

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.

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 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`.

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.

```

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 McKinn 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.¹ 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 ($\text{File} : \text{Type}^{\circ}$) and integers are unlimited resources ($\text{Int} : \text{Type}^{\bullet}$).

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.

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

¹Linear types differ from uniqueness types, which instead track the number of references to a value.

The outer unlimited function (\rightarrow^\bullet) yields a linear function (\rightarrow°) 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 $\vdash \text{Type}^\bullet \leq \text{Type}^\circ$ in F_{eff}° . For instance, consider the polymorphic identity function.

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

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 $\vdash \text{Int} : \text{Type}^\circ$ and $\vdash \text{File} : \text{Type}^\circ$, and if R is an effect row type:

$$\begin{aligned} \text{id } R \text{ File} &: \text{File} \rightarrow^\bullet \text{File} ! \{ R \} \\ \text{id } R \text{ Int} &: \text{Int} \rightarrow^\bullet \text{Int} ! \{ R \} \end{aligned}$$

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 F_{eff}° . Consider the following function `dubiousWritex`, 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.

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

The `do Choose ()` expression invokes operation `Choose` with a unit argument. F_{eff}° adapts an effect system based on Rémy-style row polymorphism [Hillerström and Lindley 2016; Lindley and Cheney 2012]. Effect types in F_{eff}° are rows containing operation labels with their signatures and ended with potential row variables. The effect type $\{ \text{Choose} : () \rightarrow \text{Bool} \}$ denotes that `dubiousWritex` may invoke the operation `Choose`, which takes a unit and returns a boolean value as indicated by its signature $() \rightarrow \text{Bool}$. The problem arises when we handle `Choose` using multi-shot continuations.

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

The file "C.txt" is opened and the file handle is bound to f before `dubiousWritex 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 $A \rightarrow^Y B$ is annotated with a linearity Y to denote its control flow linearity. The `dubiousWrite✓` function can now be rewritten to correctly track control flow linearity as follows.

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

The type of `dubiousWrite✓` now tracks that the operation $\text{Choose} : () \rightarrow^{\circ} \text{Bool}$ invoked by it is control flow linear. We also annotate let-bindings with linearity information. In $\text{let}^Y x \leftarrow M \text{ in } N$, the term N has control flow linearity Y , and in particular the \circ annotations on the let-bindings in `dubiousWrite✓` permit the use of the linear file handle throughout.

The linear type system of F_{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 `dubiousWrite✓` with the same multi-shot handler.

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

This is ill-typed: as Choose is control flow linear, the resumption r is given a linear function type and cannot be invoked twice.

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 $\vdash \text{Row}^{\circ} \leq \text{Row}^{\bullet}$, 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.

$$\begin{aligned} \text{tossCoin} &: \forall \mu^{\text{Row}^{\bullet}}. (() \rightarrow^{\bullet} \text{Bool} ! \{ \mu \}) \rightarrow^{\bullet} \text{String} ! \{ \mu \} \\ \text{tossCoin} &= \Lambda \mu^{\text{Row}^{\bullet}}. \lambda^{\bullet} g. \text{let}^{\bullet} b \leftarrow g () \text{ in if } b \text{ then "heads" else "tails"} \end{aligned}$$

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

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

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 $\text{Row}^\circ \leq \text{Row}^\bullet$. This is because when handling, we directly look at the control flow linearity on operation signatures instead of their kinds, where $\text{no} \rightarrow^\circ$ can be upcast to \rightarrow^\bullet . This can be seen more clearly from the typing rules in Section 3.2. We formally state the soundness of F_{eff}° in Sections 3.4 and 3.5.

2.3 Qualified Linear Types

As we have seen from the examples so far, F_{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 `Print : String \rightarrow ()` before returning.

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

Depending on different choices of 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.

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

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 Q_{eff}° with qualified linear types, `verboseId` can be written and ascribed a principal type as follows.

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

The linearity variables ϕ and ϕ' quantify over \circ and \bullet . We do not use kinds to represent linearity of types variables; instead, all linearity information is represented using predicates of the form $\tau \leq \tau'$, 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 $\alpha \leq \phi$, 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 Q_{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 F_{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 F_{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 → ()`.

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

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 F_{eff}° , omitting the full operation signatures for simplicity, we could write $R = R_1 = R_2 = \{\text{Get} : \circ, \text{Print} : \bullet\}$ 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 F_{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 $R_1 = \{\text{Get} : \circ\}$, $R_2 = \{\text{Print} : \bullet\}$, and $R = \{\text{Get} : \circ, \text{Print} : \bullet\}$, 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.

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

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 F_{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, Q_{eff}° also supports qualified effect types inspired by ROSE [Morris and McKinna 2019]. In Q_{eff}° , the function `sandwichClose` can be given the following type. Note that here we still choose to fix functions to be unlimited for readability.

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

The constraints $\mu_1 \leq \mu$ and $\mu_2 \leq \mu$ express that rows μ_1 and μ_2 are contained in μ , and the constraint $\text{File} \leq \mu_1$ 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 Q_{eff}° infers these row subtyping constraints without the need for any annotation. The qualified linear types and qualified effect types of Q_{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 F_{eff}° , a System F-style fine-grain call-by-value calculus with linear types and effect handlers. F_{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 F_{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 F_{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 F_{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].

