l(v) \ : \ \tau \mid \epsilon$ and P *satisfiable*, then $l \ \in \epsilon$.

This is a powerful lemma 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 `exn` effect, it will never throw an exception.

3.5 Control-Flow Linearity Properties of the Type System

Equipped with our type system we can establish some terminology.

Definition 1. (*Effect Linearity*)
We say an expression e **has linearity** a if the following holds: $P \mid \Gamma \ \vdash \ e \ : \ \tau \mid \epsilon$, P is *satisfiable* and $P \ \Vdash \ \epsilon \ \sqsubseteq \ a$.

An expression e is *effect linear* if e has linearity linear.

Definition 2. (*Well Typed Contexts*)
We call a context E **well typed** for some expression e if $P \mid \Gamma \ \vdash \ \mathsf{E}[e] \ : \ \tau \mid \epsilon$ and P is satisfiable.

The terminology allows us to state an important property of our type system, related to control-flow linearity:

Lemma 2. (*Soundness of Linearity*)
If e is well typed, then $e \ \longmapsto^* \ \mathsf{E}' \ \cdot \ \mathsf{handle}^l_a \ \cdot \ \mathsf{H}^l \ \cdot \ op^l(x)$ implies, that for all $\mathsf{handle}^{l'}_{a'} \ \in \mathsf{H}^l$ it follows $a \ \leqslant \ a'$.

Proof. We proceed by induction over the structure of H^l. If $\mathsf{H}^l \ = \ \mathsf{handle}^{l'}_{a'} \ \cdot \ \ldots$, then by Handle for some effect ϵ' the predicate $\epsilon' \ \sqsubseteq \ a'$ must be entailed by P. By lemma 1, $l \ \in \epsilon'$ and thus $\mathsf{lin}(l) \ \leqslant \ a'$. Since Handle, requires $a \ \leqslant \ \mathsf{lin}(l)$ we have $a \ \leqslant \ a'$. □

This lemma expresses a runtime guarantee: Each l-handler that is applied during the evaluation of e respects the linearity constraints of all l'-handlers it dynamically encloses.

We believe the effect system presented in this section has a good "power-to-weight ratio". The small operational semantics of effect handlers is unchanged, and the main ingredients of the type system, namely Hindley-Milner style type inference, row-based effect types, and qualified types, are all well understood and easy to implement.

4 Control-Flow Linearity, Formally

In this section, we capture our notion of control-flow linearity more formally. In particular, we say *an expression is control-flow linear if it can be evaluated using a restricted reduction relation of* linear reductions ($\longmapsto_1$) as defined by the rules in Figure 5. The linear reduction rules capture the essence of what one intuitively understands as control-flow linearity. Later we prove that any expression that is typed as effect linear can be reduced using linear reduction, and thus is control-flow linear. In this paper, we formally exercise the relationship between linear effects and control-flow linearity. Extending the treatment to abortive and affine effects is straightforward and doesn't pose any additional challenges. The rules of the linear reduction relation capture two important aspects of linearity. Reducing a linear expression only involves either context preserving reductions (Kong1), or reductions of effect operations where the handler is guaranteed to resume exactly once (Handle1).

Context preservation. The rule Kong1 can be thought of as the "boyscout rule": leave your context as you have found it. Pure expressions or expressions where effects are handled locally by handlers within e are reduced using Kong1 without modifying the context. To be able to precisely define context preservation, we need to separate the expression e from the context E it is evaluated in. To this end, we introduce a special evaluation context $\#[\mathsf{E}]$ that acts as *boundary marker*. We thus write $\mathsf{E} \ \cdot \ \# \ \cdot \ e$ to separate e from its context E. The boundary marker only occurs in linear reductions and there are no rules to introduce or eliminate it. In particular, it does not occur in any contexts E or H^l as defined in Figure 3b.

