Soundly Handling Linearity

Soundly Handling Linearity

WENHAO TANG, The University of Edinburgh, United Kingdom
DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland
SAM LINDLEY, The University of Edinburgh, United Kingdom
J. GARRETT MORRIS, University of Iowa, USA

We propose a novel approach to soundly combining linear types with multi-shot effect handlers. Linear type systems statically ensure that resources such as file handles and communication channels are used exactly once. Effect handlers provide a rich modular programming abstraction for implementing features ranging from exceptions to concurrency to backtracking. Whereas conventional linear type systems bake in the assumption that continuations are invoked exactly once, effect handlers allow continuations to be discarded (e.g. for exceptions) or invoked more than once (e.g. for backtracking). This mismatch leads to soundness bugs in existing systems such as the programming language Links, which combines linearity (for session types) with effect handlers. We introduce control-flow linearity as a means to ensure that continuations are used in accordance with the linearity of any resources they capture, ruling out such soundness bugs.

We formalise the notion of control-flow linearity in a System F-style core calculus F°_{eff} equipped with linear types, an effect type system, and effect handlers. We define a linearity-aware semantics in order to formally prove that F°_{eff} preserves the integrity of linear values in the sense that no linear value is discarded or duplicated. In order to show that control-flow linearity can be made practical, we adapt Links based on the design of F°_{eff}, in doing so fixing a long-standing soundness bug.

Finally, to better expose the potential of control-flow linearity, we define an ML-style core calculus Q°_{eff}, based on qualified types, which requires no programmer provided annotations, and instead relies entirely on type inference to infer control-flow linearity. Both linearity and effects are captured by qualified types. Q°_{eff} overcomes a number of practical limitations of F°_{eff}, supporting abstraction over linearity, linearity dependencies between type variables, and a much more fine-grained notion of control-flow linearity.

CCS Concepts: • **Theory of computation** → **Control primitives**; **Type structures**.

Additional Key Words and Phrases: control-flow linearity, multi-shot continuations, linear resources

ACM Reference Format:
Wenhao Tang, Daniel Hillerström, Sam Lindley, and J. Garrett Morris. 2024. Soundly Handling Linearity. *Proc. ACM Program. Lang.* 8, POPL, Article 54 (January 2024), 29 pages. https://doi.org/10.1145/3632896

1 INTRODUCTION

Many programming languages support linear resources such as file handles, communication channels, network connections, and so forth. Special care must be taken to preserve the integrity of linear resources in the presence of first-class continuations that may be invoked multiple times [Friedman and Haynes 1985], as a linear resource may be inadvertently be accessed more than once. Java [Pressler 2018] and OCaml [Sivaramakrishnan et al. 2021] have each recently been retrofitted with facilities for programming with first-class continuations that must be invoked

Authors' addresses: Wenhao Tang, The University of Edinburgh, United Kingdom, wenhao.tang@ed.ac.uk; Daniel Hillerström, Huawei Zurich Research Center, Switzerland, daniel.hillerstrom@ed.ac.uk; Sam Lindley, The University of Edinburgh, United Kingdom, sam.lindley@ed.ac.uk; J. Garrett Morris, University of Iowa, USA, garrett-morris@uiowa.edu.

ACM 2475-1421/2024/1-ART54
https://doi.org/10.1145/3632896

exactly once, partly in order to avoid such pitfalls. Nonetheless, multi-shot continuations are a compelling feature, supporting applications such as backtracking search [Friedman et al. 1984] and probabilistic programming [Kiselyov and Shan 2009]. In this paper we explore how to soundly handle linearity in the presence of multi-shot effect handlers [Plotkin and Pretnar 2013].

We first illustrate the issues with combining linearity with multi-shot effect handlers by exhibiting a soundness bug in the programming language Links [Cooper et al. 2006], which is equipped with linear session-typed channels [Lindley and Morris 2017] and effect handlers with multi-shot continuations [Hillerström et al. 2020a]. We begin by defining a function `outch` that forks a child process and returns an output channel for communicating with it. The idea is that we will use a combination of exceptions and multi-shot continuations to send two integers, rather than an integer followed by a string, along the endpoint (with session type `!Int.!String.End`) returned by the function `outch`.

```
sig outch : () ~> !Int.!String.End
fun outch() {
  fork(fun(ic) {
    var (i, ic) = receive(ic);     # receive the integer
    var (s, ic) = receive(ic);     # receive the string
    println(intToString(i) ^^ s);  # convert, concat, and print
    close(ic)                      # close the input channel
  })
}
```

The primitive `fork` creates a child process and two endpoints of a session-typed channel. One endpoint is passed to the child process and the other endpoint is returned to the caller. Here the function returns an output endpoint of type `!Int.!String.End` and the child process is supplied with an input endpoint of type `?Int.?String.End`. The child receives an integer and a string on the input endpoint, then prints them out before closing the endpoint.

Now we invoke `outch` in a context in which we exploit the power of multi-shot continuations to return twice and the power of exceptions to abort the current computation.

```
handle({
  var oc = outch();
  var msg = if (do Choose) 42 else 84; # choose an integer message to send
  var oc = send(msg, oc);
  do Fail;                             # this is our exception
  var oc = send("well-typed", oc);
  close(oc)
}) {
  case <Fail> -> ()
  case <Choose => resume> -> resume(true); resume(false)
}
```

We handle a computation that performs two operations: 1) `Choose : () => Bool`; and 2) `Fail : forall a. () => a`. The handled computation invokes `outch`, forking a child process and binding the output endpoint of the resulting channel to `oc`. Next, it invokes the operation `Choose` to select between two possible integer messages, which is sent on the channel. Then, it performs the `Fail` operation, before sending a string along the channel and closing it. This is all very well and satisfies the type-checker; however, the described control flow is not actually what happens, because in fact the continuation of `Choose` is invoked twice and the continuation of `Fail` is never invoked. The

behaviours of `Fail` and `Choose` are defined by the corresponding operation clauses of the handler. For `Fail` the captured continuation is discarded (it must be: it is never bound); for `Choose` the continuation is bound to `resume` and invoked twice: first with `true` and then with `false`.

Running the program causes a segmentation fault when printing the received values, as it erroneously attempts to concatenate a string with an integer. To see why, follow the control flow of the parent process. It performs `Choose`, which initially selects `42` and sends it over the channel. The child process receives this integer and subsequently expects to receive a string. Back on the parent process execution is aborted via `Fail`, which causes the initial invocation of `resume` to return, leading to the second invocation of `resume`, which restores the aborted context at the point of selecting an integer. Now `Choose` selects `84` and sends it over the channel. The child process receives this second integer, mistakenly treating it as a string.

In this paper we rule out such soundness bugs by tracking *control-flow linearity*: a means to statically assure how often a continuation may be invoked, mediating between linear resources and effectful operations to ensure that effect handlers cannot violate linearity constraints on resources.

The main contributions of this paper are:

- We give high-level overview of the main ideas of the paper through a series of worked examples that illustrate the difficulties of combining effect handlers with linearity, how they can be resolved by tracking control-flow linearity, and how the approach can be refined using qualified types [Jones 1994] (Section 2).
- We introduce F°_{eff} (pronounced "F-eff-pop"), a System F-style core calculus equipped with linear types, an effect type system, and effect handlers (Section 3). We prove syntactic type soundness and a semantic linear safety property.
- Inspired by F°_{eff} we implement control-flow linearity in Links, fixing a long-standing type-soundness bug (Section 4).
- Motivated by expressiveness limitations of F°_{eff} we introduce Q°_{eff} (pronounced "Q-eff-pop"), an ML-style core calculus inspired by Quill [Morris 2016] and Rose [Morris and McKinna 2019], based on qualified types (Section 5). We prove soundness and completeness of type inference for Q°_{eff}. Along the way, we identify a semantic soundness bug in Quill and conjecture a fix.

Section 6 outlines how control-flow linearity applies to shallow handlers [Hillerström and Lindley 2018]. Section 7 discusses related work and Section 8 conclude and discusses future work.

2 OVERVIEW

In this section, we give a high-level overview of the main ideas of the paper by way of a series of examples. We first compare standard value linearity with non-standard control-flow linearity, illustrating how the latter may be tracked in an explicit calculus F°_{eff} (Section 3). For readability we omit uninteresting syntactic artifacts from our examples. We show how control-flow linearity allows linear resources and multi-shot continuations to coexist peacefully. We then highlight two limitations of F°_{eff}: linear types require syntactic overhead which harms modularity, and row-polymorphism based effect types lead to coarse tracking of control-flow linearity. We exploit qualified types to relax both limitations in an ML-style calculus Q°_{eff} (Section 5).

2.1 Value Linearity

Value linearity classifies the *use* of values: linear values must be used exactly once whereas unlimited values can be used zero, one, or multiple times (linear types differ from uniqueness types, which instead track the number of references to a value). Equivalently, value linearity characterises whether values contain linear resources: linear values can contain linear resources whereas unlimited values cannot. Conventional linear type systems track value linearity. F°_{eff} adapts the subkinding-based

linear type system of F° [Mazurak et al. 2010]. The linearity Y of a value type is part of its kind Type^Y and can be either linear $\circ$ or unlimited $\bullet$. For example, file handles are linear resources ($\mathsf{File} : \mathsf{Type}^\circ$) and integers are unlimited resources ($\mathsf{Int} : \mathsf{Type}^\bullet$).

A linearity annotation on a λ-abstraction defines the linearity of the function itself. Consider the following function faithfulWrite which takes a file handle f and returns another function that takes a string s, faithfully writes s to f, and then closes the file handle.

$$\begin{aligned}&\mathsf{faithfulWrite} \;:\; \mathsf{File} \to^\bullet (\mathsf{String} \to^\circ ())\\ &\mathsf{faithfulWrite} = \lambda^\bullet f.(\lambda^\circ s.\mathbf{let}\;\; f' \leftarrow \mathsf{write}\,(s, f)\;\mathbf{in}\;\mathsf{close}\,f')\end{aligned}$$

The outer unlimited function ($\to^\bullet$) yields a linear function ($\to^\circ$) expecting a string. The linear type system dictates that the inner function is linear as it captures the linear file handle f.

One important property of value linearity is that unlimited value types can be treated as linear value types, as it is always safe to use unlimited values (which contain no linear resources) just once. This property is embodied by the subkinding relation $\vdash \mathsf{Type}^\bullet \leq \mathsf{Type}^\circ$ in $\mathrm{F}^\circ_{\mathrm{eff}}$. For instance, consider the polymorphic identity function.

$$\begin{aligned}&\mathsf{id} \;:\; \forall \mu^{\mathsf{Row}}\, \alpha^{\mathsf{Type}^\circ}.\, \alpha \to^\bullet \alpha\,!\,\{\mu\}\\ &\mathsf{id} = \Lambda \mu^{\mathsf{Row}}\, \alpha^{\mathsf{Type}^\circ}.\, \lambda^\bullet x.\, x\end{aligned}$$

The return type of the function is a computation type $\alpha\,!\,\{\mu\}$ where α is the linear type of values returned (x is used exactly once) and μ is the row of effects performed by the function. (We chose to omit the corresponding effect annotations in the signature of faithfulWrite because they are empty, but henceforth we will write them explicitly.) Subkinding allows the identity function to be applied to both linear and unlimited values. It is always sound to use an unlimited value exactly once. Thus, we have both $\vdash \mathsf{Int} : \mathsf{Type}^\circ$ and $\vdash \mathsf{File} : \mathsf{Type}^\circ$, and if R is an effect row type:

$$\begin{aligned}&\mathsf{id}\; R\; \mathsf{File} : \mathsf{File} \to^\bullet \mathsf{File}\,!\,\{R\}\\ &\mathsf{id}\; R\; \mathsf{Int}\;\; : \mathsf{Int} \to^\bullet \mathsf{Int}\,!\,\{R\}\end{aligned}$$

2.2 Control-Flow Linearity

Control-flow linearity tracks how many times control may enter a local context: a control-flow-linear context must be entered exactly once; a control-flow-unlimited context may be entered zero, one, or multiple times. Equivalently, control-flow linearity characterises whether a local context captures linear resources: a control-flow-linear context can capture linear resources; a control-flow-unlimited context cannot.

To better explain control-flow linearity, we first reprise the soundness problem due to the interaction of linear resources and multi-shot continuations of Section 1 via a simpler example in $\mathrm{F}^\circ_{\mathrm{eff}}$. Consider the following function $\mathsf{dubiousWrite}_{\not\mathsf{x}}$, which takes a file handle and non-deterministically writes "A" or "B" to it depending on the result of Choose. We ignore control-flow linearity for now.

$$\begin{aligned}&\mathsf{dubiousWrite}_{\not\mathsf{x}} \;:\; \mathsf{File} \to^\bullet ()\,!\,\{\mathsf{Choose} : () \twoheadrightarrow \mathsf{Bool}\}\\ &\mathsf{dubiousWrite}_{\not\mathsf{x}} = \lambda^\bullet f.\\ &\quad \mathbf{let}\; b \leftarrow (\mathbf{do}\;\mathsf{Choose}\,())^{\{\mathsf{Choose}:()\twoheadrightarrow\mathsf{Bool}\}}\;\mathbf{in}\\ &\quad \left.\begin{array}{l}\mathbf{let}\; s \leftarrow \mathbf{if}\; b\;\mathbf{then}\;\text{"A"}\;\mathbf{else}\;\text{"B"}\;\mathbf{in}\\ \mathbf{let}\; f' \leftarrow \mathsf{write}\,(s, f)\;\mathbf{in}\;\mathsf{close}\,f'\end{array}\right\}\text{continuation of Choose}\end{aligned}$$

The **do** Choose () expression invokes operation Choose with a unit argument. $\mathrm{F}^\circ_{\mathrm{eff}}$ adapts an effect system based on Rémy-style row polymorphism [Hillerström and Lindley 2016; Lindley and Cheney 2012]. Effect types in $\mathrm{F}^\circ_{\mathrm{eff}}$ are rows containing operation labels with their signatures and ended with potential row variables. The effect type $\{\mathsf{Choose} : () \twoheadrightarrow \mathsf{Bool}\}$ denotes that $\mathsf{dubiousWrite}_{\not\mathsf{x}}$ may

invoke the operation Choose, which takes a unit and returns a boolean value as indicated by its signature $() \twoheadrightarrow \mathsf{Bool}$. The problem arises when we handle Choose using multi-shot continuations.

$$\textbf{let}\ f \leftarrow \mathsf{open}\,\text{"C.txt"}\ \textbf{in handle}\ (\mathsf{dubiousWrite}_{\text{✗}}\ f)\ \textbf{with}\ \{\mathsf{Choose}\ \_\ r \mapsto r\ \mathsf{true}\,;r\ \mathsf{false}\}$$

The file "C.txt" is opened and the file handle is bound to f before $\mathsf{dubiousWrite}_{\text{✗}}\ f$ is handled by an effect handler that handles the Choose operation. In the handler clause, r binds the continuation of Choose, which expects a parameter of type Bool. As r is invoked twice (first with true and then with false), the file handle f is written and closed twice, which leads to a runtime error because it is closed before the second write. The essential problem is that the continuation of Choose should be used linearly as it captures the linear file handle f, but it is invoked twice by the effect handler. Conventional linear type systems cannot detect this kind of error as they only track value linearity.