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

Fig. 1. Syntax of types, kinds, contexts, values and computations of F_{eff}° .

F_{eff}° explicitly distinguishes between value types and computation types as well as their terms. Value types include type variables α , function types $A \rightarrow^Y C$, 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 $A \rightarrow^Y B$, or polymorphic θ in its presence. An operation signature $A \rightarrow^Y B$ 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 VW and type application VT are standard. A computation $(\text{return } V)^E$ returns the value V . An operation invocation $(\text{do } \ell V)^E$ invokes the operation ℓ with parameter V . They are both annotated with their effect types for deterministic typing. Sequencing $\text{let}^Y x \leftarrow M \text{ in } N$ evaluates M and binds its result to x in N . The linearity Y basically indicates the control flow linearity of N . Handling $\text{handle } M \text{ with } H$ handles computation M with handler H . Handlers are given by a return clause $\text{return } x \mapsto M$, 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 Presence^Y and row types $\text{Row}_{\mathcal{L}}^Y$ 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., $\vdash \text{File} : \circ$ and $\vdash \text{Int} : \bullet$. 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 $\vdash \text{Type}^\bullet \leq \text{Type}^\circ$ 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 from

$$\begin{array}{c}
\boxed{\vdash Y \leq Y'} \quad \boxed{\vdash K \leq K'} \\
\text{S-LIN} \quad \text{S-TYPE} \quad \text{S-PRES} \quad \text{S-ROW} \\
\frac{}{\vdash \bullet \leq \circ} \quad \frac{\vdash Y \leq Y'}{\vdash \text{Type}^Y \leq \text{Type}^{Y'}} \quad \frac{\vdash Y' \leq Y}{\vdash \text{Presence}^Y \leq \text{Presence}^{Y'}} \quad \frac{\vdash Y' \leq Y}{\vdash \text{Row}_{\mathcal{L}}^Y \leq \text{Row}_{\mathcal{L}}^{Y'}} \\
\boxed{\Delta \vdash T : K} \\
\text{K-TYVAR} \quad \text{K-FORALL} \quad \text{K-FUN} \quad \text{K-COMP} \\
\frac{}{\Delta, \alpha : K \vdash \alpha : K} \quad \frac{\Delta, \alpha : K \vdash C : \text{Comp}}{\Delta \vdash \forall \alpha^K. C : \text{Type}^Y} \quad \frac{\Delta \vdash A : \text{Type}^{Y'} \quad \Delta \vdash C : \text{Comp}}{\Delta \vdash A \rightarrow^Y C : \text{Type}^Y} \quad \frac{\Delta \vdash A : \text{Type}^Y \quad \Delta \vdash E : \text{Effect}}{\Delta \vdash A!E : \text{Comp}} \\
\text{K-EFFECT} \quad \text{K-PRESENT} \quad \text{K-ABSENT} \quad \text{K-EMPTYROW} \\
\frac{\Delta \vdash R : \text{Row}_\emptyset}{\Delta \vdash \{R\} : \text{Effect}} \quad \frac{}{\Delta \vdash A \rightarrow^Y B : \text{Presence}^Y} \quad \frac{}{\Delta \vdash \text{Abs} : \text{Presence}^Y} \quad \frac{}{\Delta \vdash \cdot : \text{Row}_{\mathcal{L}}^Y} \\
\text{K-EXTENDROW} \quad \text{K-HANDLER} \quad \text{K-UPCAST} \\
\frac{\Delta \vdash P : \text{Presence}^Y \quad \Delta \vdash R : \text{Row}_{\mathcal{L} \uplus \{\ell\}}^Y}{\Delta \vdash \ell : P; R : \text{Row}_{\mathcal{L}}^Y} \quad \frac{\Delta \vdash C : \text{Comp} \quad \Delta \vdash D : \text{Comp}}{\Delta \vdash C \Rightarrow D : \text{Handler}} \quad \frac{\Delta \vdash T : K \quad \vdash K \leq K'}{\Delta \vdash T : K}
\end{array}$$

Fig. 2. Kinding and subkinding rules for F_{eff}° .

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 $\vdash \text{Row}_{\mathcal{L}}^\circ \leq \text{Row}_{\mathcal{L}}^\bullet$ and $\vdash \text{Presence}^\circ \leq \text{Presence}^\bullet$ 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 $\text{Type}^\bullet \leq \text{Type}^\circ$, 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 $\text{Row}^\circ \leq \text{Row}^\bullet$, 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 $\Delta; \Gamma \vdash H : C \Rightarrow D$, state respectively that: value V has type A , computation M has type C and handler H has type $C \Rightarrow D$ in contexts Δ and Γ . As usual, the type contexts and types are well formed under the kind contexts.

$$\begin{array}{c}
 \boxed{\Delta \vdash \Gamma : Y} \\
 \\
 \text{L-EMPTY} \\
 \frac{}{\Delta \vdash \cdot : Y} \\
 \\
 \boxed{\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2} \\
 \\
 \text{C-EMPTY} \\
 \frac{}{\Delta \vdash \cdot = \cdot + \cdot} \\
 \\
 \text{C-LINLEFT} \\
 \frac{\Delta \vdash A : \text{Type}^\circ \quad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + \Gamma_2} \\
 \\
 \text{C-LINRIGHT} \\
 \frac{\Delta \vdash A : \text{Type}^\circ \quad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = \Gamma_1 + (\Gamma_2, x : A)} \\
 \\
 \text{L-EXTEND} \\
 \frac{\Delta \vdash \Gamma : Y \quad \Delta \vdash A : \text{Type}^Y}{\Delta \vdash (\Gamma, x : A) : Y} \\
 \\
 \text{C-UNL} \\
 \frac{\Delta \vdash A : \text{Type}^\bullet \quad \Delta \vdash \Gamma = \Gamma_1 + \Gamma_2}{\Delta \vdash \Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)}
 \end{array}$$

Fig. 3. Linearity of contexts and context splitting.

$$\begin{array}{c}
 \boxed{\Delta; \Gamma \vdash V : A} \quad \boxed{\Delta; \Gamma \vdash M : C} \quad \boxed{\Delta; \Gamma \vdash H : C \Rightarrow D} \\
 \\
 \text{T-VAR} \\
 \frac{\Delta \vdash \Gamma : \bullet}{\Delta; \Gamma, x : A \vdash x : A} \\
 \\
 \text{T-ABS} \\
 \frac{\Delta \vdash \Gamma : Y \quad \Delta \vdash A : \text{Type}^{Y'}}{\Delta; \Gamma, x : A \vdash M : C} \\
 \\
 \text{T-TABS} \\
 \frac{\Delta \vdash \Gamma : Y \quad \alpha \notin \text{ftv}(\Gamma) \quad \Delta, \alpha : K; \Gamma \vdash M : C}{\Delta; \Gamma \vdash \lambda^Y x^K. M : \forall^Y \alpha^K. C} \\
 \\
 \text{T-APP} \\
 \frac{\Delta; \Gamma_1 \vdash V : A \rightarrow^Y C \quad \Delta; \Gamma_2 \vdash W : A}{\Delta; \Gamma_1 + \Gamma_2 \vdash VW : C} \\
 \\
 \text{T-TAPP} \\
 \frac{\Delta; \Gamma \vdash V : \forall^Y \alpha^K. C \quad \Delta \vdash T : K}{\Delta; \Gamma \vdash VT : C[T/\alpha]} \\
 \\
 \text{T-RETURN} \\
 \frac{\Delta; \Gamma \vdash V : A \quad \Delta \vdash E : \text{Effect}}{\Delta; \Gamma \vdash (\mathbf{return} V)^E : A!E} \\
 \\
 \text{T-DO} \\
 \frac{E = \{\ell : A \rightarrow^Y B; R\} \quad \Delta; \Gamma \vdash V : A \quad \Delta \vdash E : \text{Effect}}{\Delta; \Gamma \vdash (\mathbf{do} \ell V)^E : B!E} \\
 \\
 \text{T-SEQ} \\
 \frac{\Delta; \Gamma_1 \vdash M : A! \{R\} \quad \Delta; \Gamma_2, x : A \vdash N : B! \{R\} \quad \Delta \vdash \Gamma_2 : Y \quad \Delta \vdash R : Y}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow M \mathbf{in} N : B! \{R\}} \\
 \\
 \text{T-HANDLER} \\
 \frac{H = \{\mathbf{return} x \mapsto M\} \uplus \{\ell_i p_i r_i \mapsto N_i\}_i \quad C = A! \{(\ell_i : A_i \rightarrow^Y B_i); R\} \quad D = B! \{(\ell_i : P)_i; R\} \quad \Delta \vdash \Gamma : \bullet \quad \Delta; \Gamma, x : A \vdash M : D}{\Delta; \Gamma \vdash H : C \Rightarrow D} \\
 \\
 \text{T-HANDLE} \\
 \frac{\Delta; \Gamma_1 \vdash H : C \Rightarrow D \quad \Delta; \Gamma_2 \vdash M : C}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle} M \mathbf{with} H : D}
 \end{array}$$

 Fig. 4. Typing rules for F_{eff}° .