Effect operations resume exactly once. The second rule Handle1 models the requirement that handlers should implement effect operations by resuming exactly once. Note that this is only a requirement for handlers outside of $\#[\mathsf{E}]$, modeled by the handler context $\mathsf{H}^l_\# \ \doteq \ \mathsf{H}^l \ \cdot \ \# \ \cdot \ \mathsf{H}^l$ which is a handler context that does not contain l handlers, but contains exactly one boundary marker. Effect operations handled by handlers within e are instead reduced using Kong1 and the regular (*handle*) rule. To handle an effect operation, we capture the continuation and proceed with normal evaluation using the reduction relation $\longmapsto$. However, we require the reduction to end in a continuation call. The freshness condition on k requires that k does not occur anywhere in the expression that is being reduced. By choosing a fresh k

$$\frac{e \longmapsto e'}{\mathsf{E} \cdot \# \cdot e \longmapsto_1 \mathsf{E} \cdot \# \cdot e'}\ [\textsc{Kong1}] \qquad \frac{op(x) \to e \in h \qquad \mathsf{k}\ \textit{fresh} \qquad \mathsf{k} \notin \mathsf{E}',\ w \quad \mathsf{E} \cdot e[x \mapsto v,\ \textit{resume} \mapsto \mathsf{k}] \longmapsto^* \mathsf{E}' \cdot \mathsf{k}(w)}{\mathsf{E} \cdot \mathsf{handle}^l\ \{\ h\ \} \cdot \mathsf{H}^l_\# \cdot op^l(v) \longmapsto_1 \mathsf{E}' \cdot \mathsf{handle}^l\ \{\ h\ \} \cdot \mathsf{H}^l_\# \cdot w}\ [\textsc{Handle1}]$$

Fig. 5. Linear Reductions.

and requiring that it does not occur in the result of the evaluation step, we operationally guarantee that the continuation can only be called exactly once.

Definition 3. (*Control-Flow Linearity*)
We call an expression e **control-flow linear** in a context E if and only if it can be evaluated using linear reduction, $\mathsf{E} \cdot \# \cdot e \longmapsto_1^* \mathsf{E}' \cdot \# \cdot v$, or it diverges, $\mathsf{E} \cdot \# \cdot e \Uparrow$.

Here, $\longmapsto_1^*$ is the reflexive, transitive closure of $\longmapsto_1$. The relation $\longmapsto_1$ has a few interesting properties. It guarantees that the evaluation context always contains exactly one occurrence of #, the boundary between the linear expression and its context. Also, # can never occur in a term evaluated by $\longmapsto^*$. However, the evaluation using $\longmapsto_1$ can get stuck: The premise of Handle1 requires the expression to evaluate to a resumption of the very same continuation k. It will get stuck if it evaluates to either a value (by ignoring the continuation) or to a resumption of another continuation k' which has been introduced in a previous application of Handle1 (e.g. resuming more than once). These are exactly the two cases that lead to non-linear control-flow.

Coming up with the right operational definition of control-flow linearity was surprisingly tricky. For example, requiring the context E to be fixed as in $\mathsf{E} \cdot e \longmapsto^* \mathsf{E} \cdot v$ is both very restrictive (the context is not allowed to change at all) and wrong: during the evaluation of e to v the context can arbitrarily be duplicated, discarded, reconstructed – as long as it is equal in the end. Also, only restricting every continuation to be resumed at most once is wrong too, as inside e arbitrary effects are allowed (as long as they are handled locally in e).

Some further interesting properties are:

Lemma 3. (*Completeness of linear reduction*)
$\mathsf{E} \cdot \# \cdot e \longmapsto_1^* \mathsf{E}' \cdot \# \cdot v$ implies $\mathsf{E} \cdot e \longmapsto^* \mathsf{E}' \cdot v$.

This lemma states that we can also use normal reduction $\longmapsto^*$ instead of linear reduction, while resulting in the same context E' and value v.

Lemma 4. (*Linear reduction preserves pure contexts*)
We define *pure contexts* F [3] which only contain applications and bindings, but no handlers as:

$$\mathsf{F} ::= \square \mid \mathsf{F}(e) \mid v(\mathsf{F}) \mid \mathsf{val}\ x\ =\ \mathsf{F};\ e$$

Now, if $\mathsf{E} \cdot \mathsf{F} \cdot \# \cdot e \longmapsto_1^* \mathsf{E}' \cdot e'$ then:

a. $\mathsf{E}' = \mathsf{E}'' \cdot \mathsf{F}$ and,
b. for any other well-typed pure context F', we have $\mathsf{E} \cdot \mathsf{F}' \cdot \# \cdot e \longmapsto_1^* \mathsf{E}'' \cdot \mathsf{F}' \cdot \# \cdot e'$.