Motivated by the observation that only a local context, reified as the continuation of an operation, may be captured by a multi-shot handler, we track control-flow linearity at the granularity of operations. We use the control-flow linearity of an operation to represent the control-flow linearity of the continuation of the operation. Control-flow-linear operations can be used in contexts which may contain linear resources, whereas control-flow-unlimited operations cannot. An operation signature $A \twoheadrightarrow^{Y} B$ is annotated with a linearity Y to denote its control-flow linearity. The $\mathsf{dubiousWrite}_{\text{✗}}$ function can now be rewritten to correctly track control-flow linearity as follows.

$$\begin{array}{l}
\mathsf{dubiousWrite}_{\checkmark} : \mathsf{File} \to^{\bullet} () \,!\, \{\mathsf{Choose} : () \twoheadrightarrow^{\circ} \mathsf{Bool}\} \\
\mathsf{dubiousWrite}_{\checkmark} = \lambda^{\bullet} f. \\
\quad \left.\begin{array}{l}
\textbf{let}^{\circ} b \leftarrow (\textbf{do}\ \mathsf{Choose}\ ())^{\{\mathsf{Choose}:() \twoheadrightarrow^{\circ} \mathsf{Bool}\}}\ \textbf{in} \\
\textbf{let}^{\circ} s \leftarrow \textbf{if}\ b\ \textbf{then}\ \text{"A"}\ \textbf{else}\ \text{"B"}\ \textbf{in} \\
\textbf{let}^{\circ} f' \leftarrow \mathsf{write}\,(s, f)\ \textbf{in}\ \mathsf{close}\ f'
\end{array}\right\} \text{continuation of Choose}
\end{array}$$

Now, the type of $\mathsf{dubiousWrite}_{\checkmark}$ specifies that the operation $\mathsf{Choose} : () \twoheadrightarrow^{\circ} \mathsf{Bool}$ is control-flow linear (i.e. the continuation of Choose is linear). We also annotate let-bindings with linearity information. In $\textbf{let}^{Y} x \leftarrow M\ \textbf{in}\ N$, the term N has control-flow linearity Y, and in particular the $\circ$ annotations on the let-bindings in $\mathsf{dubiousWrite}_{\checkmark}$ permit the use of the linear file handle throughout.

The linear type system of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ uses the control-flow linearity of operations to restrict the use of continuations in handlers, which ensures that control-flow-linear contexts are entered only once. For instance, consider the handling of $\mathsf{dubiousWrite}_{\checkmark}$ with the same multi-shot handler.

$$\textbf{let}\ f \leftarrow \mathsf{open}\,\text{"C.txt"}\ \textbf{in handle}\ (\mathsf{dubiousWrite}_{\checkmark}\ f)\ \textbf{with}\ \{\mathsf{Choose}\ \_\ r \mapsto r\ \mathsf{true}\,;r\ \mathsf{false}\}$$

This is ill-typed due to the fact that Choose is control-flow linear, which means the resumption r has a linear function type, meaning it must be applied exactly once.

We lift the control-flow linearity of operations to effect row types and reflect it in their kinds Row^{Y}. Similar to value linearity, we also have a subkinding relation for control-flow linearity. Recall that the control-flow linearity of (the operations in) effect row types is actually the control-flow linearity of their contexts, not themselves. This induces a duality between value linearity and control-flow linearity paralleling the duality between positive values and negative continuations. As a consequence, the subkinding relation for control-flow linearity is $\vdash \mathsf{Row}^{\circ} \leq \mathsf{Row}^{\bullet}$, the reverse of that for value linearity. Intuitively, this says that control-flow-linear operations can be treated as control-flow-unlimited operations, because it is safe to use control-flow-linear operations in unlimited contexts. For example, consider the following function tossCoin which takes a function that returns a boolean and tosses a coin using this function.

$$\begin{array}{l}
\mathsf{tossCoin} : \forall \mu^{\mathsf{Row}^{\bullet}}.((() \to^{\bullet} \mathsf{Bool} \,!\, \{\mu\}) \to^{\bullet} \mathsf{String} \,!\, \{\mu\} \\
\mathsf{tossCoin} = \Lambda \mu^{\mathsf{Row}^{\bullet}}.\lambda^{\bullet} g.\ \textbf{let}^{\bullet}\ b \leftarrow g\,()\ \textbf{in if}\ b\ \textbf{then}\ \text{"heads"}\ \textbf{else}\ \text{"tails"}
\end{array}$$

As no linear resource is used, the effect type of tossCoin and its parameter is given by a control-flow-unlimited row variable $\mu : \mathsf{Row}^\bullet$. Via subkinding, we can instantiate μ with operations with either control-flow linearity. For instance, suppose we have $\vdash R_1 : \mathsf{Row}^\bullet$ and $\vdash R_2 : \mathsf{Row}^\circ$ for $R_1 = \mathsf{Choose} : () \twoheadrightarrow^\bullet \mathsf{Bool}$ and $R_2 = \mathsf{Choose} : () \twoheadrightarrow^\circ \mathsf{Bool}$, then:

$$\begin{aligned}&\mathsf{tossCoin}\ R_1\ (\lambda^\bullet().(\mathbf{do}\ \mathsf{Choose}\ ())^{\{R_1\}}) : \mathsf{String}\,!\,\{R_1\}\\ &\mathsf{tossCoin}\ R_2\ (\lambda^\bullet().(\mathbf{do}\ \mathsf{Choose}\ ())^{\{R_2\}}) : \mathsf{String}\,!\,\{R_2\}\end{aligned}$$

The subkinding relation of control-flow linearity only influences how operations are used, not how they are handled. We can *use* control-flow-linear operations as control-flow-unlimited operations (i.e., use them in unlimited contexts), but this does not imply that we can *handle* control-flow-linear operations as control-flow-unlimited operations (i.e., handle them by resuming any number of times). Our linear type system does not allow control-flow-linear operations to be handled by multi-shot handlers despite the subkinding relation $\mathsf{Row}^\circ \le \mathsf{Row}^\bullet$. This is because when handling, we directly look at the control-flow linearity on operation signatures instead of their kinds, where no $\twoheadrightarrow^\circ$ can be upcast to $\twoheadrightarrow^\bullet$. This can be seen more clearly from the typing rules in Section 3.2. We formally state the soundness of $\mathrm{F}^\circ_{\mathrm{eff}}$ in Sections 3.4 and 3.5.

2.3 Qualified Linear Types

As we have seen from the examples so far, $\mathrm{F}^\circ_{\mathrm{eff}}$ requires linearity annotations on λ-abstractions and let-bindings. Though this can suffice for an explicit calculus, it can prove cumbersome for practical programming languages and curtail the modularity of programs. Unfortunately, we cannot entirely overcome these limitations by introducing subsumption relations between types, or using Hindley-Milner type inference to infer them. The reason is that there are inner dependencies on the linearity. For instance, consider the following function verboseId which is almost the same as the function id in Section 2.1 but outputs the log message "id is called" using the operation $\mathsf{Print} : \mathsf{String} \twoheadrightarrow ()$ before returning.

$$\begin{aligned}&\mathsf{verboseId}\ :\ \forall \mu^{\mathsf{Row}^{Y_1}}\ \alpha^{\mathsf{Type}^{Y_2}}.\, \alpha \to^{Y_0} \alpha\,!\,\{\mathsf{Print} : \mathsf{String} \twoheadrightarrow^{Y_3} ()\,;\mu\}\\ &\mathsf{verboseId} = \Lambda \mu^{\mathsf{Row}^{Y_1}}\ \alpha^{\mathsf{Type}^{Y_2}}.\, \lambda^{Y_0} x.\, \mathbf{let}^{Y_4}\ () \leftarrow \mathbf{do}\ \mathsf{Print}\ \text{"id is called"}\ \mathbf{in}\ x\end{aligned}$$

Depending on different choices of Y_0, Y_1, Y_2, Y_3, and Y_4, we can give ten well typed variations of verboseId. Their types are shown as follows, omitting primary kinds and signatures for readability.

$$\begin{array}{ll}\forall \mu^\bullet\ \alpha^\bullet.\alpha \to^\bullet \alpha\,!\,\{\mathsf{Print} : \bullet\,;\mu\} & \forall \mu^\bullet\ \alpha^\bullet.\alpha \to^\circ \alpha\,!\,\{\mathsf{Print} : \bullet\,;\mu\}\\ \forall \mu^\bullet\ \alpha^\bullet.\alpha \to^\bullet \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\} & \forall \mu^\bullet\ \alpha^\bullet.\alpha \to^\circ \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\}\\ \forall \mu^\circ\ \alpha^\bullet.\alpha \to^\bullet \alpha\,!\,\{\mathsf{Print} : \bullet\,;\mu\} & \forall \mu^\circ\ \alpha^\bullet.\alpha \to^\circ \alpha\,!\,\{\mathsf{Print} : \bullet\,;\mu\}\\ \forall \mu^\circ\ \alpha^\bullet.\alpha \to^\bullet \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\} & \forall \mu^\circ\ \alpha^\bullet.\alpha \to^\circ \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\}\\ \forall \mu^\circ\ \alpha^\circ.\alpha \to^\bullet \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\} & \forall \mu^\circ\ \alpha^\circ.\alpha \to^\circ \alpha\,!\,\{\mathsf{Print} : \circ\,;\mu\}\end{array}$$

The key observation is that the control-flow linearity of the operation Print (as well as the row variable μ) depends on the value linearity of the parameter type α, because the parameter x is used in the continuation of Print. To express this kind of dependency, we use a linear type system based on qualified types inspired by Quill [Morris 2016]. In the ML-style calculus $\mathrm{Q}^\circ_{\mathrm{eff}}$ with qualified linear types, verboseId can be written and ascribed a principal type as follows.

$$\begin{aligned}&\mathsf{verboseId}\ :\ \forall \alpha\, \mu\, \phi\, \phi'.\, (\alpha \preceq \phi) \Rightarrow \alpha \to^{\phi'} \alpha\,!\,\{\mathsf{Print} : \phi; \mu\}\\ &\mathsf{verboseId} = \lambda x.\, \mathbf{do}\ \mathsf{Print}\ \text{"42"}\,;\, x\end{aligned}$$

The linearity variables ϕ and ϕ' quantify over $\circ$ and $\bullet$. We do not use kinds to represent linearity of type variables; instead, all linearity information is represented using predicates of the form $\tau \preceq \tau'$, where τ is a value type, row type or linearity type ($\circ$, $\bullet$ or a linearity variable). The type scheme of

verboseId is extended with the predicate $\alpha \preceq \phi$, meaning that the value linearity of α is less than that of ϕ, which is the control-flow linearity of Print. This type scheme succinctly expresses all ten possibilities listed above. The type inference algorithm of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ (Section 5.4) infers all such linearity dependency constraints without the need for any type, effect, or linearity annotations.

2.4 Qualified Effect Types

In addition to the syntactic overhead of linear types, the row-based effect system of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ is also not entirely satisfying when tracking control-flow linearity. Row-based effect systems have demonstrated their practicality in research languages such as Links [Hillerström and Lindley 2016], Koka [Leijen 2017], and Frank [Lindley et al. 2017]. In such effect systems, sequenced computations must have the same effect type, which can be smoothly realised by unification in systems based on Hindley-Milner type inference. However, though fixing effect types between sequenced computations is often acceptable, it does introduce some imprecision, and this can become more pronounced when control-flow linearity is brought into the mix.

To see the problem concretely in $\mathrm{F}^{\circ}_{\mathrm{eff}}$, consider the following function verboseClose which takes a file handle, reads a string using the operation Get : () ↠ String, closes the file handle, and outputs the string using the operation Print : String ↠ ().

$$\begin{aligned}&\mathsf{verboseClose} \;:\; \mathsf{File} \to^{\bullet} () \,!\, \{R\}\\ &\mathsf{verboseClose} = \lambda^{\bullet} f.\; \mathbf{let}^{\circ} s \leftarrow (\mathbf{do}\ \mathsf{Get}\ ())^{\{R_1\}}\ \mathbf{in}\ \mathbf{let}^{\bullet} () \leftarrow \mathsf{close}\, f\ \mathbf{in}\ (\mathbf{do}\ \mathsf{Print}\, s)^{\{R_2\}}\end{aligned}$$

Note that the second **let**-binding does not need to be annotated as linear, because the linear resource f does not appear after it. The linear resource f also does not appear in the continuation of Print. Since R_1, R_2, and R should be equal in the row-based effect system of $\mathrm{F}^{\circ}_{\mathrm{eff}}$, omitting the full operation signatures for simplicity, we could write $R = R_1 = R_2 = \{\mathsf{Get} : \circ, \mathsf{Print} : \bullet\}$ in the ideal case. However, this is actually ill-typed because all operations in R_1 should be control-flow linear, as the linear resource f is used in their continuations.

An intuitive way to relax this limitation of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ is to introduce a trivial subtyping relation on concrete effect row types. We say R_1 is a subtype of R_2, if all operation labels in R_1 are also in R_2 with the same signatures, and when R_1 ends with a row variable, R_2 must end with the same row variable. Then, in the verboseClose example, we can write $R_1 = \{\mathsf{Get} : \circ\}$, $R_2 = \{\mathsf{Print} : \bullet\}$, and $R = \{\mathsf{Get} : \circ, \mathsf{Print} : \bullet\}$, which are safe given that R_1 and R_2 are both subtypes of R.

We call the subtyping relation trivial because it does not allow subtyping between row variables; an open row R_1 is a subtype of R_2 only if R_2 contains the same row variable as R_1. For the above verboseClose example this works, but for other functions which make greater use of polymorphism, it can still seem overly-restrictive. For instance, consider the following function sandwichClose which takes two functions and a file handle, and makes a sandwich using them.

$$\begin{aligned}&\mathsf{sandwichClose} \;:\; (() \to^{\bullet} () \,!\, \{R_1\}, \mathsf{File}, () \to^{\bullet} () \,!\, \{R_2\}) \to^{\bullet} () \,!\, \{R\}\\ &\mathsf{sandwichClose} = \lambda^{\bullet}(g, f, h).\; \mathbf{let}^{\circ} () \leftarrow g\,()\ \mathbf{in}\ \mathbf{let}^{\bullet} () \leftarrow \mathsf{close}\, f\ \mathbf{in}\ h\,()\end{aligned}$$

Using our trivial-subtyping workaround, we require both R_1 and R_2 to be subtypes of R. The problem appears when we try to be polymorphic over R_1 and R_2. Because they are subtypes of the same row type R, their row variables must be the same, i.e., we can only write $R_1 = R_2 = \mu$ in $\mathrm{F}^{\circ}_{\mathrm{eff}}$.

To support non-trivial subtyping relations between row variables, we may again use qualified types, this time to express row subtyping constraints. In addition to qualified linear types, $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ also supports qualified effect types inspired by Rose [Morris and McKinna 2019]. In $\mathrm{Q}^{\circ}_{\mathrm{eff}}$, the function sandwichClose can be given the following type. Note that here we still choose to fix functions to be unlimited for readability.

$$\begin{aligned}\mathsf{sandwichClose} \;&:\; \forall \mu_1\, \mu_2\, \mu. (\mu_1 \leqslant \mu, \mu_2 \leqslant \mu, \mathsf{File} \preceq \mu_1) \\ &\Rightarrow (() \to^{\bullet} () \,!\, \{\mu_1\}, \mathsf{File}, () \to^{\bullet} () \,!\, \{\mu_2\}) \to^{\bullet} () \,!\, \{\mu\} \\ \mathsf{sandwichClose} \;&=\; \lambda^{\bullet}(g, f, h).\ \mathbf{let}\ () \leftarrow g\,()\ \mathbf{in}\ \mathbf{let}\ () \leftarrow \mathsf{close}\, f\ \mathbf{in}\ h\,()\end{aligned}$$

The constraints $\mu_1 \leqslant \mu$ and $\mu_2 \leqslant \mu$ express that rows μ_1 and μ_2 are contained in μ, and the constraint $\mathsf{File} \preceq \mu_1$ expresses that the value linearity of File is less than the control-flow linearity of μ_1, which essentially means that μ_1 is control-flow linear. As in Section 2.3, the type inference algorithm of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ infers these row subtyping constraints without the need for any annotation. The qualified linear types and qualified effect types of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ are decidable. We give a constraint solving algorithm which checks the satisfiability of both linearity constraints and row constraints in Section 5.6.