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 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 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 $\ell_i : A_i \rightarrow^{Y_i} B_i$, 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 $\text{Row}^\circ \leq \text{Row}^\bullet$ 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✓` in Section 2.2 is an example. Note that technically, the third sequencing $\mathbf{let}^\circ f' \leftarrow \mathbf{write}(s, f) \mathbf{in} \mathbf{close} f'$ can be changed to \mathbf{let}^\bullet because no linear variable bound outside is used by the context $\mathbf{let} f' \leftarrow _ \mathbf{in} \mathbf{close} f'$.

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 defining a trivial subtyping relation on effect row types as shown in Figure 5.

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

$$\frac{\Delta \vdash R : K}{\Delta \vdash R \leq R : K} \quad \frac{\Delta \vdash R_1 \leq R_2 : K \quad \Delta \vdash R_2 \leq R_3 : K}{\Delta \vdash R_1 \leq R_3 : K} \quad \frac{\Delta \vdash \mu : K}{\Delta \vdash \cdot \leq \mu : K}$$

$$\frac{\Delta \vdash P : \text{Presence}^Y \quad \Delta \vdash R_1 \leq R_2 : \text{Row}_{\mathcal{L}\psi\{\ell\}}^Y}{\Delta \vdash \ell : \text{Abs}; R_1 \leq \ell : P; R_2 : \text{Row}_{\mathcal{L}}^Y} \quad \frac{\Delta \vdash P : \text{Presence}^Y \quad \Delta \vdash R_1 \leq R_2 : \text{Row}_{\mathcal{L}\psi\{\ell\}}^Y}{\Delta \vdash \ell : P; R_1 \leq \ell : P; R_2 : \text{Row}_{\mathcal{L}}^Y}$$

Fig. 5. Trivial subtyping for effect row types.

The subtyping relation $\Delta \vdash R \leq R' : K$ 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' . The slightly more precise typing rule for

sequencing is shown as follows.

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

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

$$\begin{array}{l} \text{E-APP} \quad (\lambda^Y x^A. M) V \rightsquigarrow M[V/x] \\ \text{E-TAPP} \quad (\Lambda^Y \alpha^K. M) T \rightsquigarrow M[T/\alpha] \\ \text{E-SEQ} \quad \mathbf{let}^Y x \leftarrow (\mathbf{return} V)^E \mathbf{in} N \rightsquigarrow N[V/x] \\ \text{E-RET} \quad \mathbf{handle} (\mathbf{return} V)^E \mathbf{with} H \rightsquigarrow N[V/x], \quad \text{where } (\mathbf{return} x \mapsto N) \in H \\ \text{E-OP} \quad \mathbf{handle} \mathcal{E}[(\mathbf{do} \ell V)^E] \mathbf{with} H \rightsquigarrow N[V/p, (\lambda^Y y^B. \mathbf{handle} \mathcal{E}[(\mathbf{return} y)^E] \mathbf{with} H)/r], \\ \quad \text{where } \ell \notin \text{bl}(\mathcal{E}), (\ell p r \mapsto N) \in H, \text{ and } (\ell : A \rightarrow^Y B) \in E \\ \text{E-LIFT} \quad \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \quad \text{if } M \rightsquigarrow N \\ \text{Evaluation contexts } \mathcal{E} ::= [] \mid \mathbf{let}^Y x \leftarrow \mathcal{E} \mathbf{in} N \mid \mathbf{handle} \mathcal{E} \mathbf{with} H \\ \text{bl}([]) = \emptyset \quad \text{bl}(\mathbf{let}^Y x \leftarrow \mathcal{E} \mathbf{in} N) = \text{bl}(\mathcal{E}) \quad \text{bl}(\mathbf{handle} \mathcal{E} \mathbf{with} H) = \text{bl}(\mathcal{E}) \cup \text{dom}(H) \end{array}$$

Fig. 6. Small-step operational semantics of F_{eff}° .

Figure 6 gives a standard small-step operational semantics for F_{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.

3.4 Metatheory

We now prove a type soundness result for F_{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 $\ell \notin \text{bl}(\mathcal{E})$.

Syntactic type soundness of F_{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 F_{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 $E = \{\ell : A \rightarrow^\bullet B; R\}$ and $\ell \notin \text{bl}(\mathcal{E})$, then there exists $\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2$ such that $\Delta \vdash \Gamma_1 : \bullet$ and $\Delta; \Gamma_2, y : B \vdash \mathcal{E}[(\mathbf{return} y)^E] : C$.
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 F_{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 F_{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.

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

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

$$\begin{aligned} \text{lin}(V) &= \begin{cases} \text{true} & \text{if } \cdot; \vdash V : A \text{ and } \cdot \not\vdash A : \bullet \\ \text{false} & \text{otherwise} \end{cases} \\ \text{tag}(V) &= \begin{cases} (V^\circ, \{V^\circ\}) & \text{if } \text{lin}(V) \text{ and } V \neq W^\circ \text{ for any } W \\ (V, \emptyset) & \text{otherwise} \end{cases} \end{aligned}$$

The predicate $\text{lin}(V)$ 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 $\text{tag}(V)$ 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 $M \xrightarrow[\mathcal{T}]{\mathcal{S}} N$, 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 F_{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 $\text{tag}(V)$), 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.

We write $\mathcal{L}(M)$, $\mathcal{L}(V)$, $\mathcal{L}(\mathcal{E})$ and $\mathcal{L}(\mathcal{F})$ 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.

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

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:

L-APP	$(\lambda^Y x^A.M) V$	$\overset{S}{0} \rightsquigarrow M[V'/x]$, where $(V', S) = \text{tag}(V)$
L-TAPP	$(\Lambda^Y \alpha^K.M) T$	$\overset{0}{0} \rightsquigarrow M[T/\alpha]$
L-SEQ	$\text{let}^Y x \leftarrow \text{return } V \text{ in } N$	$\overset{S}{0} \rightsquigarrow N[V'/x]$, where $(V', S) = \text{tag}(V)$
L-RET	$\text{handle } (\text{return } V)^E \text{ with } H$	$\overset{S}{0} \rightsquigarrow N[V'/x]$, where $(\text{return } x \mapsto N) \in H, (V', S) = \text{tag}(V)$
L-OP	$\text{handle } \mathcal{E}[(\text{do } \ell V)^E] \text{ with } H$	$\overset{S}{0} \rightsquigarrow N[V'/p, W'/r]$, where $\ell \notin \text{bl}(\mathcal{E}), (\ell p r \mapsto N) \in H, (\ell : A \twoheadrightarrow^Y B) \in E,$ $W = \lambda^Y y^B. \text{handle } \mathcal{E}[(\text{return } y)^E] \text{ with } H,$ $(V', S_1) = \text{tag}(V), (W', S_2) = \text{tag}(W), S = S_1 \cup S_2$
L-REMOVE	$\mathcal{F}[V^\circ]$	$\overset{0}{\{V^\circ\}} \rightsquigarrow \mathcal{F}[V]$
L-LIFT	$\mathcal{E}[M]$	$\overset{S}{\mathcal{F}} \rightsquigarrow \mathcal{E}[N]$, if $M \overset{S}{\mathcal{F}} \rightsquigarrow N$
Evaluation contexts	$\mathcal{E} ::= [] \mid \text{let}^Y x \leftarrow \mathcal{E} \text{ in } N \mid \text{handle } \mathcal{E} \text{ with } H$	
Tag-removing contexts	$\mathcal{F} ::= [] V \mid [] T$	

Fig. 7. Linearity-aware small-step operational semantics of F_{eff}° .

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

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 F_{eff}° , if $M \overset{S}{\mathcal{F}} \rightsquigarrow N$, then N is linear safe and $\mathcal{L}(M) \cup S = \mathcal{L}(N) \cup \mathcal{T}$.*

The proof can be found in Appendix A.4.

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 \bullet and Any for \circ . The latter is Any as *any* value can be soundly used once. The subkinding relation $\vdash \text{Type}^\bullet \leq \text{Type}^\circ$ ($\text{Unl} \leq \text{Any}$) 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 $\text{fun id}(x)\{x\} : (a::\text{Any}) \rightarrow (a::\text{Any})$. Here, we can instantiate the type variable a to a linear type, such as $! \text{Int}$. 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 oc = send(s, c); close(c) } }
```