This is a powerful lemma as it captures the notion that a control-flow linear expression cannot modify its enclosing pure context up to its first handler. Moreover, no handler can capture the pure context F since the lemma is stated parametrically over any other context F'.

Our presentation of linear reduction in general, and the Handle1 rule in particular is specialized to the setting of algebraic effects and handlers. However, the concept is very general and can easily be translated to other control operators like shift/reset [11, 17, 39, 48].

4.1 Syntactic Linearity

As we have seen earlier, it is transparent to our type system *how* handlers are annotated with their linearity. All annotations could be provided by the user without any further automated checks. While this is very flexible, to state our main theorem precisely we require handlers to only be annotated as linear if they are *syntactically linear*. Syntactic linearity was defined informally in Section 2.4, but now we can define it formally as follows.

Definition 4. (*Syntactic Linearity*)
We call a handler syntactically linear if all operations have the shape $(op(x) \to \textit{resume}(e)) \in h$ with $\textit{resume} \notin \mathsf{fv}(e)$.

Syntactically abortive and affine handlers can be defined analogously.

4.2 Effect Linearity Implies Control-Flow Linearity

We are now ready to state our main theorem. Under the requirement that only syntactically linear handlers are annotated with linear:

Theorem 2. (*Effect linearity is sound*)
For any expression e and a corresponding well typed context E, if $P \mid \Gamma \vdash \mathsf{E} \cdot e : \tau \mid \epsilon$ and $P \Vdash \epsilon \sqsubseteq \mathsf{linear}$ then e is control-flow linear in E (Definition 3)

Proof. (Of Theorem 2). For any expression e and a corresponding well typed context E, where $P \mid \Gamma \vdash \mathsf{E} \cdot e : \tau \mid \epsilon$ and $P \Vdash \epsilon \sqsubseteq \mathsf{linear}$ we need to show that e is control-flow linear under E (Definition 3). By soundness of our type system (Theorem 1) we have that either $\mathsf{E} \cdot e \longmapsto^* v$ or it diverges $\mathsf{E} \cdot e \Uparrow$. We can always choose a point in the reduction sequence such that

$$\underbrace{\mathsf{E} \cdot e \longmapsto^* \mathsf{E}' \cdot w}_{\text{should reduce linearly}} \longmapsto^* v$$

or where it diverges before or after evaluating a subexpression to w. Of course there usually are many intermediate

Extended Syntax:

Kinds	k	::=	...	
		|	a	control-flow linearity
Predicates	p	::=	$\ldots \mid \psi \leqslant a$	

Extended Type Constants:

Type constants	...			
	$a, \ldots$	::	a	lin. annotation constants
	$\langle \_ \mid \_ \rangle$	::	(a, x, e) → e	annotated effect extension

Extended Shorthands:

Linearity variable	φ	$\doteq$	α^{a}
Linearity type	ψ	$\doteq$	$a \mid \varphi$
Short hand	$\langle l \mid \epsilon \rangle$	$\doteq$	$\langle \varphi\ l \mid \epsilon \rangle$ for some fresh φ

Modified linearity resolving:

$$\frac{\Vdash \psi \leqslant a \quad \Vdash \epsilon \sqsubseteq a}{\Vdash \langle \psi\ \tau \mid \epsilon \rangle \sqsubseteq a}\ [\textsc{lin-hd}]$$

Updated linearity polymorphic handle rule:

$$\frac{\begin{array}{l} S(l) = \{op_1, \ldots, op_n\} \quad \cdot \mid \Gamma \vdash op_i : \tau_i \to \langle \_\ l \rangle\ \tau_i' \mid \langle\rangle \\ P \mid \Gamma, x : \tau \vdash e_r : \tau_r \mid \epsilon \quad P \mid \Gamma, resume : \tau_i' \to \epsilon\ \tau_r,\ x_i : \tau_i \vdash e_i : \tau_r \mid \epsilon \\ P \Vdash \epsilon \sqsubseteq a \qquad P \mid \Gamma \vdash e : \tau \mid \langle\ a\ l \mid \epsilon \rangle \end{array}}{P \mid \Gamma \vdash \mathsf{handle}^l_a\{\ op_1(x_1) \to e_1;\ \ldots;\ op_n(x_n) \to e_n;\ \mathsf{return}\ x \to e_r\ \}(e) : \tau_r \mid \epsilon}\ [\textsc{Handle}]$$

Fig. 6. Extension to support linearity polymorphism.

points where the reduction results in a value but we can show there is a particular one that is the result of reducing e linearly, i.e. $\mathsf{E} \cdot \# \cdot e \longmapsto_1^* \mathsf{E}' \cdot \# \cdot w$ (or where it diverges), which proves control-flow linearity of e under E.