3 AN EXPLICIT HANDLER CALCULUS WITH LINEAR TYPES

In this section, we present the syntax, type-and-effect system, operational semantics and metatheory of $\mathrm{F}^{\circ}_{\mathrm{eff}}$, a System F-style fine-grain call-by-value calculus with linear types and effect handlers. $\mathrm{F}^{\circ}_{\mathrm{eff}}$ is based on the core language of Links which adapts the subkinding-based linear type system of F° [Mazurak et al. 2010] and a row-based effect system [Hillerström and Lindley 2016; Lindley and Cheney 2012]. The linear type system and effect system of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ are extended to track control-flow linearity, which addresses the soundness problem arising from the interference of linear resources and multi-shot continuations. We show that $\mathrm{F}^{\circ}_{\mathrm{eff}}$ is truly linearity safe by defining a linearity-aware semantics and proving that no linear resource is discarded or duplicated during evaluation in the presence of multi-shot effect handlers.

3.1 Syntax and Kinding Rules

Figure 1 shows the syntax of types, kinds, contexts, values, and computations of $\mathrm{F}^{\circ}_{\mathrm{eff}}$. We introduce a syntactic category Y for linearity consisting of $\bullet$ and $\circ$, which intuitively means unlimited and linear, respectively. The meaning of linearity varies for values and effects; value types track value linearity, and effect types track control-flow linearity. Everything relevant to linearity is highlighted in the figure. The remaining part is a relatively standard fine-grain call-by-value calculus with effect handlers and row-based effect system [Hillerström et al. 2020a].

$\mathrm{F}^{\circ}_{\mathrm{eff}}$ explicitly distinguishes between value types and computation types as well as their terms. Value types include type variables α, function types $A \to^{Y} C$, and polymorphic types $\forall^{Y}\alpha^{K}.C$. Value terms include value variables x, λ-abstractions $\lambda^{Y}x^{A}.M$, and type abstractions $\Lambda^{Y}\alpha^{K}.M$. Function types, polymorphic types, and abstractions are annotated with their value linearity Y. In examples we will freely make use of base types and algebraic data types whose treatment is quite standard. We elect to allow polymorphic computation types rather than applying the value restriction.

A computation type $A \,!\, E$ comprises a result value type A and an effect type E specifying the operations that the computation might perform. Effect types $\{R\}$ are represented by row types R. Each operation label in rows is annotated with a presence type P, which indicates that the label is either absent Abs, present with signature $A \twoheadrightarrow^{Y} B$, or polymorphic θ in its presence. An operation signature $A \twoheadrightarrow^{Y} B$ describes an operation with parameter of type A that returns a result of type B and whose control-flow linearity is Y. Row types are either open (ending with a row variable μ) or closed (ending with $\cdot$, which we often omit). We identify rows up to reordering of labels and ignore absent labels in closed row types [Rémy 1994]. Handler types $C \rightrightarrows D$ represent handlers transforming computations of type C to computations of type D. By convention, we let α range over value type variables, μ over row type variables, and θ over presence type variables, but we also let α range over all over them (e.g. when binding quantifiers of unspecified kind).

Function application $V\,W$ and type application $V\,T$ are standard. A computation $(\mathbf{return}\ V)^{E}$ returns the value V. An operation invocation $(\mathbf{do}\ \ell\ V)^{E}$ invokes the operation ℓ with parameter

$$
\begin{array}{lll}
\text{Value types} & A, B & ::= \alpha \mid A \to^{Y} C \mid \forall^{Y} \alpha^{K}.C \\
\text{Computation types} & C, D & ::= A \,!\, E \\
\text{Effect types} & E & ::= \{R\} \\
\text{Row types} & R & ::= \ell : P; R \mid \mu \mid \cdot \\
\text{Presence types} & P & ::= \mathsf{Abs} \mid A \twoheadrightarrow^{Y} B \mid \theta \\
\text{Handler types} & F & ::= C \rightrightarrows D \\
\text{Types} & T & ::= A \mid R \mid P \mid C \mid E \mid F \\
\text{Kinds} & K & ::= \mathsf{Type}^{Y} \mid \mathsf{Row}_{\mathcal{L}}{}^{Y} \mid \mathsf{Presence}^{Y} \mid \mathsf{Effect} \mid \mathsf{Comp} \mid \mathsf{Handler} \\
\text{Linearity} & Y & ::= \bullet \mid \circ \\
\text{Label sets} & \mathcal{L} & ::= \emptyset \mid \{\ell\} \uplus \mathcal{L} \\
\text{Type contexts} & \Gamma & ::= \cdot \mid \Gamma, x : A \\
\text{Kind contexts} & \Delta & ::= \cdot \mid \Delta, \alpha : K \\
\text{Values} & V, W & ::= x \mid \lambda^{Y} x^{A}.M \mid \Lambda^{Y} \alpha^{K}.M \\
\text{Computations} & M, N & ::= V\,W \mid V\,T \mid (\mathbf{return}\ V)^{E} \mid (\mathbf{do}\ \ell\ V)^{E} \\
 & & \quad \mid \mathbf{let}^{Y} x \leftarrow M\ \mathbf{in}\ N \mid \mathbf{handle}\ M\ \mathbf{with}\ H \\
\text{Handlers} & H & ::= \{\mathbf{return}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto M\} \uplus H
\end{array}
$$

Fig. 1. Syntax of Types, Kinds, Contexts, Values and Computations of $\mathrm{F}^{\circ}_{\mathrm{eff}}$

V. They are both annotated with their effect types for deterministic typing. Sequencing $\mathbf{let}^{Y}\, x \leftarrow M\ \mathbf{in}\ N$ evaluates M and binds its result to x in N. The linearity Y basically indicates the control-flow linearity of N. Handling $\mathbf{handle}\ M\ \mathbf{with}\ H$ handles computation M with handler H. Handlers are given by a return clause $\mathbf{return}\ x \mapsto M$, which binds the returned value as x in M, and a list of operation clauses $\ell\ p\ r \mapsto M$, which bind the operation parameter to p and continuation to r in M.

We have six kinds K, one for each syntactic category of types. Kinds are parameterised by linearity Y. The kinds of value types Type^{Y} denote value linearity, and the kinds of presence types $\mathsf{Presence}^{Y}$ and row types $\mathsf{Row}_{\mathcal{L}}{}^{Y}$ denote control-flow linearity. The label set $\mathcal{L}$ tracks the labels that should not appear in a row, which is used to avoid duplicated labels in rows. The kinds of effect, computation, and handler types are not annotated with any linearity information. Type contexts Γ associate value variables with types, and kind contexts Δ associate type variables with kinds.

Figure 2 gives the kinding rules. Linearity-relevant parts are highlighted. The kinding relation $\Delta \vdash T : K$ states that type T has kind K in context Δ. The subkinding relation $\vdash K \leq K'$ states that K is a subkind of K'. We sometimes write simply $\Delta \vdash T : Y$ for value, row and presence types when the underlying kind is clear. The kinding rules for effect, computation, and handler types are standard [Hillerström et al. 2020a] and irrelevant to linearity (K-Effect, K-Comp, and K-Handler).

The kind context maintains kinds for variables (K-TyVar). The value linearity of function and polymorphic types comes from their annotations (K-Forall and K-Fun). Base types have their own value linearity, e.g., $\vdash \mathsf{File} : \circ$ and $\vdash \mathsf{Int} : \bullet$. The value linearity of (omitted) algebraic datatypes like pair types (A, B) is lifted from their components; $\vdash (A, B) : \circ$ if either $\vdash A : \circ$ or $\vdash B : \circ$.

As shown in Section 2.1, for value linearity, we have a subkinding relation $\vdash \mathsf{Type}^{\bullet} \leq \mathsf{Type}^{\circ}$ given by subkinding rules S-Lin and S-Type. This allows us to use unlimited value types as linear value types since it is always safe to use unlimited values linearly (e.g., the function id in Section 2.1).

We track control-flow linearity at the granularity of operations, and lift it to the kinds of presence types and row types. Absent labels and empty rows can be given any control-flow linearity (K-Absent and K-EmptyRow). The control-flow linearity of present labels comes directly

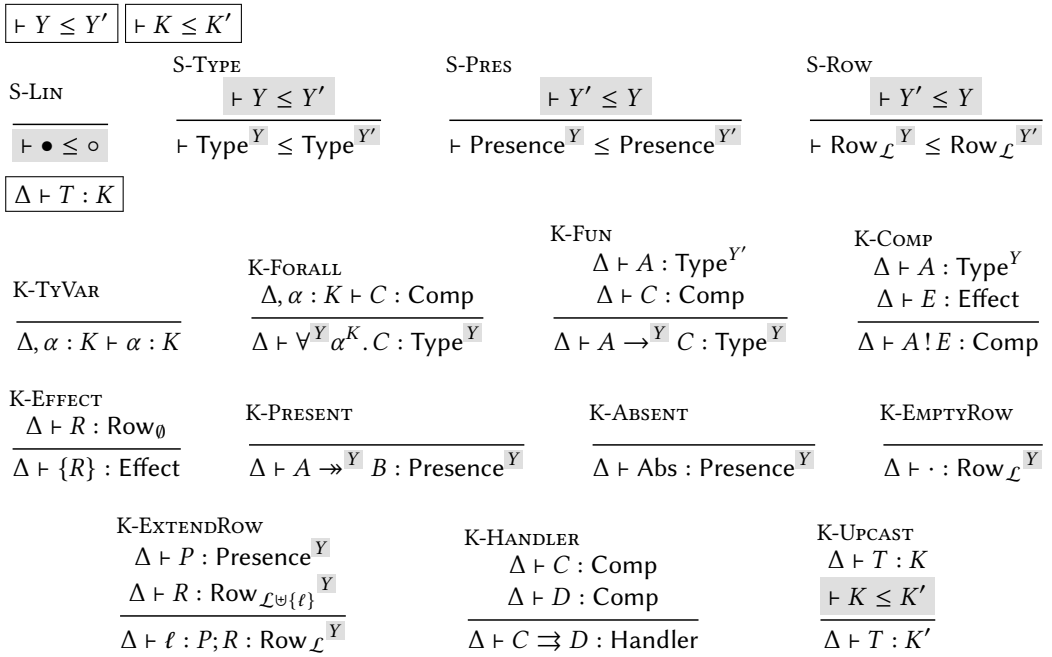

Fig. 2. Kinding and Subkinding Rules for $\mathrm{F}^{\circ}_{\mathrm{eff}}$

from operation signatures (K-Present). The control-flow linearity of row extensions are given by the labels and remaining rows (K-ExtendRow).

As shown in Section 2.2, control-flow linearity is dual to value linearity in some sense: we have $\vdash \mathsf{Row}_{\mathcal{L}}{}^{\circ} \leq \mathsf{Row}_{\mathcal{L}}{}^{\bullet}$ and $\vdash \mathsf{Presence}^{\circ} \leq \mathsf{Presence}^{\bullet}$ given by subkinding rules S-Lin, S-Pres, and S-Row. This allows linear effect rows to be used as unlimited effect rows as it is always safe to use control-flow-linear operations in unlimited contexts (e.g., the function tossCoin in Section 2.2).

3.2 Typing Rules

We define two auxiliary relations in Figure 3 for typing rules. The judgement $\Delta \vdash \Gamma : Y$ states that under kind context Δ all types in Γ have linearity Y. As the subkinding relation for value linearity holds that $\mathsf{Type}^{\bullet} \leq \mathsf{Type}^{\circ}$, the relation $\Delta \vdash \Gamma : \bullet$ guarantees that all variables in Γ are unlimited and the relation $\Delta \vdash \Gamma : \circ$ is a tautology. Dually, as the subkinding relation for control-flow linearity holds that $\mathsf{Row}^{\circ} \leq \mathsf{Row}^{\bullet}$, the relation $\Delta \vdash R : \circ$ guarantees that all operations in R are control-flow linear and the relation $\Delta \vdash R : \bullet$ is a tautology. The context splitting judgement $\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2$ states that under kind context Δ the type context Γ is well formed and can be split into two contexts Γ_1 and Γ_2 such that each linear variable only appears in one of them. We write $\Delta \vdash \Gamma_1 + \Gamma_2$ when we only care about splitting results, and write $\Gamma_1 + \Gamma_2$ in typing rules when the kind context Δ is clear.

The typing rules for values, computations, and handlers are given in Figure 4. Linearity-relevant parts are highlighted. The relations $\Delta; \Gamma \vdash V : A$, $\Delta; \Gamma \vdash M : C$, and $\Delta; \Gamma \vdash H : C \rightrightarrows D$, state respectively that: value V has type A, computation M has type C and handler H has type $C \rightrightarrows D$ in contexts Δ and Γ. As usual, the type contexts and types are well formed under the kind contexts.