²Equivalently, we can say that “for every value subterm W with an unlimited type”.

The inferred type is $(!(a :: \text{Any}).\text{End}) \rightarrow (a :: \text{Any}) \sim @ (\)$. The `faithfulSend` function takes a polymorphic channel `c` and returns a linear function (indicated by $\sim @$ instead of the usual arrow \rightarrow) 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 $\sim @$, in addition to the existing operation space denoted by \Rightarrow . The subkinding relation $\vdash \text{Row}^\circ \leq \text{Row}^\bullet$ ($\text{Lin} \leq \text{Any}$) 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` from Section 2.2.

```
sig dubiousSend : (!String.End) {Choose:()  $\sim @$  Bool|_::Lin} $\sim \rightarrow$  ()
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:() $\sim @$ 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 $\sim @$ to bind linear continuations of control-flow-linear operations.

```
fun(c) {handle ({xlin; dubiousSend(c)}) {case <Choose  $\sim @$  r>  $\rightarrow$  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° , `LINKS` suffers from the limitations outlined in Section 2. In the next section, we present a considerably more expressive calculus, Q_{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.

5 AN IMPLICIT CALCULUS WITH QUALIFIED TYPES

In this section, we propose Q_{eff}° , an ML-style calculus which enhances F_{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 McKinn 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 $\mathcal{Q}_{\text{eff}}^{\circ}$. 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 $\mathcal{F}_{\text{eff}}^{\circ}$, 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	$\text{TySch} \ni \sigma ::= \rho \mid \forall \alpha. \sigma$
Predicates	$\text{Pred} \ni \pi ::= \tau_1 \leq \tau_2 \mid R_1 \leq R_2$ $\mid R \perp \mathcal{L}$	Type contexts	$\text{Env} \ni \Gamma ::= \cdot \mid \Gamma, x : \sigma$
		Predicate sets	$\text{PSet} \ni P ::= \cdot \mid P, \pi$

Fig. 8. Syntax of qualified types of $\mathcal{Q}_{\text{eff}}^{\circ}$.

Linearity. In addition to concrete linearities \circ and \bullet , $\mathcal{Q}_{\text{eff}}^{\circ}$ 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 $\forall \alpha \mu \phi. \alpha \rightarrow^{\phi} \alpha ! \{\mu\}$, 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 $\tau_1 \leq \tau_2$ means the linearity of τ_1 is less than τ_2 (e.g., $\bullet \leq \circ$). 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 $R_1 \leq R_2$ 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 $\text{Row}_{\mathcal{L}}$ 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 $\mathcal{Q}_{\text{eff}}^{\circ}$; 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 $P \vdash \Gamma \leq \tau$), discussed in the following section.