We prove this using induction over the reduction steps considering the possible positions of the boundary marker #. In particular, for each step, or sequence of steps, in the reduction sequence $\mathsf{E} \cdot e \longmapsto^* \mathsf{E}' \cdot e'$, we show that we can also reduce linearly as

$$\mathsf{E}_1 \cdot \# \cdot \mathsf{E}_2 \cdot e \longmapsto_1 \mathsf{E}_1' \cdot \# \cdot \mathsf{E}_2' \cdot e'$$

where $\mathsf{E} = \mathsf{E}_1 \cdot \mathsf{E}_2$ and $\mathsf{E}' = \mathsf{E}_1' \cdot \mathsf{E}_2'$. We are done when reducing to a value as $\mathsf{E} \cdot \# \cdot v$.

case application If the initial reduction step is application, we have:

$$\mathsf{E} \cdot (\lambda x.\ e)\ v \longmapsto \mathsf{E} \cdot e[x \mapsto v] \quad \textbf{(1)}$$

There are two cases for the boundary marker. We are done, if $\mathsf{E} \cdot (\lambda x.\ e) \cdot \# \cdot v$: the initial expression is evaluated linearly and we pass it on as an argument. Otherwise, we have $\mathsf{E} = \mathsf{E}_1 \cdot \mathsf{E}_2$ and we can use congruence to show:

$$\begin{array}{l} \mathsf{E}_1 \cdot \# \cdot \mathsf{E}_2 \cdot (\lambda x.\ e)\ v \\ \longmapsto_1 \{\ \textsc{Kong1},\ (1)\ \} \\ \mathsf{E}_1 \cdot \# \cdot \mathsf{E}_2 \cdot e[x \mapsto v] \end{array}$$

Note, that since the original reduction (1) also used congruence, thus it is sound to replace this reduction step with linear congruence. The cases for δ, let, and return are similar.

case handler With an initial reduction step for the Handle rule, we have

$$\begin{array}{l} \mathsf{E} \cdot \mathsf{handle}^l_a\{h\} \cdot \mathsf{H}^l \cdot op(v) \\ \longmapsto \quad \textbf{(2)} \\ \mathsf{E} \cdot e[x \mapsto v,\ resume \mapsto r] \end{array}$$

where $op(x) \to e \in h$ and $r = \lambda y.\ \mathsf{handle}^l_a\{h\} \cdot \mathsf{H}^l \cdot y$. Again, we consider two cases for the position of the boundary marker. If it occurs outside the handle context, with $\mathsf{E} = \mathsf{E}_1 \cdot \mathsf{E}_2$, we can apply congruence directly as before:

$$\begin{array}{l} \mathsf{E}_1 \cdot \# \cdot \mathsf{E}_2 \cdot \mathsf{handle}^l_a\{h\} \cdot \mathsf{H}^l \cdot op(v) \\ \longmapsto_1 \{\ \textsc{kong1},\ (2)\ \} \\ e[x \mapsto v,\ resume \mapsto r] \end{array}$$

Otherwise, the boundary marker must be part of the captured handler context itself, as in:

$$\mathsf{E} \cdot \mathsf{handle}^l_a \cdot \mathsf{H}^l_1 \cdot \# \cdot \mathsf{H}^l_2 \cdot op^l(v)$$

Proving this case is more involved. First, Lemma 1 gives us that $l \in \epsilon$. By premise $P \Vdash \epsilon \sqsubseteq$ linear and linearity resolving we have $\mathsf{lin}(l) \leqslant$ linear. Due to the type rule Handle we also know $a \leqslant \mathsf{lin}(l)$ and thus $a \leqslant$ linear. Hence, the handler must itself be linear and fulfill the syntactic linearity requirements, which gives us:

$$op(x) \to resume(e) \in h, \qquad resume \notin \mathsf{fv}(e) \quad \textbf{(3)}$$