The T-Var rule requires the remaining context to be unlimited. The T-Abs and T-TAbs rules check the value linearity of functions and polymorphic computations against that of the context via

$\boxed{\Delta \vdash \Gamma : Y}$

$$\textsc{L-Empty}\ \frac{}{\Delta \vdash \cdot : Y} \qquad \textsc{L-Extend}\ \frac{\Delta \vdash \Gamma : Y \qquad \Delta \vdash A : \mathsf{Type}^{Y}}{\Delta \vdash (\Gamma, x : A) : Y}$$

$\boxed{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}$

$$\textsc{C-Empty}\ \frac{}{\Delta \vdash \cdot = \cdot + \cdot} \qquad \textsc{C-Unl}\ \frac{\Delta \vdash A : \mathsf{Type}^{\bullet} \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)}$$

$$\textsc{C-LinLeft}\ \frac{\Delta \vdash A : \mathsf{Type}^{\circ} \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + \Gamma_2} \qquad \textsc{C-LinRight}\ \frac{\Delta \vdash A : \mathsf{Type}^{\circ} \qquad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = \Gamma_1 + (\Gamma_2, x : A)}$$

Fig. 3. Linearity of Contexts and Context Splitting

$\boxed{\Delta; \Gamma \vdash V : A}$ $\boxed{\Delta; \Gamma \vdash M : C}$ $\boxed{\Delta; \Gamma \vdash H : C \rightrightarrows D}$

$$\textsc{T-Var}\ \frac{\Delta \vdash \Gamma : \bullet}{\Delta; \Gamma, x : A \vdash x : A} \qquad \textsc{T-Abs}\ \frac{\Delta \vdash \Gamma : Y \qquad \Delta \vdash A : \mathsf{Type}^{Y'} \qquad \Delta; \Gamma, x : A \vdash M : C}{\Delta; \Gamma \vdash \lambda^{Y} x^{A}.\, M : A \to^{Y} C} \qquad \textsc{T-TAbs}\ \frac{\Delta \vdash \Gamma : Y \qquad \alpha \notin \mathsf{ftv}(\Gamma) \qquad \Delta, \alpha : K; \Gamma \vdash M : C}{\Delta; \Gamma \vdash \Lambda^{Y} \alpha^{K}.\, M : \forall^{Y} \alpha^{K}.\, C}$$

$$\textsc{T-App}\ \frac{\Delta; \Gamma_1 \vdash V : A \to^{Y} C \qquad \Delta; \Gamma_2 \vdash W : A}{\Delta; \Gamma_1 + \Gamma_2 \vdash V\, W : C} \qquad \textsc{T-TApp}\ \frac{\Delta; \Gamma \vdash V : \forall^{Y} \alpha^{K}.\, C \qquad \Delta \vdash T : K}{\Delta; \Gamma \vdash V\, T : C[T/\alpha]} \qquad \textsc{T-Return}\ \frac{\Delta; \Gamma \vdash V : A \qquad \Delta \vdash E : \mathsf{Effect}}{\Delta; \Gamma \vdash (\mathbf{return}\ V)^{E} : A\,!\,E}$$

$$\textsc{T-Do}\ \frac{E = \{\ell : A \twoheadrightarrow^{Y} B; R\} \qquad \Delta; \Gamma \vdash V : A \qquad \Delta \vdash E : \mathsf{Effect}}{\Delta; \Gamma \vdash (\mathbf{do}\ \ell\ V)^{E} : B\,!\,E} \qquad \textsc{T-Seq}\ \frac{\Delta; \Gamma_1 \vdash M : A\,!\,\{R\} \qquad \Delta; \Gamma_2, x : A \vdash N : B\,!\,\{R\} \qquad \Delta \vdash \Gamma_2 : Y \qquad \Delta \vdash R : Y}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^{Y}\, x \leftarrow M\ \mathbf{in}\ N : B\,!\,\{R\}}$$

$$\textsc{T-Handle}\ \frac{\Delta; \Gamma_1 \vdash H : C \rightrightarrows D \qquad \Delta; \Gamma_2 \vdash M : C}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

$$\textsc{T-Handler}\ \frac{\begin{array}{c} H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \\ C = A\,!\,\{(\ell_i : A_i \twoheadrightarrow^{Y_i} B_i)_i; R\} \qquad D = B\,!\,\{(\ell_i : P)_i; R\} \\ \Delta \vdash \Gamma : \bullet \qquad \Delta; \Gamma, x : A \vdash M : D \\ [\Delta; \Gamma, p_i : A_i, r_i : B_i \to^{Y_i} D \vdash N_i : D]_i \end{array}}{\Delta; \Gamma \vdash H : C \rightrightarrows D}$$

Fig. 4. Typing Rules for $\mathrm{F}^{\circ}_{\mathrm{eff}}$

the premise $\Delta \vdash \Gamma : Y$. The typing rules for function application and type application are standard (T-App and T-TApp). Note that we need to split the context in the T-App rule to avoid duplicating linear variables. The T-Return rule does not constrain the effects. The T-Do rule ensures that

$$\boxed{\Delta \vdash R \leqslant R' : K}$$

$$\frac{\Delta \vdash R : K}{\Delta \vdash R \leqslant R : K} \qquad \frac{\Delta \vdash R_1 \leqslant R_2 : K \qquad \Delta \vdash R_2 \leqslant R_3 : K}{\Delta \vdash R_1 \leqslant R_3 : K} \qquad \frac{\Delta \vdash \mu : K}{\Delta \vdash \cdot \leqslant \mu : K}$$

$$\frac{\Delta \vdash P : \mathsf{Presence}^Y \quad \Delta \vdash R_1 \leqslant R_2 : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}{}^Y}{\Delta \vdash \ell : \mathsf{Abs}; R_1 \leqslant \ell : P; R_2 : \mathsf{Row}_{\mathcal{L}}{}^Y} \qquad \frac{\Delta \vdash P : \mathsf{Presence}^Y \quad \Delta \vdash R_1 \leqslant R_2 : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}{}^Y}{\Delta \vdash \ell : P; R_1 \leqslant \ell : P; R_2 : \mathsf{Row}_{\mathcal{L}}{}^Y}$$

Fig. 5. Trivial Subtyping for Effect Row Types

the operation ℓ and its parameter V agree with the effect signature E. The T-Handle rule uses a handler of type $C \Rightarrow D$ to handle a computation of type C.

The T-Handler rule checks that (deep) handlers must not use any linear variables via the premise $\Delta \vdash \Gamma : \bullet$ because they are recursively applied during evaluation. More importantly, it connects the control-flow linearity of operations with the value linearity of resumption functions. In the typing judgement of each operation clause $\ell_i : A_i \twoheadrightarrow^{Y_i} B_i$, the continuation r_i is given the value linearity Y_i, which is exactly the control-flow linearity of ℓ_i that restricts the use of ℓ_i's continuation. Concretely, when $Y_i = \circ$, the continuation of ℓ_i may use some linear resources. Making r_i linear guarantees that they are used exactly once. When $Y_i = \bullet$, the continuation of ℓ_i must not use any linear resources and r_i is unlimited. Note that the subkinding relation $\mathsf{Row}^\circ \leq \mathsf{Row}^\bullet$ does not influence the handling behaviour, because the T-Handler rule uses the linearity annotations on operation signatures.

The T-Seq rule for sequencing is the most important rule for tracking control-flow linearity, because this is the primary source of sequential control flow in a fine-grain call-by-value calculus. Though handling is another source of sequential control flow, deep handlers are unlimited and cannot influence control-flow linearity. We will discuss the extension of shallow handlers which may capture linear resources and influence control-flow linearity in Section 6.

Remember that for $\mathbf{let}^Y x \leftarrow M \ \mathbf{in} \ N$, the linearity annotation Y indicates the control-flow linearity of N which determines how many times the control can enter N. Concretely, when $Y = \circ$, N may use some linear variables bound outside ($\Delta \vdash \Gamma_2 : \circ$), and all operations in M should be control-flow linear ($\Gamma \vdash R : \circ$); when $Y = \bullet$, N cannot use any linear variables from the context ($\Delta \vdash \Gamma_2 : \bullet$), and operations in M have no restriction on their control-flow linearity ($\Delta \vdash R : \bullet$). The dubiousWrite$_\checkmark$ in Section 2.2 is an example. Note that technically, the third sequencing $\mathbf{let}^\circ f' \leftarrow \mathsf{write}\,(s, f) \ \mathbf{in} \ \mathsf{close}\, f'$ can be changed to $\mathbf{let}^\bullet$ because no linear variable bound outside is used by the context $\mathbf{let}\, f' \leftarrow \_ \ \mathbf{in} \ \mathsf{close}\, f'$.

As we observed by the function verboseClose in Section 2.4, the fact that the T-Seq rule requires the M and N to have the same effect type is too restrictive for tracking control-flow linearity. We can improve it by using a trivial subtyping relation between effect types as follows.

$$\text{T-SeqSub}\quad \frac{\Delta; \Gamma_1 \vdash M : A \,!\, \{R_1\} \qquad \Delta; \Gamma_2, x : A \vdash N : B \,!\, \{R_2\} \qquad \Delta \vdash \Gamma_2 : Y \qquad \Delta \vdash R_1 : Y \qquad \Delta \vdash R_1 \leqslant R : K \qquad \Delta \vdash R_2 \leqslant R : K}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow M \ \mathbf{in} \ N : B \,!\, \{R\}}$$

The trivial subtyping relation on effect row types are shown in Figure 5. The judgement $\Delta \vdash R \leqslant R' : K$ makes it explicit that R and R' are well kinded and can be given kind K under kind context Δ. It simply requires that all operation labels with their signatures and row variable in R must also

appear in R'. This subtyping relation does not allow non-trivial subtyping between row variables. We consider a more expressive alternative using qualified types in Section 5.

3.3 Operational Semantics

$$\begin{array}{lll}
\textsc{E-App} & (\lambda^Y x^A.M)\ V \rightsquigarrow M[V/x] & \\
\textsc{E-TApp} & (\Lambda^Y \alpha^K.M)\ T \rightsquigarrow M[T/\alpha] & \\
\textsc{E-Seq} & \mathbf{let}^Y x \leftarrow (\mathbf{return}\ V)^E\ \mathbf{in}\ N \rightsquigarrow N[V/x] & \\
\textsc{E-Ret} & \mathbf{handle}\ (\mathbf{return}\ V)^E\ \mathbf{with}\ H \rightsquigarrow N[V/x], & \text{where } (\mathbf{return}\ x \mapsto N) \in H \\
\textsc{E-Op} & \mathbf{handle}\ \mathcal{E}[(\mathbf{do}\ \ell\ V)^E]\ \mathbf{with}\ H \rightsquigarrow N[V/p, (\lambda^Y y^B.\mathbf{handle}\ \mathcal{E}[(\mathbf{return}\ y)^E]\ \mathbf{with}\ H)/r], & \\
 & \text{where } \ell \notin \mathsf{bl}(\mathcal{E}), (\ell\ p\ r \mapsto N) \in H, \text{ and } (\ell : A \to^Y B) \in E & \\
\textsc{E-Lift} & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], & \text{if } M \rightsquigarrow N
\end{array}$$

$$\text{Evaluation contexts} \quad \mathcal{E} ::= [\ ] \mid \mathbf{let}^Y x \leftarrow \mathcal{E}\ \mathbf{in}\ N \mid \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H$$

$$\mathsf{bl}([\ ]) = \emptyset \qquad \mathsf{bl}(\mathbf{let}^Y x \leftarrow \mathcal{E}\ \mathbf{in}\ N) = \mathsf{bl}(\mathcal{E}) \qquad \mathsf{bl}(\mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H) = \mathsf{bl}(\mathcal{E}) \cup \mathsf{dom}(H)$$

Fig. 6. Small-step Operational Semantics of $\mathrm{F}^{\circ}_{\mathrm{eff}}$

Figure 6 gives a standard small-step operational semantics for $\mathrm{F}^{\circ}_{\mathrm{eff}}$ [Hillerström et al. 2020a]. It is clear from the definition of evaluation contexts that let-binding and handling are indeed the only two constructs that influence the control flow. The function $\mathsf{bl}(-)$ computes the set of *bound operation labels* in an evaluation context $\mathcal{E}$, i.e. the operation labels for which a suitable handler has been installed. The purpose of this function is to ensure that any operation invocation $(\mathbf{do}\ \ell\ V)$ is always handled by the innermost suitable handler.

3.4 Metatheory

We now prove a type soundness result for $\mathrm{F}^{\circ}_{\mathrm{eff}}$. First we define normal forms of computations.

Definition 3.1 (Computation Normal Forms). We say a computation M is in a normal form with respect to E, if it is either of the form $M = (\mathbf{return}\ V)^{E'}$ or $M = \mathcal{E}[(\mathbf{do}\ \ell\ V)^{E'}]$ for $\ell \in E$ and $\ell \notin \mathsf{bl}(\mathcal{E})$.

Syntactic type soundness of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ relies on progress and subject reduction. The proofs can be found in Appendices A.2 and A.3.

Theorem 3.2 (Progress). *If $\vdash M : A\,!\,E$, then either there exists N such that $M \rightsquigarrow N$ or M is in a normal form with respect to E.*

Theorem 3.3 (Subject reduction). *If $\Delta; \Gamma \vdash M : C$ and $M \rightsquigarrow N$, then $\Delta; \Gamma \vdash N : C$.*

We now show that our tracking of value linearity and control-flow linearity in the type system is sound, by proving that linear variables never appear in terms that are claimed to be unlimited. In $\mathrm{F}^{\circ}_{\mathrm{eff}}$, a term is claimed to be unlimited if it appears in an unlimited value, a control-flow-unlimited context, or a deep handler. The following theorem covers all three of these cases.

Theorem 3.4 (Unlimited is unlimited).

1. *Unlimited values are unlimited: if $\Delta; \Gamma \vdash V : A$ and $\Delta \vdash A : \bullet$, then $\Delta \vdash \Gamma : \bullet$.*
2. *Unlimited continuations are unlimited: if $\Delta; \Gamma \vdash \mathcal{E}[(\mathbf{do}\ \ell\ V)^E] : C$ for $E = \{\ell : A \twoheadrightarrow^{\bullet} B; R\}$ and $\ell \notin \mathsf{bl}(\mathcal{E})$, then there exists $\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2$ such that $\Delta \vdash \Gamma_1 : \bullet$ and $\Delta; \Gamma_1, y : B \vdash \mathcal{E}[(\mathbf{return}\ y)^E] : C$.*

$$\text{L-App} \quad (\lambda^Y x^A.M)\ V \quad \overset{\mathcal{S}}{\underset{\emptyset}{\rightsquigarrow}}\ M[V'/x], \text{ where } (V', \mathcal{S}) = \mathsf{tag}(V)$$

$$\text{L-TApp} \quad (\Lambda^Y \alpha^K.M)\ T \quad \overset{\emptyset}{\underset{\emptyset}{\rightsquigarrow}}\ M[T/\alpha]$$

$$\text{L-Seq} \quad \mathbf{let}^Y x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \quad \overset{\mathcal{S}}{\underset{\emptyset}{\rightsquigarrow}}\ N[V'/x], \text{ where } (V', \mathcal{S}) = \mathsf{tag}(V)$$

$$\text{L-Ret} \quad \mathbf{handle}\ (\mathbf{return}\ V)^E\ \mathbf{with}\ H \quad \overset{\mathcal{S}}{\underset{\emptyset}{\rightsquigarrow}}\ N[V'/x], \text{ where } (\mathbf{return}\ x \mapsto N) \in H, (V', \mathcal{S}) = \mathsf{tag}(V)$$

$$\text{L-Op} \quad \mathbf{handle}\ \mathcal{E}[(\mathbf{do}\ \ell\ V)^E]\ \mathbf{with}\ H \quad \overset{\mathcal{S}}{\underset{\emptyset}{\rightsquigarrow}}\ N[V'/p, W'/r], \text{ where } \ell \notin \mathsf{bl}(\mathcal{E}),\ (\ell\ p\ r \mapsto N) \in H,\ (\ell : A \twoheadrightarrow^Y B) \in E,\ W = \lambda^Y y^B.\mathbf{handle}\ \mathcal{E}[(\mathbf{return}\ y)^E]\ \mathbf{with}\ H,\ (V', \mathcal{S}_1) = \mathsf{tag}(V), (W', \mathcal{S}_2) = \mathsf{tag}(W), \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$$

$$\text{L-Remove} \quad \mathcal{F}[V^\circ] \quad \underset{\{V^\circ\}}{\overset{\emptyset}{\rightsquigarrow}}\ \mathcal{F}[V]$$

$$\text{L-Lift} \quad \mathcal{E}[M] \quad \overset{\mathcal{S}}{\underset{\mathcal{T}}{\rightsquigarrow}}\ \mathcal{E}[N], \quad \text{if } M\ \overset{\mathcal{S}}{\underset{\mathcal{T}}{\rightsquigarrow}}\ N$$

$$\text{Evaluation contexts} \quad \mathcal{E} ::= [\,] \mid \mathbf{let}^Y x \leftarrow \mathcal{E}\ \mathbf{in}\ N \mid \mathbf{handle}\ \mathcal{E}\ \mathbf{with}\ H$$

$$\text{Tag-removing contexts} \quad \mathcal{F} ::= [\,]\ V \mid [\,]\ T$$

Fig. 7. Linearity-aware Small-step Operational Semantics of $\mathrm{F}^\circ_{\mathrm{eff}}$

3. Deep handlers are unlimited: if $\Delta; \Gamma \vdash H : C \rightrightarrows D$, *then* $\Delta \vdash \Gamma : \bullet$.

The proof can be found in Appendix A.1.