Rule Q-LET demonstrates the treatment of linearity in $\mathcal{Q}_{\text{eff}}^{\circ}$. 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 $\mathcal{Q}_{\text{eff}}^{\circ}$ to generalise subtyping between effect rows. It states that if V is a value of type A_{ℓ} , then **do** ℓV 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

$P \mid \Gamma \vdash V : A$	$P \mid \Gamma \vdash M : C$	$P \mid \Gamma \vdash H : C \Rightarrow D$
------------------------------	------------------------------	--

<p style="text-align: center;">Q-LET</p> $\frac{Q \mid \Gamma_1, \Gamma \vdash V : A \quad \sigma = \text{gen}((\Gamma_1, \Gamma), Q \Rightarrow A) \quad P \mid \Gamma_2, \Gamma, x : \sigma \vdash M : C \quad P \vdash \Gamma \leq \bullet}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{let} \ x = V \ \mathbf{in} \ M : C}$	<p style="text-align: center;">Q-Do</p> $\frac{P \mid \Gamma \vdash V : A_\ell \quad P \vdash \{\ell : A_\ell \rightarrow^Y B_\ell\} \leq R}{P \mid \Gamma \vdash \mathbf{do} \ \ell \ V : B_\ell! \{R\}}$
<p style="text-align: center;">Q-SEQ</p> $\frac{P \mid \Gamma_1, \Gamma \vdash M : A! \{R_1\} \quad P \mid \Gamma_2, \Gamma, x : A \vdash N : B! \{R_2\} \quad P \vdash R_1 \leq R \quad P \vdash R_2 \leq R \quad P \vdash \Gamma_2 \leq R_1 \quad P \vdash \Gamma \leq \bullet}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B! \{R\}}$	<p style="text-align: center;">Q-HANDLER</p> $\frac{H = \{\mathbf{return} \ x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_i \quad C = A! \{(\ell_i : A_i \rightarrow^{Y_i} B_i)_i; R_1\} \quad D = B! \{R_2\} \quad P \mid \Gamma, x : A \vdash M : D \quad [P \mid \Gamma, p_i : A_i, r_i : B_i \rightarrow^{Y_i} D \vdash N_i : D]_i \quad P \vdash \Gamma \leq \bullet \quad P \vdash R_1 \leq R_2 \quad P \vdash R_1 \perp \{\ell_i\}_i}{P \mid \Gamma \vdash H : C \Rightarrow D}$

where $\text{gen}(\Gamma, \rho) = \forall(\text{ftv}(\rho) \setminus \text{ftv}(\Gamma)).\rho$.

Fig. 9. Selected syntax-directed Typing Rules for Q_{eff}°

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

Rule Q-SEQ demonstrates the remaining novelty of qualified types in Q_{eff}° . Several of its uses of entailment follow the previous patterns. The bindings in Γ are available in both M and N , so $P \vdash \Gamma \leq \bullet$ requires that their types be unlimited. We want flexibility in combining the effects in M and N , so the conditions $P \vdash R_i \leq R$ 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 $P \vdash \Gamma_2 \leq R_1$. 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 \leq 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 $P \vdash \Gamma \leq \tau$ which compares the linearity of all variables in Γ and τ .

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 \leq 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 entailment rules for base types, but in practice we would have axioms like $P \vdash \text{Int} \leq \bullet$ and

$P \vdash \pi$	$P \vdash Q$	$P \vdash \sigma \leq \tau$	$P \vdash \Gamma \leq \tau$	
$\frac{\text{P-SUBSUME}}{\pi \in P} \quad \frac{\text{P-REFL}}{P \vdash \tau \leq \tau} \quad \frac{\text{P-LIN}}{P \vdash \tau \leq \circ} \quad \frac{\text{P-UNL}}{P \vdash \bullet \leq \tau} \quad \frac{\text{P-FUN}}{P \vdash Y \leq \tau}}{P \vdash (A \rightarrow^Y C) \leq \tau}$				
$\frac{\text{P-ROW}}{[P \vdash \tau \leq Y]_{(I:A \rightarrow^Y B) \in R}} \quad \frac{\text{P-SUB}}{\text{set}(R_1) \subseteq \text{set}(R_2)} \quad \frac{\text{P-LACK}}{\text{dom}(R) \cap \mathcal{L} = \emptyset} \quad \frac{\text{P-PREDSET}}{[P \vdash \pi]_{\pi \in Q}}$	$\frac{P \vdash \tau \leq \mu \text{ when } \mu \in R}{P \vdash \tau \leq R} \quad \frac{P \vdash R_1 \leq R_2}{P \vdash R_1 \leq R_2} \quad \frac{P \vdash R \perp \mathcal{L}}{P \vdash R \perp \mathcal{L}} \quad \frac{P \vdash Q}{P \vdash Q}$			
$\frac{\text{P-QUANTIFIER}}{P \vdash [\tau' / \alpha] \sigma \leq \tau \text{ for some } \tau'}{P \vdash (\forall \alpha. \sigma) \leq \tau}$	$\frac{\text{P-QUALIFIER}}{P \vdash \pi \quad P \vdash \rho \leq \tau}{P \vdash (\pi \Rightarrow \rho) \leq \tau}$	$\frac{\text{P-CONTEXT}}{[P \vdash \sigma \leq \tau]_{(x:\sigma) \in \Gamma}}{P \vdash \Gamma \leq \tau}$		

Fig. 10. Entailment relations for predicates and other judgement relations.

$P \vdash \circ \leq$ File. For row predicates, we write $\text{set}(R)$ for the set of all elements (comprising operation labels with their signatures and row variables) of R , and $\text{dom}(R)$ 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 $P \vdash \Gamma \leq \tau$ is defined using $P \vdash \sigma \leq \tau$ 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 $\text{id} = \lambda x. \mathbf{return} \ x$ which is obviously unlimited. We give id a polymorphic type $\forall \phi \alpha \mu. \alpha \rightarrow^\phi \alpha! \{\mu\}$ 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 $\sigma = \forall \phi \mu. (\alpha \leq \phi) \Rightarrow () \rightarrow^\phi \alpha! \{\mu\}$ 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 $\sigma \leq \bullet$. Obviously, we want to know from $\sigma \leq \bullet$ that α should be unlimited since x is also duplicated. One

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

$$\frac{\frac{\frac{P \vdash \alpha \leq \phi' \quad \frac{P \vdash \phi' \leq \bullet}{P \vdash () \rightarrow^{\phi'} \alpha! \{\mu'\} \leq \bullet} \text{P-FUNCTION}}{P \vdash (\alpha \leq \phi') \Rightarrow () \rightarrow^{\phi'} \alpha! \{\mu'\} \leq \bullet} \text{P-QUALIFIER}}{P \vdash (\forall \phi \mu. (\alpha \leq \phi) \Rightarrow () \rightarrow^{\phi} \alpha! \{\mu\}) \leq \bullet} \text{P-QUANTIFIER}}$$