With $\mathsf{H}^l = \mathsf{H}_1^l \cdot \mathsf{H}_2^l$ we would like to apply the Handle1 rule. We can first derive:

$$\begin{array}{l} \mathsf{E} \cdot \mathsf{handle}_a^l\{h\} \cdot \mathsf{H}_1^l \cdot \# \cdot \mathsf{H}_2^l \cdot op(v) \\ \longmapsto \{\ (2),\ (3)\ \} \\ \mathsf{E} \cdot (resume(e))[x \mapsto v,\ resume \mapsto r] \\ = \{\ resume \notin \mathsf{fv}(e)\ \} \\ \mathsf{E} \cdot r \cdot e[x \mapsto v] \end{array}$$

From the Handle type rule, we know $resume(e)$ is linear, and thus by the App rule e is linear too. Using this, we can invoke the induction hypothesis to show:

$$\begin{array}{l} \mathsf{E} \cdot r \cdot \# \cdot e[x \mapsto v] \\ \longmapsto_1^* \{\ \text{induction}\ \} \\ \mathsf{E}'' \cdot \# \cdot w \\ = \{\ \text{Lemma 4 (a)}\ \} \\ \mathsf{E}' \cdot r \cdot \# \cdot w \end{array}$$

Applying Lemma 4 (b), we can now conclude we can do the same reduction by using an abstract pure context k instead of using the concrete r context, and derive

$$\mathsf{E} \cdot \mathsf{k} \cdot \# \cdot e[x \mapsto v] \longmapsto_1^* \mathsf{E}' \cdot \mathsf{k} \cdot \# \cdot w$$

Using completeness of linear reduction (Lemma 3) we can derive our needed premise:

$$\mathsf{E} \cdot \mathsf{k} \cdot e[x \mapsto v] \longmapsto^* \mathsf{E}' \cdot \mathsf{k} \cdot w$$

Together with (3), this fulfills all needed premises to apply the Handle1 rule, and we finally derive

$$\begin{array}{l} \mathsf{E} \cdot \mathsf{handle}_a^l\{h\} \cdot \mathsf{H}_1^l \cdot \# \cdot \mathsf{H}_2^l \cdot op(v) \\ \longmapsto_1 \\ \mathsf{E} \cdot \mathsf{handle}_a^l\{h\} \cdot \mathsf{H}_1^l \cdot \# \cdot \mathsf{H}_2^l \cdot w \end{array}$$

□

This concludes our proof of soundness of effect linearity. While it appears involved, it is a good example of how to reason about control-flow linearity of an expression. In particular, it shows how the boundary marker # is used to delimit the linear expression from its context and how Lemma 4 can be used to move pure contexts in and out of our linearity reasoning.

5 Type System Extensions

The qualified effect system presented in the Section 3 only requires simple extensions to an already existing type and effect system. Yet, it already provides powerful guarantees about control-flow linearity. In this section, we continue to explore the design space of a qualified effect system.

5.1 Control-Flow Linearity Polymorphism

Sometimes, fixing the linearity of an effect up front on declaration of the effect is too restrictive. For instance, we might want to type check programs like:

$$\mathsf{handle}_{\mathsf{linear}}^{amb} \cdot \mathsf{handle}_{\mathsf{linear}}^{state} \cdot flip()$$

In the previous system this is not possible since the effect declaration of *amb* fixes its linearity to be wild. However, a handler implementation might choose to implement the *amb* effect linearly to be able to cooperate with *state*.

In this section, we show a generalization of linearity predicates on effect rows by introducing linearity polymorphism. Instead of annotating effect *declarations* with the required linearity, effect operations are *polymorphic* in the linearity. The actual linearity then is determined by particular effect handler implementations – while potentially being subject to predicates and linearity constraints. This allows us to assign types to programs like the above example.

Figure 6 extends the syntax of kinds and predicates to account for *linearity variables* φ. Since the linearity is no longer declared as part of an effect, we track it individually for each effect label in an effect row: effect extension $\langle \_ \mid \_ \rangle$ is replaced by annotated effect extension $\langle \_\ \_ \mid \_ \rangle$ and we use the shorthand $\langle l \mid \epsilon \rangle$ to mean $\langle \varphi\ l \mid \epsilon \rangle$ for some fresh linearity variable φ. Effect operations op_i are now assumed to have the type scheme:

$$op_i\ :\ \forall \varphi\ \mu.\ \tau \to \langle \varphi\ l \mid \mu \rangle\ \tau'$$

As before, they are polymorphic in the effect row μ. However, they are also polymorphic in the linearity φ of effect type l.