However, Theorem 3.4 only cares about the static tracking of linear variables. It says nothing about the use of linear values during evaluation directly. In the next section, we prove that in $\mathrm{F}^\circ_{\mathrm{eff}}$ no linear value is ever discarded or duplicated during evaluation, by defining a linearity-aware semantics inspired by Walker [2005], Mazurak et al. [2010], and Morris [2016].

3.5 Linearity Safety of Evaluation

In this section, we design a linearity-aware semantics of $\mathrm{F}^\circ_{\mathrm{eff}}$, extending the small-step operational semantics to track the introduction and elimination of linear values, and prove that all linear values are used exactly once during evaluation.

We first extend the syntax of values with values marked with linear tags V° to indicate linear values during evaluation. The typing rules simply ignore the linear tags.

$$\text{Values} \quad V ::= \cdots \mid V^\circ$$

We restrict attention to closed computations and define two auxiliary functions $\mathsf{lin}(V)$ and $\mathsf{tag}(V)$ for closed values as follows.

$$\mathsf{lin}(V) = \begin{cases} \text{true} & \text{if } \cdot; \cdot \vdash V : A \text{ and } \cdot \nvdash A : \bullet \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathsf{tag}(V) = \begin{cases} (V^\circ, \{V^\circ\}) & \text{if } \mathsf{lin}(V) \text{ and } V \neq W^\circ \text{ for any } W \\ (V, \emptyset) & \text{otherwise} \end{cases}$$

The predicate $\mathsf{lin}(V)$ holds when V is a genuine linear value as opposed to an unlimited value that has been upcast to be linear by subkinding. The operation $\mathsf{tag}(V)$ tags a value as linear if it is and has not been tagged, and yields a pair of the possibly tagged V and a multiset containing the value if it is newly tagged and nothing otherwise.

The linearity-aware semantics is given in Figure 7. We augment the previous reduction relation $M \rightsquigarrow N$ with two multi-sets $M\ \overset{\mathcal{S}}{\underset{\mathcal{T}}{\rightsquigarrow}}\ N$, where $\mathcal{S}$ contains the linear values introduced by this

reduction step, and $\mathcal{T}$ contains the linear values eliminated by this reduction step. Note that in $\mathrm{F}^{\circ}_{\mathrm{eff}}$, we cannot duplicate or discard a value before we bind it. We introduce linear values at the first time they are bound to variables (L-App, L-Seq, L-Ret and L-Op). Take L-App for example. When V is a non-tagged real linear value (the first case of $\mathrm{tag}(V)$), we tag it and add it to the multiset of introduced linear values. Otherwise, V is either not really linear or has been tagged already (which implies that we have already introduced it). We do not need to update the multisets. We eliminate linear values when they are destructed (L-Remove). As we only have term abstraction and type abstraction as value constructors, the tag-removing contexts $\mathcal{F}$ capture the elimination of these two cases. It is easy to extend the linearity-aware semantics with other value constructors. The relationship between the two semantics is straightforward: erasing the linear tags from the linearity-aware semantics yields the original semantics.

We write $\mathscr{L}(M)$, $\mathscr{L}(V)$, $\mathscr{L}(\mathcal{E})$ and $\mathscr{L}(\mathcal{F})$ for the multisets of tagged linear values within M, V, $\mathcal{E}$, and $\mathcal{F}$, respectively. They are given by the homomorphic extension of the following equation.

$$\mathscr{L}(V^{\circ}) = \{V^{\circ}\} \cup \mathscr{L}(V)$$

We define the notion of linear safety similarly to Theorem 3.4. A term is linear safe if there are no tagged linear values in terms that are claimed to be unlimited.

Definition 3.5 (Linear Safety). A well-typed computation M or value V is *linear safe* if and only if:

(1) For every value subterm W of the form $\lambda^{\bullet}x^{A}.N$ or $\Lambda^{\bullet}\alpha^{K}.N$, $\mathscr{L}(W) = \emptyset$.
(2) For every computation subterm N of the form $\mathcal{E}[(\mathbf{do}\ \ell\ V)^{\{\ell:A\twoheadrightarrow^{\bullet}B\,;R\}}]$ where $\ell \notin \mathrm{bl}(\mathcal{E})$, $\mathscr{L}(\mathcal{E}) = \emptyset$.
(3) For every handler subterm H, $\mathscr{L}(H) = \emptyset$.

(An alternative way to read Item 1 is as "for every value subterm W with an unlimited type".)

Finally, the following theorem states that linear safety is preserved by evaluation, and tagged linear values are not duplicated or discarded during evaluation.

Theorem 3.6 (Reduction Safety). *For any closed, well-typed and linear safe computation M in $\mathrm{F}^{\circ}_{\mathrm{eff}}$, if $M \overset{\mathcal{S}}{\underset{\mathcal{T}}{\rightsquigarrow}} N$, then N is linear safe and $\mathscr{L}(M) \cup \mathcal{S} = \mathscr{L}(N) \cup \mathcal{T}$.*

The proof can be found in Appendix A.4. Note that tracking linear values explicitly during evaluation is important for showing that they are indeed used safely. Otherwise, it is even unclear how to state what reduction safety means in the original semantics.

4 CONTROL-FLOW LINEARITY IN LINKS

In this section, we describe our implementation of control-flow linearity tracking in Links. The implementation fixes a long-standing type soundness bug in Links arising from the interaction between session types and effect handlers, as we described in the introduction.

Links is an ML-style language with type inference, linearly typed session types (based on F° [Lindley and Morris 2017]), and a row-based effect type system [Hillerström and Lindley 2016]. In Links we write `Unl` for • and `Any` for ∘. The latter is `Any` as *any* value can be soundly used once. The subkinding relation $\vdash \mathtt{Type}^{\bullet} \leq \mathtt{Type}^{\circ}$ ($\mathtt{Unl} \leq \mathtt{Any}$) allows type variables of kind `Any` to be unified with types of either kind. This allows us to write functions that may accept both linear and nonlinear values, e.g. the identity function `fun id(x){x} : (a::Any) -> (a::Any)`. Here, we can instantiate the type variable `a` to a linear type, such as `!Int.End`, or an unlimited type, such as `Int`.

To make type inference deterministic, Links makes use of two different keywords for defining unlimited functions and linear functions, which are **`fun`** and **`linfun`** respectively. For instance, we can define a channel version of the function faithfulWrite in Section 2.1 as follows.

```
fun faithfulSend(c) { linfun (s) { var c = send(s, c); close(c) } }
```

The inferred type is `(!(a::Any).End) -> (a::Any) ~@ ()`. The `faithfulSend` function takes a polymorphic channel `c` and returns a linear function (indicated by `~@` instead of the usual arrow `~>`) that sends a polymorphic value s over the channel `c`. If we wanted to we could restrict the inferred type of the channel `c` and the input s by supplying a type annotation to either.

To track control-flow linearity we repurpose the existing effect system and add two new control flow kinds `Any` (for $\bullet$) and `Lin` (for $\circ$) to signify whether a given context allows control flow to be unlimited or linear. We further add a new effectful operation space for control-flow-linear operations, which is syntactically denoted by the arrow `=@`, in addition to the existing operation space denoted by `=>`. The subkinding relation $\vdash \text{Row}^\circ \leq \text{Row}^\bullet$ (`Lin` $\leq$ `Any`) is implemented by allowing row variables of kind `Any` to be unified with both control-flow-linear and unlimited operations and other row variables of arbitrary kinds. In contrast, row variables of kind `Lin` can only be unified with control-flow-linear operations and row variables of kind `Lin`. The change from `Unl` to `Lin` is consistent with the duality between value linearity and control-flow linearity.

Since Links is a practical programming language, sequencing is often implicit. Instead of writing linearity annotations on all sequencing, we assume that control-flow linearity is unlimited by default, and introduce the keyword **`xlin`** to switch the control-flow linearity to linear. We also add the construct **`lindo`** to invoke control-flow-linear operations in addition to the existing **`do`** for control-flow-unlimited operations. To illustrate the use of these extensions, let us consider a channel version of the function dubiousWrite$_\checkmark$ from Section 2.2.

```
sig dubiousSend : (!String.End) {Choose:() =@ Bool|_::Lin}~> ()
fun dubiousSend(c) {xlin; var c = send(if (lindo Choose) "A" else "B", c); close(c)}
```

The `dubiousSend` takes a channel `c`, non-deterministically sends `"A"` or `"B"` through it depending on the result of the operation `Choose`, and closes the remaining channel. We use **`xlin`** to switch the control-flow linearity to linear so that we can use the linear channel `c` and must use the control-flow-linear operation `Choose:() =@ Bool` with the keyword **`lindo`**. If we replace **`lindo`** with **`do`** then Links correctly rejects the code as the continuation captures the linear endpoint `c`. The example from the introduction will be rejected for the same reason. For linear effect handlers, we use the linear arrow syntax `=@` to bind linear continuations of control-flow-linear operations.

```
fun(c) {handle ({xlin; dubiousSend(c)}) {case <Choose =@ r> -> xlin; r(true)} }
```

Here, we interpret the operation `Choose` as `true`. The use of **`xlin`** in the `Choose`-clause is necessary because the reified continuation r is linear. As the continuation is used linearly, Links correctly accepts this program.

Our implementation works well with previous programs using the effect handler feature in Links and fixes the type soundness bug. However, being based on F$^\circ$, Links suffers from the limitations outlined in Section 2. In the next section, we present a considerably more expressive calculus, Q$^\circ_{\text{eff}}$, which uses qualified types for both linearity and effects, enabling a much more fine-grained analysis of control-flow linearity, and avoiding the need to distinguish between linear and non-linear variants of term syntax. We leave the implementation of Q$^\circ_{\text{eff}}$ to future work.

5 AN IMPLICIT CALCULUS WITH QUALIFIED TYPES

In this section, we propose Q$^\circ_{\text{eff}}$, an ML-style calculus which enhances F$^\circ_{\text{eff}}$ (and its implementation in Links) in two directions: minimising syntactic overheads and improving accuracy of control-flow linearity tracking. The core idea is to use qualified types for both linear types and effect types. The qualified linear type system is inspired by Quill [Morris 2016], which eliminates the linearity annotations on terms and supports principal types. The qualified effect system is inspired by the row containment predicate of Rose [Morris and McKinna 2019] and the subtyping-based effect

system of Eff [Karachalias et al. 2020; Pretnar 2014], which allows non-trivial subtyping constraints between row variables.

5.1 Syntax

Figure 8 shows the syntax of qualified types of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$. We name some syntactic categories for defining meta functions. The remaining syntax is given in full in Appendix B.1, which is mostly identical to that of $\mathrm{F}^{\circ}_{\mathrm{eff}}$, except that we introduce generalising let-bindings **let** $x = V$ **in** M to replace explicit type abstraction and implicit instantiation in place of type application and remove all type annotations and linearity annotations.

Linearity	$Y ::= \phi \mid \bullet \mid \circ$	Qualified types	$\rho ::= A \mid \pi \Rightarrow \rho$
Types	$\tau ::= A \mid R \mid Y$	Type schemes	$\mathsf{TySch} \ni \sigma ::= \rho \mid \forall \alpha.\sigma$
Predicates	$\mathsf{Pred} \ni \pi ::= \tau_1 \preceq \tau_2 \mid R_1 \leqslant R_2$	Type contexts	$\mathsf{Env} \ni \Gamma ::= \cdot \mid \Gamma, x : \sigma$
	$\mid \ R \perp \mathcal{L}$	Predicate sets	$\mathsf{PSet} \ni P ::= \cdot \mid P, \pi$

Fig. 8. Syntax of Qualified Types of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$

Linearity. In addition to concrete linearities $\circ$ and $\bullet$, $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ has linearity variables ϕ. This is essential to have principal types and more expressive constraints. For example, the identity function $\lambda x.\mathbf{return}\ x$ can be given the principal type $\forall \alpha\, \mu\, \phi.\, \alpha \to^{\phi} \alpha\, !\, \{\mu\}$, which can be instantiated to either a linear function (by instantiating ϕ to $\circ$) or an unlimited function (by instantiating ϕ to $\bullet$).

Qualified types. The syntactic category τ includes value types, row types, and linearity types. Qualified types ρ restrict value types by predicates. The linearity predicate $\tau_1 \preceq \tau_2$ means the linearity of τ_1 is less than τ_2 (e.g., $\bullet \preceq \circ$). Note that we allow directly using value types and row types in the linearity predicates, since every value type has its value linearity, and every effect row type has its control-flow linearity. The row predicates $R_1 \leqslant R_2$ means R_1 is a sub-row of R_2, and $R \perp \mathcal{L}$ means R does not contain labels in $\mathcal{L}$.

Kinding. For conciseness we omit kinds and infer the kind of a type variable from its name. As usual, we let α range over value types, μ range over row types, and ϕ range over linearity types. We also let α range over all of them in the definition of type schemes $\forall \alpha.\sigma$. All rows are assumed to be well-formed (no duplicated labels). To simplify type inference, the predicate $\mu \perp \mathcal{L}$ will be used in place of kinds $\mathsf{Row}_{\mathcal{L}}$ to track labels that may not occur in rows. This is just a convenience, though, as the corresponding kinds of row type variables can be computed from the inferred types.

5.2 Typing

Figure 9 gives representative syntax-directed typing rules for $\mathrm{Q}^{\circ}_{\mathrm{eff}}$; the remaining rules are given in full in Appendix B.2. The judgement $P \mid \Gamma \vdash M : C$ states that, under predicate assumptions P and typing assumptions Γ, the term M has type C, and similarly for the judgements for values and handlers. As usual for qualified type systems, the typing rules depend on an entailment relation $P \vdash \pi$ (and an auxiliary relation $P \vdash \Gamma \preceq \tau$), discussed in the following section.

Rule Q-Let demonstrates the treatment of linearity in $\mathrm{Q}^{\circ}_{\mathrm{eff}}$. We divide the context in three: Γ_1 is used exclusive in the bound term V, Γ_2 is used exclusively in the body M, and Γ is used in both (and so its types must be unlimited).

Rule Q-Do demonstrates the use of constraints in $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ to generalise subtyping between effect rows. It states that if V is a value of type A_ℓ, then **do** $\ell\ V$ has result type B_ℓ and effect row R. We assume that the parameter and result types of operations are given by an implicit global context

$\boxed{P \mid \Gamma \vdash V : A}$ $\boxed{P \mid \Gamma \vdash M : C}$ $\boxed{P \mid \Gamma \vdash H : C \rightrightarrows D}$

Q-Let
$$\frac{Q \mid \Gamma_1, \Gamma \vdash V : A \qquad \sigma = \mathsf{gen}((\Gamma_1, \Gamma), Q \Rightarrow A) \qquad P \mid \Gamma_2, \Gamma, x : \sigma \vdash M : C \qquad P \vdash \Gamma \preceq \bullet}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \textbf{let } x = V \textbf{ in } M : C}$$

Q-Do
$$\frac{P \mid \Gamma \vdash V : A_\ell \qquad P \vdash \{\ell : A_\ell \twoheadrightarrow^Y B_\ell\} \leqslant R}{P \mid \Gamma \vdash \textbf{do } \ell\ V : B_\ell\,!\,\{R\}}$$

Q-Seq
$$\frac{P \mid \Gamma_1, \Gamma \vdash M : A\,!\,\{R_1\} \qquad P \mid \Gamma_2, \Gamma, x : A \vdash N : B\,!\,\{R_2\} \qquad P \vdash R_1 \leqslant R \qquad P \vdash R_2 \leqslant R \qquad P \vdash \Gamma_2 \preceq R_1 \qquad P \vdash \Gamma \preceq \bullet}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \textbf{let } x \leftarrow M \textbf{ in } N : B\,!\,\{R\}}$$

Q-Handler
$$\frac{H = \{\textbf{return } x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \qquad C = A\,!\,\{(\ell_i : A_i \twoheadrightarrow^{Y_i} B_i)_i; R_1\} \qquad D = B\,!\,\{R_2\} \qquad P \mid \Gamma, x : A \vdash M : D \qquad [P \mid \Gamma, p_i : A_i, r_i : B_i \rightarrow^{Y_i} D \vdash N_i : D]_i \qquad P \vdash \Gamma \preceq \bullet \qquad P \vdash R_1 \leqslant R_2 \qquad P \vdash R_1 \perp \{\ell_i\}_i}{P \mid \Gamma \vdash H : C \rightrightarrows D}$$

where $\mathsf{gen}(\Gamma, \rho) = \forall(\mathsf{ftv}(\rho) \backslash \mathsf{ftv}(\Gamma)).\rho$.