In P-QUANTIFIER we instantiate ϕ and μ with variables ϕ' and μ' . In order to prove $\sigma \leq \bullet$ from P , we must then prove $\alpha \leq \phi'$ and $\phi' \leq \bullet$. Note that ϕ' and μ' are not fresh, but should instead appear in P , e.g., we might have $P = \{\alpha \leq \phi', \phi' \leq \bullet\}$. If we instead assumed $\alpha \leq \phi$, 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 $\sigma \leq \bullet$. 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 $\llbracket P \rrbracket_{\text{sat}} = \{\theta \mid \cdot \vdash \theta P\}$. Transitivity of \leq is admissible when considering the solutions of predicates, e.g., $\llbracket \phi_1 \leq \phi_2, \phi_2 \leq \bullet \rrbracket_{\text{sat}} = \llbracket \phi_1 \leq \phi_2, \phi_2 \leq \bullet, \phi_1 \leq \bullet \rrbracket_{\text{sat}} = \{[\bullet/\phi_1, \bullet/\phi_2]\}$. 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 $\mathbb{Q}_{\text{eff}}^{\circ}$; 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^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^W emits the constraint that the singleton effect row be included in the output row. Rule Q-SEQ^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 $\theta|_{\Gamma} = \theta'|_{\Gamma}$.

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

Q-LET^W

$$\frac{\Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \quad \sigma = \text{gen}(\theta_1 \Gamma, P_1 \Rightarrow A) \quad \theta_1 \Gamma, x : \sigma \vdash M : C \dashv \theta_2, P_2, \Sigma_2 \quad Q = \text{un}(\theta_2 \theta_1 \Gamma |_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : \sigma) |_{\Sigma_2^c})}{\Gamma \vdash \mathbf{let} \ x = V \ \mathbf{in} \ M : C \dashv \theta_2 \theta_1, P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)}$$

Q-Do^W

$$\frac{\Gamma \vdash V : A \dashv \theta_1, P, \Sigma \quad A \sim A_\ell : \theta_2 \quad \mu, \phi \ \text{fresh} \quad Q = \text{sub}((\ell : A_\ell \xrightarrow{\phi} B_\ell), \mu)}{\Gamma \vdash \mathbf{do} \ \ell \ V : B_\ell ! \{\mu\} \dashv \theta_2 \theta_1, \theta_2 P \cup Q, \Sigma}$$

Q-SEQ^W

$$\frac{\Gamma \vdash M : A ! \{R_1\} \dashv \theta_1, P_1, \Sigma_1 \quad \theta_1 \Gamma, x : A \vdash N : B ! \{R_2\} \dashv \theta_2, P_2, \Sigma_2 \quad \mu \ \text{fresh} \quad Q = \text{un}(\theta_2 \theta_1 \Gamma |_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : A) |_{\Sigma_2^c}) \cup \text{leq}(\theta_2 \theta_1 \Gamma |_{\Sigma_2}, \theta_2 R_1) \cup \text{sub}(\theta_2 R_1, \mu) \cup \text{sub}(R_2, \mu)}{\Gamma \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B ! \mu \dashv \theta_2 \theta_1, \theta_2 P_1 \cup P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)}$$

$$\text{leq}(\Gamma, \tau) = \text{factorise}(\Gamma \leq \tau) \quad \text{un}(\Gamma) = \text{leq}(\Gamma, \bullet) \quad \text{sub}(R_1, R_2) = \text{factorise}(R_1 \leq R_2)$$

Fig. 11. Selected type inference rules for Q_{eff}°

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.*

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 Factoring Predicates

factorise : Pred \rightarrow PSetfactorise($\tau \leq \tau$) = \emptyset factorise($\tau \leq \circ$) = \emptyset factorise($\bullet \leq \tau$) = \emptyset factorise($A \rightarrow^Y C \leq \tau$) = factorise($Y \leq \tau$)factorise($\tau \leq K ; \mu$) =factorise($\tau \leq K$) \cup factorise($\tau \leq \mu$)factorise($\tau \leq K$) = $\bigcup_{(\ell : A \rightarrow^Y B) \in K} \text{factorise}(\tau \leq Y)$ factorise($R_1 \leq R_2$) = \emptyset , when $\text{set}(R_1) \subseteq \text{set}(R_2)$ factorise($R \perp \mathcal{L}$) = \emptyset , when $\text{dom}(R) \cap \mathcal{L} = \emptyset$ factorise(π) = π factorise : (TySch \leq Type) \rightarrow PSetfactorise($(\forall \alpha. \sigma) \leq \tau$) =factorise($[\beta/\alpha] \sigma \leq \tau$) for some fresh β factorise($(\pi \Rightarrow \sigma) \leq \tau$) =factorise(π) \cup factorise($\sigma \leq \tau$)factorise : (Env \leq Type) \rightarrow PSetfactorise($\Gamma \leq \tau$) = $\bigcup_{(x:\sigma) \in \Gamma} \text{factorise}(\sigma \leq \tau)$ factorise : PSet \rightarrow PSetfactorise(P) = $\bigcup_{\pi \in P} \text{factorise}(\pi)$

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 $(\forall\alpha.\sigma) \leq \tau$. Rule P-QUANTIFIER requires that we find some instance such that $\sigma[\tau'/\alpha] \leq \tau$. 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. The proof can be found in Appendix C.1.

THEOREM 5.3 (CORRECTNESS OF FACTORISATION). *If $\text{factorise}(P) = Q$, then $Q \vdash P$ and $P \vdash Q$. If $\text{factorise}(\Gamma \leq \tau) = Q$, then $Q \vdash \Gamma \leq \tau$ and for any $P \vdash \Gamma \leq \tau$, there exists θ such that $P \vdash \theta Q$.*

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 $\text{solve}(P)$ 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, $R_1 \leq R_2$ and $\tau \leq R_2$ actually implies $\tau \leq R_1$. To resolve the interaction, the algorithm proceeds by first transforming row subtyping constraints to those of the forms $\mu \leq R$, so that we can always simply instantiate μ on the left to the empty row \cdot for which $\tau \leq \cdot$ always holds. Then, the algorithm computes the transitive closure of linearity constraints and rejects $\circ \leq \bullet$. 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 $\llbracket P \rrbracket_{\text{sat}} \theta$ for the substitution set $\{\theta' \theta \mid \theta' \in \llbracket P \rrbracket_{\text{sat}}\}$.