In addition to predicates on effect rows, we now also include predicates on linearity types ψ. Take the following example predicate set:

$$\{\ \langle \varphi_1\ state,\ \mathsf{linear}\ file \mid \mu \rangle\ \sqsubseteq\ \mathsf{affine}\ \}$$

Using the modified linearity resolving from Figure 6 we can simplify the predicates to:

$$\{\ \varphi_1\ \leqslant\ \mathsf{affine},\ \mathsf{linear}\ \leqslant\ \mathsf{affine},\ \mu\ \sqsubseteq\ \mathsf{linear}\ \}$$

Also discharging the predicate $\mathsf{linear} \leqslant \mathsf{affine}$ on linearity constants, we finally obtain

$$\{\ \varphi_1\ \leqslant\ \mathsf{affine},\ \mu\ \sqsubseteq\ \mathsf{linear}\ \}$$

Figure 6 also updates the Handle type rule accordingly. Instead of looking up the linearity using the signature environment, the body e is now immediately type checked against the effect row $\langle a\ l \mid \epsilon \rangle$. In consequence all linearity annotations of unhandled uses of effect l in e are required to unify with the linearity annotation a on the handler.

6 Discussion

While algebraic effects and effect handlers seem to complicate reasoning about control flow at first sight, we embrace the concepts and use them to solve the problem. In particular, effect systems with support for effect handlers are centered around two important concepts:

1. Effect types on a function inform the caller about the possible side effects the function might have on the world. In particular, if a certain effect is absent in the effect type, the caller can rest assured that the function will not use that effect.

2. Effect handlers allow to factor code with control effects into two separate components. Code that uses effect operations, and handler definitions that specify the semantics of the effect operations. In consequence, reasoning about the correctness of the implementation of effect operations can often be localized to the effect handler.

We extend the effect system with *qualified effect types* and use predicates to collect further information about effect types. This effectively groups effects into separate effect classes. In particular:

- Instead of reasoning about the absence of a single effect, we can now reason about the absence of a *whole class* of effects. Importantly, this extends to the case of effect polymorphic expressions.
- At the same time, handler definitions can use predicates to restrict the handled function to only use a certain class of effects. This restriction is essential when reasoning about the handler implementation locally in order to verify whether the handler itself satisfies some predicate.

The choice of predicates and the particular classes is arbitrary. In this paper, we focused on grouping effects according to their linearity. We reduced the problem of verifying that an expression fulfills a certain property to two smaller problems: does the expression only use effects of a certain class? Does every handler for effects in that class maintain the desired property?

7 Related Work

Algebraic effects [45] and their extension with handlers [46, 47] can express many control-flow mechanisms in programming languages without needing to extend the compiler or language. Examples are iterators, async-await, concurrency, exceptions, etc.[12, 25, 30, 38, 54]. Recently, there are various implementations of effect handlers, either embedded in languages like Haskell [30, 54], Java [9], Scala [8], or C [37], or built into a language, like Eff [4], Links [25], Frank [42], Koka [39], and Multi-core OCaml [12, 52].

Multi-core OCaml [13, 14] implements a restricted form of algebraic effects and handlers where every continuation can only be resumed at most once; i.e. all effects are affine. This is partially done for efficiency, but also because ML has global state, it is generally unsafe to use multiple resumptions anyhow. This is also observed in other implementations of algebraic effects and handlers embedded in imperative host systems, like the Effekt library for Java and Scala [8, 9], where the library must be written carefully to not itself use mutation where multiple resumptions might interact.

Implementing monadic regions with freer monads, Kiselyov and Ishii [31] remark that all possible effects `r` of a `Eff r` computation "need to be checked whether the effect is known to be benign". The present paper can be seen as an attempt of formalizing the notion of "being benign" as being control-flow linear and tracking this well-behavedness in the type system.

The Scheme language always supported delimited continuations and also has struggled with initialization- and finalization for continuations that exited early or resumed more than once. The `unwind-protect` in Scheme is like a `finally` clause, while `dynamic-wind` is like `initially` / `finally` with a pre- and postlude [10, 16, 19, 44]. Sitaram [49] describes how the standard `dynamic-wind` is not good enough:

> While this may seem like a natural extension of the first-order unwind-protect to a higher-order control scenario, it does not tackle the pragmatic need that unwind-protect addresses, namely, the need to ensure that a kind of 'clean-up' happens only for those jumps that **significantly exit** the block, and not for those that are a **minor excursion**. The crux is identifying which of these two categories a jump falls into.