Fig. 9. Selected Syntax-directed Typing Rules for $\mathrm{Q}^{\circ}_{\mathrm{eff}}$

$\Pi = \{\ell_1 : A_{\ell_1} \twoheadrightarrow B_{\ell_1}, \cdots\}$. R must license effect ℓ. We again rely on entailment: the constraints P must be sufficient to show that the singleton row $\{\ell : A_\ell \twoheadrightarrow^Y B_\ell\}$ is contained within R.

Rule Q-Seq demonstrates the remaining novelty of qualified types in $\mathrm{Q}^{\circ}_{\mathrm{eff}}$. Several of its uses of entailment follow the previous patterns. The bindings in Γ are available in both M and N, so $P \vdash \Gamma \preceq \bullet$ requires that their types be unlimited. We want flexibility in combining the effects in M and N, so the conditions $P \vdash R_i \leqslant R$ assure that the effects of each are included in the effects of the entire computation. This allows us to avoid having to unify row types in examples like sandwichClose (Section 2.4) which causes inaccuracy for tracking control-flow linearity. Finally, N is in the continuation of all operations in M, so the value linearity of types in Γ_2 must be less than the control-flow linearity of operations in R_1. Note that the two kinding judgements in T-Seq in Figure 4 are now combined into one entailment judgement $P \vdash \Gamma_2 \preceq R_1$. The duality we have identified between value linearity and control-flow linearity is reflected by the fact that value types appear on the left of $\preceq$ and effect row types appear on the right.

Rule Q-Handler uses the lacking predicate $P \vdash R_1 \perp \{\ell_i\}_i$ to ensure that the handled operations are not in the remaining part of the input effect row R_1, and requires R_1 to be a sub-row of the output effect row R_2. This is used to allow the handled operations ℓ_i to appear in R_2.

5.3 Entailment

Figure 10 defines the entailment relations between predicates $P \vdash Q$. It also defines an auxiliary entailment relation $P \vdash \Gamma \preceq \tau$ which compares the linearity of all variables in Γ and τ. The algorithmic version of these relations will be given in Section 5.5.

These two entailment relations are both defined as the conjunction of sub-relations as indicated by P-PredSet and P-Context. For $P \vdash Q$, we only need to use entailment relations of the form $P \vdash \pi$. The P-Subsume is standard. The linearity predicate $\preceq$ is reflexive (P-Refl), with $\circ$ as top (P-Lin) and $\bullet$ as bottom (P-Unl) elements. The two-way rules P-Fun and P-Row define the linearity of functions and rows. We make use of the fact that in the linearity predicates generated by typing rules, functions only appear on the left, and rows only appear on the right. Here we do not include

$\boxed{P \vdash \pi}\ \boxed{P \vdash Q}\ \boxed{P \vdash \sigma \preceq \tau}\ \boxed{P \vdash \Gamma \preceq \tau}$

$$\textsc{P-Subsume}\ \frac{\pi \in P}{P \vdash \pi} \qquad \textsc{P-Refl}\ \frac{}{P \vdash \tau \preceq \tau} \qquad \textsc{P-Lin}\ \frac{}{P \vdash \tau \preceq \circ} \qquad \textsc{P-Unl}\ \frac{}{P \vdash \bullet \preceq \tau} \qquad \textsc{P-Fun}\ \frac{P \vdash Y \preceq \tau}{P \vdash (A \to^{Y} C) \preceq \tau}$$

$$\textsc{P-Row}\ \frac{[P \vdash \tau \preceq Y]_{(l:A \twoheadrightarrow^{Y} B) \in R} \quad P \vdash \tau \preceq \mu \text{ when } \mu \in R}{P \vdash \tau \preceq R} \qquad \textsc{P-Sub}\ \frac{\mathsf{set}(R_1) \subseteq \mathsf{set}(R_2)}{P \vdash R_1 \leqslant R_2} \qquad \textsc{P-Lack}\ \frac{\mathsf{dom}(R) \cap \mathcal{L} = \emptyset}{P \vdash R \perp \mathcal{L}} \qquad \textsc{P-PredSet}\ \frac{[P \vdash \pi]_{\pi \in Q}}{P \vdash Q}$$

$$\textsc{P-Quantifier}\ \frac{P \vdash [\tau'/\alpha]\sigma \preceq \tau \text{ for some } \tau'}{P \vdash (\forall \alpha.\sigma) \preceq \tau} \qquad \textsc{P-Qualifier}\ \frac{P \vdash \pi \qquad P \vdash \rho \preceq \tau}{P \vdash (\pi \Rightarrow \rho) \preceq \tau} \qquad \textsc{P-Context}\ \frac{[P \vdash \sigma \preceq \tau]_{(x:\sigma) \in \Gamma}}{P \vdash \Gamma \preceq \tau}$$

Fig. 10. Entailment Relations for Predicates and other Judgement Relations

entailment rules for base types, but in practice we would have axioms like $P \vdash \mathsf{Int} \preceq \bullet$ and $P \vdash \circ \preceq \mathsf{File}$. For row predicates, we write $\mathsf{set}(R)$ for the set of all elements (comprising operation labels with their signatures and row variables) of R, and $\mathsf{dom}(R)$ for the set of all labels of R. We define the row predicates directly by set operations (P-Sub and P-Lack).

The entailment relation $P \vdash \Gamma \preceq \tau$ is defined using $P \vdash \sigma \preceq \tau$ which compares the linearity of a type scheme σ and a type τ. Our treatment of the linearity of type schemes is novel, and addresses a soundness bug in Quill. The rule P-Quantifier which characterises the linearity of polymorphic types may be surprising. It states that the linearity of a polymorphic type $\forall \alpha.\sigma$ is less than τ if there exists an instantiation of it whose linearity is less than τ. This is because the linearity of a polymorphic type should capture the linearity of values that inhabit that type. A value of a polymorphic type can be understood as the intersection of values of all possible instantiations of the type. If one of these instantiation gives a type that is less linear than τ, then the value itself must be less linear than τ no matter what other instantiations are. For example, consider the identity function $\mathsf{id} = \lambda x.\mathbf{return}\ x$ which is obviously unlimited. We give id a polymorphic type $\forall \phi\, \alpha\, \mu.\, \alpha \to^{\phi} \alpha\,!\,\{\mu\}$ to make it possible to use it as both a linear function (by instantiating ϕ to $\circ$) and an unlimited function (by instantiating ϕ to $\bullet$). Thus, we have expressive principal types for id without adding subtyping between linearity types to the type system.

The rule P-Qualifier may also be surprising. To compare the linearity of a qualified type $\pi \Rightarrow \rho$ with τ, we require the predicate π to hold and then compare the linearity of the remaining part ρ with τ. At first glance, the condition $P \vdash \pi$ may seem unnecessary: if π must hold in instantiations of this type, surely we can assume it in checking the type's linearity. However, particularly in local definitions, predicates may mention type variables *not* quantified in those schemes. We do not want to assume anything about the instantiation of those variables. Consider the following function.

$$\lambda x.\mathbf{let}\ f = \lambda().x\ \mathbf{in}\ \mathbf{return}\ (f, f)$$

The polymorphic function f can be given the principal type $\sigma = \forall \phi\, \mu.(\alpha \preceq \phi) \Rightarrow () \to^{\phi} \alpha\,!\,\{\mu\}$ where α is the type of x. Note that the constraint mentions α, which is bound outside this type scheme. Then, since f is duplicated in $\mathbf{return}\ (f, f)$, the typing of it collects the constraint $\sigma \preceq \bullet$. Obviously, we want to know from $\sigma \preceq \bullet$ that α should be unlimited since x is also duplicated. One

possible derivation of $P \vdash \sigma \preceq \bullet$ is shown as follows.

$$\dfrac{\dfrac{\dfrac{}{P \vdash \alpha \preceq \phi'} \qquad \dfrac{\dfrac{}{P \vdash \phi' \preceq \bullet}}{P \vdash () \to^{\phi'} \alpha\,!\,\{\mu'\} \preceq \bullet}\ \text{P-Function}}{P \vdash (\alpha \preceq \phi') \Rightarrow () \to^{\phi'} \alpha\,!\,\{\mu'\} \preceq \bullet}\ \text{P-Qualifier}}{P \vdash (\forall \phi\, \mu.(\alpha \preceq \phi) \Rightarrow () \to^{\phi} \alpha\,!\,\{\mu\}) \preceq \bullet}\ \text{P-Quantifier}$$

In P-Quantifier we instantiate ϕ and μ with variables ϕ' and μ'. In order to prove $\sigma \preceq \bullet$ from P, we must then prove $\alpha \preceq \phi'$ and $\phi' \preceq \bullet$. Note that ϕ' and μ' are not fresh, but should instead appear in P, e.g., we might have $P = \{\alpha \preceq \phi', \phi' \preceq \bullet\}$. If we instead assumed $\alpha \preceq \phi$, or removed the condition entirely from P-Qualifier, then P would not need to restrict α at all. We could later instantiate α with a linear type, say File, and use this term to unsoundly copy file handles.

Readers may worry that the P-Qualifier rule is as general as it could be, because it always requires $P \vdash \pi$. For example, consider **let** $f = V$ **in** M where $f : \sigma$ does not appear freely in M. We collect the constraint $\sigma \preceq \bullet$. Constraints of V that are captured in σ do not necessarily need to be satisfied, because f is not used. However, we believe that binding unsatisfiable values has little benefits and can hide potential bugs in practice.

Note that these entailment rules are intentionally made as simple as possible. For example, we do not include any transitivity rules. The entailment rules also do not check potentially conflicted predicates in predicate sets since the rule P-Subsume allows collecting any predicates. We say that predicate set P is satisfiable if there exists a substitution θ such that $\cdot \vdash \theta P$, and define the solutions of it as $[\![P]\!]_{sat} = \{\theta \mid \cdot \vdash \theta P\}$. Transitivity of $\preceq$ is admissible when considering the solutions of predicates, e.g., $[\![\phi_1 \preceq \phi_2, \phi_2 \preceq \bullet]\!]_{sat} = [\![\phi_1 \preceq \phi_2, \phi_2 \preceq \bullet, \phi_1 \preceq \bullet]\!]_{sat} = \{[\bullet/\phi_1, \bullet/\phi_2]\}$. In Section 5.6, we will give an algorithm to check the satisfiability of constraint sets.

5.4 Type Inference

Figure 11 shows representative type inference rules for $\mathrm{Q}^{\circ}_{\mathrm{eff}}$; the remainder are given in full in Appendix B.3. Our type inference algorithm is based on Algorithm W [Damas and Milner 1982] extended for qualified types [Jones 1994]. In $\Gamma \vdash V : A \dashv \theta, P, \Sigma$, the input includes the current context Γ and value V, and the output includes the inferred type A, substitution θ, predicate set P, and variable set Σ of used term variables. Note that the predicates P are an output of inference, not an input; rather than checking entailment, as the syntax-directed type rules do, we will emit a constraint set sufficient to guarantee typing. In the next section, we discuss our algorithm to guarantee that inferred constraint sets are not unsatisfiable. As usual, the substitution θ has been already applied to A and P.

Rule Q-Let$^{\mathrm{W}}$ demonstrates the treatment of linearity. We write $\Gamma|_\Sigma$ for the type context generated by restricting Γ to variables in Σ. We begin by inferring types for V and M. Variable sets Σ_1 and Σ_2 capture those variables used in each; any variable in $\Sigma_1 \cup \Sigma_2$ must be unlimited. We also account for the possibility that the variable x may not be used in M—that is to say, that it may appear in Σ_2^{c}, the complement of the used variables Σ_2. We generate the corresponding unlimitedness constraints using the auxiliary function factorise, discussed next. Rule Q-Do$^{\mathrm{W}}$ emits the constraint that the singleton effect row be included in the output row. Rule Q-Seq$^{\mathrm{W}}$ combines these techniques.

We prove soundness and completeness of type inference with respect to the syntax-directed type system. We write $\theta|_\Gamma$ for the substitution generated by restricting the domain of θ to the free variables in Γ and $(\theta = \theta')|_\Gamma$ for $\theta|_\Gamma = \theta'|_\Gamma$.

Theorem 5.1 (Soundness). *If $\Gamma \vdash V : A \dashv \theta, P, \Sigma$, then $P \mid \theta\Gamma|_\Sigma \vdash V : A$. The same applies to computation and handler typing.*

$$\boxed{\Gamma \vdash V : A \dashv \theta, P, \Sigma} \quad \boxed{\Gamma \vdash M : C \dashv \theta, P, \Sigma} \quad \boxed{\Gamma \vdash H : C \rightrightarrows D \dashv \theta, P, \Sigma}$$

$$\text{Q-Let}^{\text{W}}\ \frac{\Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \qquad \sigma = \mathsf{gen}(\theta_1\Gamma, P_1 \Rightarrow A) \qquad \theta_1\Gamma, x : \sigma \vdash M : C \dashv \theta_2, P_2, \Sigma_2 \qquad Q = \mathsf{un}(\theta_2\theta_1\Gamma|_{\Sigma_1 \cap \Sigma_2}) \cup \mathsf{un}(\theta_2(x : \sigma)|_{\Sigma_2^c})}{\Gamma \vdash \textbf{let}\ x = V\ \textbf{in}\ M : C \dashv \theta_2\theta_1, P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \backslash x)}$$

$$\text{Q-Do}^{\text{W}}\ \frac{\Gamma \vdash V : A \dashv \theta_1, P, \Sigma \qquad A \sim A_\ell : \theta_2 \qquad \mu, \phi\ \textit{fresh} \qquad Q = \mathsf{sub}((\ell : A_\ell \twoheadrightarrow^{\phi} B_\ell), \mu)}{\Gamma \vdash \textbf{do}\ \ell\ V : B_\ell\,!\,\{\mu\} \dashv \theta_2\theta_1, \theta_2 P \cup Q, \Sigma}$$

$$\text{Q-Seq}^{\text{W}}\ \frac{\Gamma \vdash M : A\,!\,\{R_1\} \dashv \theta_1, P_1, \Sigma_1 \qquad \theta_1\Gamma, x : A \vdash N : B\,!\,\{R_2\} \dashv \theta_2, P_2, \Sigma_2 \qquad \mu\ \textit{fresh} \qquad Q = \mathsf{un}(\theta_2\theta_1\Gamma|_{\Sigma_1 \cap \Sigma_2}) \cup \mathsf{un}(\theta_2(x : A)|_{\Sigma_2^c}) \cup \mathsf{leq}(\theta_2\theta_1\Gamma|_{\Sigma_2}, \theta_2 R_1) \cup \mathsf{sub}(\theta_2 R_1, \mu) \cup \mathsf{sub}(R_2, \mu)}{\Gamma \vdash \textbf{let}\ x \leftarrow M\ \textbf{in}\ N : B\,!\,\mu \dashv \theta_2\theta_1, \theta_2 P_1 \cup P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \backslash x)}$$

$$\mathsf{leq}(\Gamma, \tau) = \mathsf{factorise}(\Gamma \preceq \tau) \qquad \mathsf{un}(\Gamma) = \mathsf{leq}(\Gamma, \bullet) \qquad \mathsf{sub}(R_1, R_2) = \mathsf{factorise}(R_1 \leqslant R_2)$$

Fig. 11. Selected Type Inference Rules for $\mathrm{Q}^{\circ}_{\mathrm{eff}}$

Theorem 5.2 (Completeness). *If $P \mid \theta\Gamma \vdash V : A$, then $\Gamma \vdash V : A' \dashv \theta', Q, \Sigma$ and there exists θ'' such that $A = \theta''A'$, $P \vdash \theta''Q$, and $(\theta = \theta''\theta')|_\Gamma$. The same applies to computation and handler typing.*

The proofs can be found in Appendix C.3 and depend on the correctness of factorise, discussed next. Note that we do not need to incorporate the subtyping relation into the statement of the completeness theorem because we only have subtyping between row types and do not allow implicit subsumption (unlike traditional subtyping systems).

5.5 Factorising Predicates

$\mathsf{factorise} : \mathsf{Pred} \to \mathsf{PSet}$
$\mathsf{factorise}(\tau \preceq \tau) = \emptyset$
$\mathsf{factorise}(\tau \preceq \circ) = \emptyset$
$\mathsf{factorise}(\bullet \preceq \tau) = \emptyset$
$\mathsf{factorise}(A \to^Y C \preceq \tau) = \mathsf{factorise}(Y \preceq \tau)$
$\mathsf{factorise}(\tau \preceq K\,;\mu) =$
  $\mathsf{factorise}(\tau \preceq K) \cup \mathsf{factorise}(\tau \preceq \mu)$
$\mathsf{factorise}(\tau \preceq K) =$
  $\bigcup_{(\ell : A \twoheadrightarrow^Y B) \in K} \mathsf{factorise}(\tau \preceq Y)$
$\mathsf{factorise}(R_1 \leqslant R_2) = \emptyset,\ \text{when } \mathsf{set}(R_1) \subseteq \mathsf{set}(R_2)$
$\mathsf{factorise}(R \perp \mathcal{L}) = \emptyset,\ \text{when } \mathsf{dom}(R) \cap \mathcal{L} = \emptyset$
$\mathsf{factorise}(\pi) = \pi$

$\mathsf{factorise} : (\mathsf{TySch} \preceq \mathsf{Type}) \to \mathsf{PSet}$
$\mathsf{factorise}((\forall\alpha.\sigma) \preceq \tau) =$
  $\mathsf{factorise}([\beta/\alpha]\sigma \preceq \tau)$ for some fresh β
$\mathsf{factorise}((\pi \Rightarrow \sigma) \preceq \tau) =$
  $\mathsf{factorise}(\pi) \cup \mathsf{factorise}(\sigma \preceq \tau)$

$\mathsf{factorise} : (\mathsf{Env} \preceq \mathsf{Type}) \to \mathsf{PSet}$
$\mathsf{factorise}(\Gamma \preceq \tau) = \bigcup_{(x:\sigma) \in \Gamma} \mathsf{factorise}(\sigma \preceq \tau)$

$\mathsf{factorise} : \mathsf{PSet} \to \mathsf{PSet}$
$\mathsf{factorise}(P) = \bigcup_{\pi \in P} \mathsf{factorise}(\pi)$

Fig. 12. Factorisation of Constraints

The factorise function is defined in Figure 12; it factors constraints into simpler predicates following the entailment rules in Figure 10. We use K to represent rows consisting of only operation labels.

The only surprising case is for $(\forall\alpha.\sigma) \preceq \tau$. Rule P-Quantifier requires that we find some instance such that $\sigma[\tau'/\alpha] \preceq \tau$. Rather than search for such an instance, we simply pick a fresh type

variable β. As a result, our type inference algorithm is likely to produce *ambiguous* type schemes, in which quantified type variables appear *only* in predicates. Such type schemes are typically rejected [Jones 1994], as the meaning of ambiguously typed terms is undefined. However, as our linearity predicates do not have any intrinsic semantics, but only constrain the use of terms, we do not believe these constraints lead to semantic ambiguity. One interesting property of factorise is that the linearity predicates in its results are only between value type variables α, row type variables μ, and linearity types Y.

We prove the correctness of factorise with respect to the entailment rules in Figure 10.

Theorem 5.3 (Correctness of factorisation). *If* $\mathsf{factorise}(P) = Q$, *then* $Q \vdash P$ *and* $P \vdash Q$. *If* $\mathsf{factorise}(\Gamma \preceq \tau) = Q$, *then* $Q \vdash \Gamma \preceq \tau$ *and for any* $P \vdash \Gamma \preceq \tau$, *there exists* θ *such that* $P \vdash \theta Q$.

The proof can be found in Appendix C.1.

5.6 Constraint Solving

Finally, we must check that inferred constraint sets are satisfiable; we do not want to conclude that a program is well-typed, but only under the assumption that a linear type is unlimited.

We define a constraint solving algorithm $\mathsf{solve}(P)$ for checking the satisfiability of the predicate set P, inspired by solving algorithms for general subtyping constraints [Pottier 1998, 2001; Pretnar 2014]. The tricky part compared to solving usual subtyping constraints is that we need to carefully deal with the interaction between row subtyping constraints and linearity constraints. For instance, $R_1 \leqslant R_2$ and $\tau \preceq R_2$ actually implies $\tau \preceq R_1$. To resolve the interaction, the algorithm proceeds by first transforming row subtyping constraints to those of the forms $\mu \leqslant R$, so that we can always simply instantiate μ on the left to the empty row $\cdot$ for which $\tau \preceq \cdot$ always holds. Then, the algorithm computes the transitive closure of linearity constraints and rejects $\circ \preceq \bullet$. The full algorithm is given in Appendix B.4. We have the following theorem on the correctness of the constraint solving algorithm, in which we write $[\![P]\!]_{sat}\theta$ for the substitution set $\{\theta'\theta \mid \theta' \in [\![P]\!]_{sat}\}$.

Theorem 5.4 (Correctness of constraint solving). *For any constraint set* P *generated by the type inference of* $\mathrm{Q}^{\circ}_{\mathrm{eff}}$, $\mathsf{solve}(P)$ *always terminates.*

- *If it fails, then* P *is not satisfiable.*
- *If it returns* (θ, Q), *then* P *is satisfiable and* $[\![P]\!]_{sat} = [\![Q]\!]_{sat}\theta$.

The proof can be found in Appendix C.4, whose main idea is to show that every step of the algorithm preserves solutions, and the output predicate set has one solution.

We leave the design of constraint simplification algorithms as practical concerns. Some existing algorithms on simplifying general subtyping constraints are promising [Pottier 1998, 2001].

6 SHALLOW HANDLERS

Up to now we have concentrated on *deep* effect handlers, which wrap the original handler around the body of captured continuations. Given this automatic reuse of the handler, the handler itself cannot capture any linear resources. In contrast, shallow handlers [Hillerström and Lindley 2018; Kammar et al. 2013] do not wrap the original handler around the body of captured continuations, which means shallow handlers *can* capture linear resources and thus influence control-flow linearity. In this section, we discuss the extensions of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ and $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ with shallow handlers and their challenges.

Let us first consider shallow handlers in $\mathrm{F}^{\circ}_{\mathrm{eff}}$. We write $H^{\dagger}$ for a shallow handler. The only difference in the operational semantics is the new E-Op† rule for handling with shallow handlers.

$$\text{E-Op}^{\dagger} \quad \mathbf{handle}\ \mathcal{E}[(\mathbf{do}\ \ell\ V)^E]\ \mathbf{with}\ H^{\dagger} \rightsquigarrow N[V/p, (\lambda^Y y^B.\mathcal{E}[(\mathbf{return}\ y)^E])/r], \quad \text{where } \ell \notin \mathsf{bl}(\mathcal{E}), (\ell\ p\ r \mapsto N) \in H^{\dagger} \text{ and } (\ell : A \to^Y B) \in E$$

Unlike in E-Op, the body of the continuation is not handled by $H^\dagger$. Whereas deep handlers perform a fold over a computation trees shallow handlers perform a case-split. As such, we know that exactly one operation clause or the return clause will be invoked, and providing all allowed operations are linear each clause may capture the same linear resources. The typing rule is as follows.

$$\textsc{T-ShallowHandler}\quad \frac{\begin{array}{c} H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \\ C = A\,!\,\{(\ell_i : A_i \twoheadrightarrow^{Y_i} B_i)_i; R\} \qquad D = B\,!\,\{(\ell_i : P)_i; R\} \\ \Delta \vdash \Gamma : Y \qquad \Delta \vdash R : Y \qquad \Delta; \Gamma, x : A \vdash M : D \\ [\Delta; \Gamma, p_i : A_i, r_i : B_i \to^{Y_i} C \vdash N_i : D]_i \end{array}}{\Delta; \Gamma \vdash H^\dagger : C \rightrightarrows D}$$

Instead of requiring value linearity of Γ to be unlimited as in the deep handler rule T-Handler, we require the value linearity of Γ to coincide with the control-flow linearity of R, the effect row of the unhandled operations. This is because the shallow handler may be captured as part of the continuations of these unhandled operations in outer handlers. Concretely, when $Y = \circ$, the shallow handler may use linear variables from the context, and unhandled operations are control-flow linear; when $Y = \bullet$, the shallow handler cannot use any linear variables from the context, and we have no restriction on the control-flow linearity of unhandled operations.

We can also easily extend $\mathrm{Q}^\circ_{\mathrm{eff}}$ with shallow handlers.

$$\textsc{Q-ShallowHandler}\quad \frac{\begin{array}{c} H = \{\mathbf{return}\ x \mapsto M\} \uplus \{\ell_i\ p_i\ r_i \mapsto N_i\}_i \\ C = A\,!\,\{(\ell_i : A_i \twoheadrightarrow^{Y_i} B_i)_i; R_1\} \qquad D = B\,!\,\{R_2\} \\ P \mid \Gamma, x : A \vdash M : D \qquad [P \mid \Gamma, p_i : A_i, r_i : B_i \to^{Y_i} C \vdash N_i : D]_i \\ P \vdash \Gamma \preceq R_1 \qquad P \vdash R_1 \leqslant R_2 \qquad P \vdash R_1 \perp \{\ell_i\}_i \end{array}}{P \mid \Gamma \vdash H : C \rightrightarrows D}$$

In place of $P \vdash \Gamma \preceq \bullet$ in Q-Handler, we have $P \vdash \Gamma \preceq R_1$, which restricts the value linearity of the type context to be less than the control-flow linearity of unhandled operations in R_1.

Shallow handlers are typically used together with recursive functions to implement more general recursive behaviours than the structural recursion of deep handlers. It is straightforward to extend $\mathrm{F}^\circ_{\mathrm{eff}}$ and $\mathrm{Q}^\circ_{\mathrm{eff}}$ with recursive functions [Hillerström et al. 2020a; Mazurak et al. 2010]. Obviously recursive functions are themselves unlimited so cannot capture linear resources, but that does not preclude explicitly threading a linear resource through a recursive function that installs a shallow handler. We use the syntax $\mathbf{rec}\ f\ x.M$ to define a recursive function f with parameter x and function body M. The typing rules and semantics rule for it in $\mathrm{F}^\circ_{\mathrm{eff}}$ and $\mathrm{Q}^\circ_{\mathrm{eff}}$ are as follows.

$$\textsc{T-Rec}\quad \frac{\Delta; \Gamma, f : A \to^\bullet C, x : A \vdash M : C \qquad \Delta \vdash \Gamma : \bullet}{\Delta; \Gamma \vdash \mathbf{rec}\ f^{A \to^\bullet C}\ x.M : A \to^\bullet C} \qquad \textsc{Q-Rec}\quad \frac{\Delta; \Gamma, f : A \to^\bullet C, x : A \vdash M : C \qquad P \vdash \Gamma \preceq \bullet}{P \mid \Gamma \vdash \mathbf{rec}\ f\ x.M : A \to^\bullet C}$$

$$\textsc{E-Rec}\quad (\mathbf{rec}\ f\ x.M)\ V \rightsquigarrow M[(\mathbf{rec}\ f\ x.M)/f, V/x]$$

As an example, we can write the following recursive function $\mathsf{withFile}\ f$ which takes a file handle f and interprets all Print operations in M as writing to file f.

$$\begin{array}{l} \mathsf{withFile}\ f = \mathbf{rec}\ \mathsf{withFile}\ f.\mathbf{handle}\ M\ \mathbf{with} \\ \qquad \{\mathbf{return}\ x \mapsto \mathsf{Close}\ f\ ; x \\ \qquad\ \mathsf{Print}\ s\ r \mapsto \mathbf{let}\ f' \leftarrow \mathsf{write}\ (s, f)\ \mathbf{in}\ \mathsf{withFile}\ f'\ r\} \end{array}$$

Note that this example can also be implemented with a deep handler by requiring the handler to return a function which takes the file handle as a parameter. Shallow handlers provide us with a more direct programming style.

Although our two new typing rules are straightforward and entirely backward compatible with the current systems, shallow handlers can actually introduce more challenges to track control-flow linearity. This is essentially because shallow handlers are more flexible than deep handlers and do not handle all invocations of the same operation uniformly. With only deep handlers, it is natural for all invocations of an operation to have the same control-flow linearity as they are handled by the same handler. However, with shallow handlers, different invocations of the same operation can be handled by different handlers, resulting in different control-flow linearity. For example, consider the following program hesitantClose which makes choices before and after closing the file f.

$$\mathsf{hesitantClose} = \lambda f.\mathbf{do}\ \mathsf{Choose}\,();\mathsf{close}\,f;\mathbf{do}\ \mathsf{Choose}\,()$$

The continuation of the first Choose contains the linear file handle f, whereas the second one does not. Technically, the handler for the second Choose can resume any number of times. However, neither the effect system of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ nor that of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ is able to ascribe a different control-flow linearity to the two invocations of Choose, which means we must handle both invocations linearly. One potential solution is to track the order and duplication of effects in the effect system. However, this kind of information is known to be too cumbersome for effect systems. A more lightweight solution is to exploit named handlers [Biernacki et al. 2020; Xie et al. 2022] to assign Choose operations in different positions to different shallow handlers. We leave the design of an ergonomic and expressive effect system for tracking control-flow linearity of shallow handlers to future work.

7 RELATED WORK

Linear Resources and Control Effects. Exception handlers with finally clauses are a common way of managing linear resources. Exception handlers provide a form of unwind protection, which enables the programmer to supply the logic to release acquired resources in the finally clause, which gets executed irrespective of whether a fault occurs. Similarly, the `defer` statement in Go [Donovan and Kernighan 2015] defers the execution of its operand until the defining function returns either successfully or via a fault. Thus the programmer can conveniently acquire a particular resource and include the deferred logic for releasing it on the next line of code. Another variation is automatic resource block management as in the C++ RAII idiom [Combette and Munch-Maccagnoni 2018] and Java's `try-with-resource` [Gosling et al. 2023], both of which offer a means for automatically acquiring and releasing resources in the static scope. In Scheme the fundamental resource protection mechanism is the procedure `dynamic-wind` [Friedman and Haynes 1985]. It is a generalisation of unwind protection intended to be used in the presence of first-class control, where control may enter and leave the same computation multiple times. It takes three functional arguments: the first is the resource acquisition procedure, which gets applied when control enters `dynamic-wind`; the second is the main computation, which may use the acquired resources; and the third is the resource release procedure, which is applied when control is about to leave `dynamic-wind`.