Interestingly, this is exactly what is addressed by our notion of control-flow linearity where "significant exit"s are affine (or abort), while "minor excursions" are linear operations.

Hieb et al. [24] proposes *control-filters* which are invoked when they are captured as part of a continuation. They are more expressive than `dynamic-wind` in that they can also modify or replace the captured continuation. However, like `dynamic-wind` they are a dynamic solution without any static guarantees about control-flow linearity.

Many modern languages support a fixed set of control-flow operations which are usually linear, with the exception of exceptions. Just for the single exception effect, most languages provide a range of special constructs to guarantee resource cleanup on exceptions, like `finally` statements [1, 23], automatic destructors [50], `defer` statements [15], and finalizers [7]. We believe that these constructs can be generalized to apply for arbitrary affine effects [40] together with control-flow linearity checking.

Linear type systems [2, 6, 21, 51] can be used to check linear resource usage. As discussed in Section 2.5, such systems provide different guarantees as offered by control-flow linearity. Linear type systems typically focus on how often a function might use its argument, while control-flow linearity centers around how often a function might return. These two concepts are not mutually exclusive.

Previous work used linear types on continuations to capture CPS-translations and control effects more precisely [5, 18]. Filinski [18] introduces the notion of *linear control*, referring to *"the very skeleton around which non-linear features are built"*. This is a tool for language implementors: restricting the expressiveness of the target language, linear types on continuations allow better reasoning about the translation itself and give back some precision that is otherwise lost by

the CPS-translation. In contrast, qualified effect types are designed as a tool for programmers. Similarly, our notion of control-flow linearity is not based on linear types, but on the operational resumption behavior. Reasoning about control-flow linearity helps the programmer to reason about resource safety. Still, we believe that linear types on continuations could also bear fruit in the source language. With linear types we could replace our syntactic check of handler linearity by a linear type check: for a handler that returns results of type τ_r, we could type the operation clause $op_i(x : \alpha_i) : \tau_i$ as $op_i : (\tau_i \to \tau_r) \multimap \alpha_i \to \tau_r$. That is, a linear handler implements its operations as linear in the `resume` continuation and consumes it exactly once. While we believe this approach is well worth exploration, it is unclear how well that would integrate with effect polymorphism and type inference. In contrast, qualified effect types integrate very well with the Hindley-Milner style type and effect system of Koka.

8 Conclusion

We described a new perspective on an old problem using the concept of control-flow linearity. By encapsulating access to external resources in effect handlers we can now locally reason about the safe usage of an external protocol. We introduced qualified effect types to check control-flow linearity. Qualified effect types are a lightweight addition to languages with row-polymorphic effect types and ML style inference, yet offer powerful guarantees. Going forward, we like to investigate further extensions with effects and handlers that are polymorphic in their linearity to enable more usage scenarios.

References

[1] 2006. Standard ECMA-334C# Language Specification 4th edition. (2006). https://ecma-international.org/publications/standards/Ecma-334.htm

[2] P.M. Achten, J.H.G. van Groningen, and M.J. Plasmeijer. 1993. High Level Specification of I/O in Functional Languages. In *Workshop Notes in Computer Science*. Springer-Verlag, 1–17. Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 June 1992.

[3] Kenichi Asai and Yukiyoshi Kameyama. 2007. *Polymorphic Delimited Continuations*. 239–254.

[4] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. DOI: http://dx.doi.org/10.1016/j.jlamp.2014.02.001

[5] Josh Berdine, Peter O'Hearn, Uday Reddy, and Hayo Thielecke. 2002. Linear Continuation-Passing. *Higher-Order and Symbolic Computation* 15, 2 (01 Sep 2002), 181–208.

[6] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. ACM, Article 5, 29 pages.

[7] Hans-J. Boehm. 2003. Destructors, Finalizers, and Synchronization. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'03)*. ACM, 262–272.

[8] Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala. In *Proceedings of the International Symposium on Scala*.

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM.