Brachthäuser and Leijen [2023] present a constraint system based on qualified types for programming with multi-shot effect handlers and linear resources in Koka. They use these constraints to mark some effects as linear. However, they do not include a linear type system and instead rely on pre-declaring the linearity of operations (i.e., no inference for control-flow linearity) and a syntactic check to ensure that resumptions are not invoked more than once. Compared to the qualified effect system of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$, their system does not support effect subtyping and abstraction over linearity.

Structural Types and Control Effects. Tov and Pucella [2011b] propose a calculus $\lambda^{\text{URAL}}(\mathscr{C})$ which extends the substructural λ-calculus λ^{URAL} [Ahmed et al. 2005] with abstract control effects $\mathscr{C}$ given by a set of effects, a pure effect, and an effect-sequencing operator. They show how to instantiate $\lambda^{\text{URAL}}(\mathscr{C})$ with concrete control effects including exceptions and shift/reset [Danvy and Filinski 1990] separately. Similar to $\text{F}^{\circ}_{\text{eff}}$ and $\text{Q}^{\circ}_{\text{eff}}$, the $\lambda^{\text{URAL}}(\mathscr{C})$ calculus also uses type-and-effect system to check that control effects do not violate the substructural usage guarantees for values. It includes a judgement on effect types to determine whether control effects may discard or duplicate their continuations, which roughly corresponds to our notion of control-flow linearity. The main difference between our work and $\lambda^{\text{URAL}}(\mathscr{C})$ is that we consider the tracking of control-flow linearity in the presence of algebraic effects and effect handlers, which are more involved than exceptions and shift/reset both statically and dynamically. While it is theoretically possible to instantiate $\lambda^{\text{URAL}}(\mathscr{C})$ to effect handlers, this task is itself highly non-trivial due to the richer effect systems of effect handlers. Conversely, we can also easily encode exceptions and shift/reset as user-defined effects in $\text{F}^{\circ}_{\text{eff}}$ and $\text{Q}^{\circ}_{\text{eff}}$ using effect handlers [Forster et al. 2019; Piróg et al. 2019].

Linear Type Systems. Type inference with linear types is a well-studied area. Mazurak et al. [2010] propose using kinds to track linearity, using subkinding to enable polymorphism over linearities. Tov and Pucella [2011a] develop an expanded approach to tracking structural restrictions in kinds; among other differences they introduce subtyping for function types and require fewer explicit linearity annotations than Mazurak et al.. Gan et al. [2014] use qualified types to characterise types that admit structural rules in a substructural type system: for example, in a linear type system, unlimited types are exactly types τ that support operations $\text{dup} : \tau \to (\tau, \tau)$ and $\text{drop} : \tau \to ()$. Morris [2016] extends the approach of Tov and Pucella to generalise the treatment of function types, introducing the linearity ordering constraint $\tau \preceq \upsilon$; he also generalises their description of unlimited types to type schemes, but does so unsoundly. In contrast, the current work does not interpret unlimited types via operations like dup and drop; we also avoid Morris's unsoundness in the treatment of type schemes. An alternative approach tracks linearity exclusively in function types, rather than in kinds. This approach is developed by Ghica and Smith [2014], McBride [2016], and Atkey [2018], and has been implemented in Idris [Brady 2021] and an extension to the GHC Haskell compiler [Bernardy et al. 2018].

Row-based Effect Types. Row types and row polymorphism are a popular way of implementing effect systems in programming languages. Links [Hillerström and Lindley 2016] adopts Rémy style row polymorphism [Rémy 1994], where the row types are able to represent the absence of labels and each label is restricted to appear at most once. Koka [Leijen 2017] and Frank [Lindley et al. 2017] use row polymorphism based on scoped labels [Leijen 2005] which allows duplicated labels. We believe the idea of tracking control-flow linearity in $\text{F}^{\circ}_{\text{eff}}$ should work well with all kinds of different row-based effect systems.

Subtyping-based Effect Types. Some versions of Eff [Bauer and Pretnar 2014; Pretnar 2014] use an effect system based on subtyping. Karachalias et al. [2020] describe an explicit target calculus ExEff with a subtyping-based effect system and a type inference algorithm that elaborates Eff source code into it. Eff uses a row-like representation of effect types and defines a subtyping relation for effect types similar to the that of $\text{Q}^{\circ}_{\text{eff}}$. One difference is that Eff incorporates full subtyping relations between all types and implicit subsumption, whereas we only introduce subtyping between row types and allow explicit subsumption in necessary positions (like Q-Seq and Q-Handle). In this respect our qualified effect system is more lightweight. Algebraic subtyping [Dolan 2016; Dolan and Mycroft 2017] combines subtyping and parametric polymorphism with elegant principal types.

It would be interesting to explore the possibility of combining linear types and effect types based on algebraic subtyping with control-flow linearity.

One-shot control operators. One-shot continuations were first introduced by Friedman and Haynes [1985] in the form of a linear variant of call/cc. Similarly, Filinski [1992] considers a one-shot variant of the C operator [Felleisen et al. 1987].

One-shot Effect Handlers. OCaml 5 [Sivaramakrishnan et al. 2021], the C++-effects library [Ghica et al. 2022], and the typed continuations proposal for adding effect handlers to WebAssembly [Hillerström et al. 2022; Phipps-Costin et al. 2023] all implement dynamically-checked one-shot effect handlers. Continuations captured by such effect handlers can be thought of as linear resources themselves, and thus play nicely with other linear resources. Any attempt to invoke a continuation more than once throws a runtime error. In contrast, our type systems can be used to statically ensure that handlers are one-shot. In fact, its considerably easier to build a system that ensures that *all* handlers are uniformly one-shot than a system like ours that supports both one-shot and multi-shot handlers, as in the former case there is no need to track the use of linear resources specially. Another advantage of one-shot continuations is that they admit efficient implementations which are compatible with linear resources, as a one-shot continuation need not copy its underlying stack [Bruggeman et al. 1996]. Hillerström et al. [2023] present a substructural type system for a calculus with effect handlers based on dual intuitionistic linear logic [Barber 1996] which restricts all effect handlers to be one-shot (actually one- or zero-shot). They use it to show an asymptotic performance gap between one-shot and multi-shot effect handlers, but are not concerned with linear resources other than continuations.

Multi-shot Effect Handlers. Eff [Bauer and Pretnar 2015], Effekt [Brachthäuser et al. 2020], Koka [Leijen 2017], and Helium [Biernacki et al. 2019] are research programming languages with multi-shot handlers. In contrast to one-shot handlers, multi-shot handlers can invoke the captured continuations an arbitrary number of times. This enables a range of interesting applications. For instance, asymptotic efficient backtracking search [Hillerström et al. 2020b], nondeterminism [Kammar et al. 2013], and UNIX fork-style concurrency [Hillerström 2022] can all be given a direct semantics in terms of multi-shot handlers. However, one obstacle is that the aforementioned languages cannot statically optimise uses of one-shot continuations, as they must conservatively expect the ambient context to have nonlinear control flow, thus requiring them to copy the continuation a priori [Hillerström 2016; Hillerström et al. 2016]. Our type systems can enable static optimisation of one-shot continuations through static identification of linear and nonlinear contexts.

8 CONCLUSION AND FUTURE WORK

We have explored the interplay between effect handlers and linear types. We have demonstrated that in order to soundly combine potentially non-linear effect handlers with linear types, it is necessary to add a mechanism for tracking control-flow linearity too. We incorporated control-flow linearity into two quite different core languages as well as realising control-flow linearity in Links.

Directions for future work include: implementing a programming language based on $\mathrm{Q}^{\circ}_{\mathrm{eff}}$; developing more precise type systems for combining control-flow linearity with shallow handlers; combining control-flow linearity with other forms of effect type systems, such as those that support generative effects, duplicate effects, capabilities, and modal effect types; adapting the constraints of $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ to algebraic subtyping [Dolan and Mycroft 2017]; and adapting control-flow linearity for uniqueness types and for quantitive type theory [Atkey 2018; McBride 2016].

DATA AVAILABILITY STATEMENT

The implementation of F°_{eff} in Links is available on Zenodo [Tang et al. 2023].

ACKNOWLEDGMENTS

This work was supported by the UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (reference number MR/T043830/1).

REFERENCES

Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A step-indexed model of substructural state. In *ICFP*. ACM, 78–91. https://doi.org/10.1145/1086365.1086376

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *LICS*. ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report ECS-LFCS-96-347. Laboratory for Foundations of Computer Science, The University of Edinburgh, UK.

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Log. Methods Comput. Sci.* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/J.JLAMP.2014.02.001

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 3, POPL (2019), 6:1–6:28. https://doi.org/10.1145/3290319

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Jonathan Immanuel Brachthäuser and Daan Leijen. 2023. *Qualified Effect Types – Taming Control-Flow through Linear Effect Handlers.* Technical Report MSR-TR-2023-42. Microsoft. https://www.microsoft.com/en-us/research/publication/qualified-effect-types/

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 126:1–126:30. https://doi.org/10.1145/3428194

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *ECOOP (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *PLDI*. ACM, 99–107. https://doi.org/10.1145/231379.231395

Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A Resource Modality for RAII. In *LOLA 2018: Workshop on Syntax and Semantics of Low-Level Languages*. 1–4.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO (Lecture Notes in Computer Science, Vol. 4709)*. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *POPL*. ACM Press, 207–212. https://doi.org/10.1145/582153.582176

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160. https://doi.org/10.1145/91556.91622

Stephen Dolan. 2016. *Algebraic Subtyping*. Ph. D. Dissertation. Computer Laboratory, University of Cambridge, United Kingdom.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *POPL*. ACM, 60–72. https://doi.org/10.1145/3009837.3009882

Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1987. A Syntactic Theory of Sequential Control. *Theor. Comput. Sci.* 52 (1987), 205–237. https://doi.org/10.1016/0304-3975(87)90109-5

Andrzej Filinski. 1992. Linear Continuations. In *POPL*. ACM Press, 27–38. https://doi.org/10.1145/143165.143174

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15. https://doi.org/10.1017/S0956796819000121

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *POPL*. ACM Press, 245–254. https://doi.org/10.1145/318593.318654

Daniel P. Friedman, Christopher T Haynes, and Eugene Kohlbecker. 1984. Programming with Continuations. In *Program Transformation and Programming Environments*, Peter Pepper (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–274. https://doi.org/10.1007/978-3-642-46490-4_23

Edward Gan, Jesse A. Tov, and Greg Morrisett. 2014. Type Classes for Lightweight Substructural Types. In *LINEARITY (EPTCS, Vol. 176)*. 34–48. https://doi.org/10.4204/EPTCS.176.4

Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-level effect handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1639–1667. https://doi.org/10.1145/3563445

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *ESOP (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. The Java Language Specification: Java SE 20 Edition. https://docs.oracle.com/javase/specs/jls/se20/html/index.html. [Accessed 2023-07-11].

Daniel Hillerström. 2022. *Foundations for Programming and Implementing Effect Handlers*. Ph. D. Dissertation. School of Informatics, The University of Edinburgh, UK.

Daniel Hillerström, Daan Leijen, Sam Lindley, Matija Pretnar, Andreas Rossberg, and KC Sivamarakrishnan. 2022. WebAssembly Typed Continuations Proposal. https://github.com/wasmfx/specfx/blob/main/proposals/continuations/Explainer.md [Accessed 2023-11-14].

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (Lecture Notes in Computer Science, Vol. 11275)*. Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020a. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

Daniel Hillerström, Sam Lindley, and John Longley. 2020b. Effects for Efficiency: Asymptotic Speedup with First-Class Control. *Proc. ACM Program. Lang.* 4, ICFP (2020), 100:1–100:29. https://doi.org/10.1145/3408982

Daniel Hillerström, Sam Lindley, and John Longley. 2023. Asymptotic Speedup with Effect Handlers. Draft.

Daniel Hillerström. 2016. *Compilation of Effect Handlers and their Applications in Concurrency*. Master by Research thesis. School of Informatics, The University of Edinburgh, UK.

Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. ML Workshop.

Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *ICFP*. ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. 2020. Explicit effect subtyping. *J. Funct. Program.* 30 (2020), e15. https://doi.org/10.1017/S0956796820000131

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded Probabilistic Programming. In *DSL (Lecture Notes in Computer Science, Vol. 5658)*. Springer, 360–384. https://doi.org/10.1007/978-3-642-03034-5_17

Daan Leijen. 2005. Extensible records with scoped labels. In *Trends in Functional Programming (Trends in Functional Programming, Vol. 6)*. Intellect, 179–194.

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *ICFP*. ACM, 283–294. https://doi.org/10.1145/1411204.1411245

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499. https://doi.org/10.1145/3009837.3009872

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. ACM, 91–102. https://doi.org/10.1145/2103786.2103798

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514. https://doi.org/10.1145/3009837.3009897

Sam Lindley and J Garrett Morris. 2017. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers* (2017), 265–286.

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282. https://doi.org/10.1145/357162.357169

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F° *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/1708016.1708027

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride,

Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

J. Garrett Morris. 2016. The best of both worlds: linear functional programming without compromise. In *ICFP*. ACM, 448–461. https://doi.org/10.1145/2951913.2951925

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 460–485. https://doi.org/10.1145/3622814

Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *FSCD (LIPIcs, Vol. 131)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16. https://doi.org/10.4230/LIPICS.FSCD.2019.30

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013).

François Pottier. 1998. *Type inference in the presence of subtyping: from theory to practice.* Ph. D. Dissertation. INRIA.

François Pottier. 2001. Simplifying Subtyping Constraints: A Theory. *Inf. Comput.* 170, 2 (2001), 153–183. https://doi.org/10.1006/inco.2001.2963

Ron Pressler. 2018. Project Loom: Fibers and Continuations for the Java Virtual Machine. https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html. Accessed 2023-04-14.

Matija Pretnar. 2014. Inferring Algebraic Effects. *Log. Methods Comput. Sci.* 10, 3 (2014). https://doi.org/10.2168/LMCS-10(3:21)2014

Didier Rémy. 1994. Theoretical Aspects of Object-oriented Programming. MIT Press, Cambridge, MA, USA, Chapter Type Inference for Records in Natural Extension of ML, 67–95.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221. https://doi.org/10.1145/3453483.3454039

Wenhao Tang, Daniel Hillerström, Sam Lindley, and Garrett Morris. 2023. *POPL24 Artifact for Soundly Handling Linearity.* https://doi.org/10.5281/zenodo.10120126

Jesse A. Tov and Riccardo Pucella. 2011a. Practical affine types. In *POPL*. ACM, 447–458. https://doi.org/10.1145/1926385.1926436

Jesse A. Tov and Riccardo Pucella. 2011b. A theory of substructural types and control. In *OOPSLA*. ACM, 625–642. https://doi.org/10.1145/2048066.2048115

David Walker. 2005. Substructural type systems. *Advanced topics in types and programming languages* (2005), 3–44.

Ningning Xie, Youyou Cong, Kazuki Ikemori, and Daan Leijen. 2022. First-Class Names for Effect Handlers. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 30–59. https://doi.org/10.1145/3563289

Received 2023-07-11; accepted 2023-11-07