[10] William D. Clinger. 2003. Implementation of `unwind-protect` in Portable Scheme. (2003). http://www.ccs.neu.edu/home/will/UWESC/uwesc.sch

[11] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. 151–160. DOI: http://dx.doi.org/10.1145/91556.91622

[12] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Proceedings of the Symposium on Trends in Functional Programming*.

[13] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Effectively Tackling the Awkward Squad. In *ML Workshop*.

[14] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective concurrency through algebraic effects. In *OCaml Workshop*.

[15] Alan A. A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language*. Addison-Wesley Publishing Co.

[16] Iulian Dragos, Antonio Cunei, and Jan Vitek. 2007. Continuations in the Java Virtual Machine. *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)* (2007).

[17] Mattias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 180–190.

[18] Andrzej Filinski. 1992. Linear Continuations. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'92)*. ACM, 27–38.

[19] Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'85)*. ACM, 245–254.

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

[21] Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50 (1987), 1–102.

[22] Jean-Yves Girard, Yves LaFont, and Paul Taylor. 1989. *Proofs and Types*. Cambridge University Press.

[23] James Gosling, Bill Joy, and Guy L. Steele. 1996. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[24] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. 1994. Subcontinuations. *Lisp Symb. Comput.* 7, 1 (Jan. 1994), 83–110.

[25] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the Workshop on Type-Driven Development*. 15–27.

[26] J.R. Hindley. 1969. The principal type scheme of an object in combinatory logic. *Trans. of the American Mathematical Society* 146 (Dec. 1969), 29–60.

[27] Mark P. Jones. 1992. A theory of qualified types. In *Proceedings of the European Symposium on Programming (ESOP'92)*, Vol. 582. Springer-Verlag, 287–306. DOI: http://dx.doi.org/10.1007/3-540-55253-7_17

[28] Mark P. Jones. 1994. *Qualified types in Theory and Practice*. Cambridge University Press.

[29] Mark P. Jones. 1995. Simplifying and Improving Qualified Types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM, 160–169.

[30] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming (ICFP'13)*. ACM, 145–158.

[31] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the Haskell Symposium*. 94–105.

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

[33] Peter J. Landin. 1998. A Generalization of Jumps and Labels. *Higher-Order and Symbolic Computation* 11, 2 (1998), 125–143. Reprint from [32].

[34] Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming*. 297–312.

[35] Daan Leijen. 2013. *Koka: Programming with Row-Polymorphic Effect Types*. Technical Report MSR-TR-2013-79. Microsoft Research.

[36] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *MSFP'14, 5th workshop on Mathematically Structured Functional Programming*. DOI: http://dx.doi.org/10.4204/EPTCS.153.8

[37] Daan Leijen. 2017. Implementing Algebraic Effects in C. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, 339–363.

[38] Daan Leijen. 2017. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development (TyDe'17)*. 16–29.

[39] Daan Leijen. 2017. Type Directed Compilation of Row-typed Algebraic Effects. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. 486–499.

[40] Daan Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization*. Technical Report MSR-TR-2018-10. Microsoft Research. 35 pages.

[41] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI'12)*. 91–102. DOI: http://dx.doi.org/10.1145/2103786.2103798

[42] Sam Lindley, Connor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'17)*. 500–514.

[43] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 248–375.

[44] Kent Pitman. 2003. Unwind-Protect versus Continuations. (2003). http://www.nhplace.com/kent/PFAQ/unwind-protect-vs-continuations-original.html

[45] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

[46] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proceedings of the European Symposium on Programming (ESOP'09)*. 80–94.

[47] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

[48] Chung-chieh Shan. 2007. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation* 20, 4 (2007), 371–401.

[49] Dorai Sitaram. 2003. Unwind-protect in portable scheme. In *Proceedings of the Scheme Workshop*.

[50] B. Stroustrup. 1994. *The Design and Evolution of C++*. Addison-Wesley Publishing Co. Chapter 16.1, Exception Handling.

[51] Philip Wadler. 1990. Linear Types Can Change the World!. In *IFIP TC2 Working Conference on Programming Concepts and Methods*. 347–359.

[52] Leo White. 2016. Effect Types for OCaml. (Software – Practice and Experience 2016). https://github.com/lpw25/ocaml-typed-effects

[53] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94. DOI: http://dx.doi.org/10.1006/inco.1994.1093

[54] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the Haskell Symposium*. 1–12.