THEOREM 5.4 (CORRECTNESS OF CONSTRAINT SOLVING). *For any constraint set P generated by the type inference of $\mathcal{Q}_{\text{eff}}^\circ$, $\text{solve}(P)$ always terminates.*

- If it fails, then P is not satisfiable.
- If it returns (θ, Q) , then P is satisfiable and $\llbracket P \rrbracket_{\text{sat}} = \llbracket Q \rrbracket_{\text{sat}} \theta$.

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 practice concerns. Some existing algorithms on simplifying general subtyping constraints [Pottier 1998, 2001] are promising.

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.

Let us first consider shallow handlers in F_{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.

$$\text{E-OP}^\dagger \quad \mathbf{handle} \mathcal{E}[(\mathbf{do} \ell V)^E] \mathbf{with} H^\dagger \rightsquigarrow N[V/p, (\lambda^Y y^B. \mathcal{E}[(\mathbf{return} y)^E])/r],$$

where $\ell \notin \text{bl}(\mathcal{E})$, $(\ell p r \mapsto N) \in H^\dagger$ and $(\ell : A \rightarrow^Y B) \in E$

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.

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

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 Q_{eff}° with shallow handlers.

$$\begin{array}{c} \text{Q-SHALLOWHANDLER} \\ H = \{\mathbf{return} x \mapsto M\} \uplus \{\ell_i p_i r_i \mapsto N_i\}_i \\ C = A! \{(\ell_i : A_i \rightarrow^{Y_i} B_i)_i; R_1\} \quad D = B! \{R_2\} \\ P \mid \Gamma, x : A \vdash M : D \quad [P \mid \Gamma, p_i : A_i, r_i : B_i \rightarrow^{Y_i} C \vdash N_i : D]_i \\ P \vdash \Gamma \leq R_1 \quad P \vdash R_1 \leq R_2 \quad P \vdash R_1 \perp \{\ell_i\}_i \\ \hline P \mid \Gamma \vdash H : C \rightrightarrows D \end{array}$$

In place of $P \vdash \Gamma \leq \bullet$ in Q-HANDLER, we have $P \vdash \Gamma \leq R_1$, 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 F_{eff}° and Q_{eff}° with recursive functions. 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. In addition to our two new typing rules, shallow handlers can actually introduce more challenges to tracking control flow linearity, especially because shallow handlers do not handle all invocations of the same operation uniformly. As a result, we can give different control flow linearity to them. We defer a full analysis of the interaction between shallow handlers and control flow linearity to future work.

7 RELATED WORK

In this section, we discuss related work on linear types, effect types, multi-shot continuations, and effect handlers.

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 on 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. Leijen and Brachthäuser [2018] 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 and a syntactic check to ensure that resumptions are not invoked more than once.

Structural types and control effects. Tov and Pucella [2011b] propose a calculus $\lambda^{\text{URAL}}(\mathcal{E})$ which extends the substructural λ -calculus λ^{URAL} [Ahmed et al. 2005] with abstract control effects \mathcal{E} given by a set of effects, a pure effect, and an effect-sequencing operator. They show how to instantiate $\lambda^{\text{URAL}}(\mathcal{E})$ with concrete control effects including exceptions and shift/reset [Danvy and Filinski 1990] separately. Similar to F_{eff}° and Q_{eff}° , the $\lambda^{\text{URAL}}(\mathcal{E})$ 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 $\lambda^{\text{URAL}}(\mathcal{E})$ 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 $\lambda^{\text{URAL}}(\mathcal{E})$ 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 F_{eff}° and Q_{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 $\text{dup} : \tau \rightarrow (\tau, \tau)$ and $\text{drop} : \tau \rightarrow ()$. Morris [2016] extends the approach of Tov and Pucella to generalise the treatment of function types, introducing the linearity ordering constraint $\tau \leq \nu$; 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 F_{eff}° should work well with all kinds of different row-based effect systems.

Subtyping-based effect types. Some versions of EFF [Bauer and Pretnar 2013; 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 Q_{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 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] 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, it's 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, and as a consequence they must copy the

continuation a priori [Hillerström et al. 2016]. Our type systems can enable static optimisation of one-shot continuations as they statically distinguish linear from 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 $\mathcal{Q}_{\text{eff}}^{\circ}$; 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 $\mathcal{Q}_{\text{eff}}^{\circ}$ to algebraic subtyping [Dolan and Mycroft 2017]; and adapting control flow linearity for uniqueness types and for quantitative type theory [Atkey 2018; McBride 2016].

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 *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 78–91. <https://doi.org/10.1145/1086365.1086376>
- Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. <https://doi.org/10.1145/3209108.3209189>
- Andrew Barber. 1996. Dual Intuitionistic Linear Logic. Technical report ECS-LFCS-96-347, University of Edinburgh.
- Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.
- Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123.
- 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.
- Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020), 30 pages. <https://doi.org/10.1145/3428194>
- Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). 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 *Proceedings of the ACM SIGPLAN’96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 99–107. <https://doi.org/10.1145/231379.231395>
- Guillaume Combette and Guillaume Munch-Maccagnoni. 2018. A resource modality for RAIL. 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.

- Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) (POPL '82). Association for Computing Machinery, New York, NY, USA, 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.
- 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.
- Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.
- 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.
- 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.
- Edward Gan, Jesse A. Tov, and Greg Morrisett. 2014. Type Classes for Lightweight Substructural Types. In *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014*. 34–48.
- 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.
- Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). 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/effect-handlers/wasm-spec/proposals/continuations/Explainer.md>.
- Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers (*TyDe 2016*). Association for Computing Machinery, New York, NY, USA, 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.
- 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.
- Daniel Hillerström, Sam Lindley, and John Longley. 2023. Asymptotic Speedup with Effect Handlers. Draft.
- Daniel Hillerström, Sam Lindley, and KC Sivamarakrishnan. 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](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.
- 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.
- Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 6 (2005), 179–194.
- Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 283–294. <https://doi.org/10.1145/1411204.1411245>
- Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) (POPL '17). Association for Computing Machinery, New York, NY, USA, 486–499. <https://doi.org/10.1145/3009837.3009872>
- Daan Leijen and Jonathan Immanuel Brachthäuser. 2018. Taming Control Flow through Linear Effect Handlers. <https://www.youtube.com/watch?v=yozwdQ4f8G4>. HOPE@ICFP'18 talk. [Accessed 2023-07-11].

- Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Languages Design and Implementation, TLDI 2012, Philadelphia, PA, USA, Saturday, January 28, 2012*, Benjamin C. Pierce (Ed.). ACM, 91–102. <https://doi.org/10.1145/2103786.2103798>
- Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (Paris, France) (POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 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 *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (Nara, Japan) (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 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, Article 12 (jan 2019), 28 pages. <https://doi.org/10.1145/3290325>
- 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.
- 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](https://doi.org/10.2168/LMCS-10(3:21)2014)
- Didier Rémy. 1994. Type Inference for Records in a Natural Extension of ML. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. Citeseer.
- K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221. <https://doi.org/10.1145/3453483.3454039>
- Jesse A. Tov and Riccardo Pucella. 2011a. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). 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 *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). 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.

A PROOFS OF F_{eff}°

In this section, we prove the theorems in Section 3.

A.1 Unlimited is Unlimited

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 $E = \{\ell : A \twoheadrightarrow^* B; R\}$ and $\ell \notin \text{bl}(\mathcal{E})$, then there exists $\Delta \vdash \Gamma = \Gamma_1 + \Gamma_2$ such that $\Delta \vdash \Gamma_1 : \bullet$ and $\Delta; \Gamma_1, y : B \vdash \mathcal{E}[(\mathbf{return} y)^E] : C$.

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

PROOF.

1. *Unlimited values are unlimited.* By induction on the typing derivation $\Delta; \Gamma \vdash V : A$.

Case T-VAR. Trivial.

Case T-ABS. $\Delta \vdash A \rightarrow^Y C : \bullet$ gives $Y = \bullet$, which then gives $\Delta \vdash \Gamma : \bullet$.

Case T-TABS. $\Delta \vdash \forall^Y \alpha^K. C : \bullet$ gives $Y = \bullet$, which then gives $\Delta \vdash \Gamma : \bullet$.

2. *Unlimited continuations are unlimited.* By $\ell \notin \text{bl}(\mathcal{E})$ and straightforward induction on typing derivations, we have $C = _! \{ \ell : A \rightarrow^\bullet B; _ \}$. By induction on $\Delta; \Gamma \vdash \mathcal{E}[(\mathbf{do} \ell V)^E] : C$.

Case

$$\frac{\text{T-Do} \quad E = \{ \ell : A \rightarrow^Y B; R \} \quad \Delta; \Gamma \vdash V : A \quad \Delta \vdash E : \text{Effect}}{\Delta; \Gamma \vdash (\mathbf{do} \ell V)^E : B!E}$$

Immediately, we have $\Delta; y : B \vdash (\mathbf{return} y)^E : B!E$ and $\Delta \vdash \cdot : \bullet$.

Case

$$\frac{\text{T-SEQ} \quad \begin{array}{l} \Delta; \Gamma_1 \vdash \mathcal{E}'[(\mathbf{do} \ell V)^E] : A'!E' \text{ (1)} \quad \Delta; \Gamma_2, x : A' \vdash N : B'!E' \\ E' = \{ \ell : A \rightarrow^\bullet B; R' \} \quad \Delta \vdash \Gamma_2 : Y \text{ (2)} \quad \Delta \vdash (\ell : A \rightarrow^\bullet B; R') : Y \text{ (3)} \end{array}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow \mathcal{E}'[(\mathbf{do} \ell V)^E] \mathbf{in} N : B'!E'}$$

By (3), we have $Y = \bullet$. Then, by (2), we have $\Delta \vdash \Gamma_2 : \bullet$. By the IH on (1), there exists $\Delta \vdash \Gamma_1 = \Gamma_{11} + \Gamma_{12}$ such that $\Delta \vdash \Gamma_{11} : \bullet$ and $\Delta; \Gamma_{11}, y : B \vdash \mathcal{E}'[(\mathbf{return} y)^E] : A'!E'$. Applying T-SEQ to it, we have $\Delta; \Gamma_3, y : B \vdash \mathbf{let}^Y x \leftarrow \mathcal{E}'[(\mathbf{return} y)^E] \mathbf{in} N : B'!E'$, $\Delta \vdash \Gamma = \Gamma_{12} + \Gamma_3$ and $\Delta \vdash \Gamma_3 : \bullet$ where $\Delta \vdash \Gamma_3 = \Gamma_2 + \Gamma_{11}$.

Case

$$\frac{\text{T-HANDLE} \quad \begin{array}{l} \Delta; \Gamma_1 \vdash \mathcal{E}'[(\mathbf{do} \ell V)^E] : A'!E' \text{ (1)} \quad \Delta; \Gamma_2 \vdash H : A'!E' \rightrightarrows B'!F' \text{ (2)} \end{array}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle} \mathcal{E}'[(\mathbf{do} \ell V)^E] \mathbf{with} H : B'!F'}$$

By (2), we have $\Delta \vdash \Gamma_2 : \bullet$. By the IH on (1), there exists $\Delta \vdash \Gamma_1 = \Gamma_{11} + \Gamma_{12}$ such that $\Delta \vdash \Gamma_{11} : \bullet$ and $\Delta; \Gamma_{11}, y : B \vdash \mathcal{E}'[(\mathbf{return} y)^E] : A'!E'$. Applying T-HANDLE to it, we have $\Delta; \Gamma_3, y : B \vdash \mathbf{handle} \mathcal{E}'[(\mathbf{return} y)^E] \mathbf{with} H : B'!F'$, $\Delta \vdash \Gamma = \Gamma_{12} + \Gamma_3$ and $\Delta \vdash \Gamma_3 : \bullet$ where $\Delta \vdash \Gamma_3 = \Gamma_2 + \Gamma_{11}$.

3. *Deep handlers are unlimited.* Directly follows from T-HANDLER. □

A.2 Progress

LEMMA A.1 (CANONICAL FORMS).

1. If $\vdash V : A \rightarrow^Y B$, then V is of shape $\lambda^Y x^A. M$.
2. If $\vdash V : \forall^Y \alpha^K. C$, then V is of shape $\Lambda^Y \alpha^K. M$.

PROOF. Directly follows from the typing rules. □

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 .

PROOF. By induction on the typing derivation $\vdash M : A!E$.

Case

$$\frac{\text{T-APP} \quad \vdash V : A \rightarrow^Y C \quad \vdash W : A}{\vdash VW : C}$$

By Lemma A.1, we have $V = \lambda^Y x^A.M$. Reduced by E-APP.

Case

$$\frac{\text{T-TAPP} \quad \Delta; \Gamma \vdash V : \forall^Y \alpha^K . C \quad \Delta \vdash T : K}{\Delta; \Gamma \vdash VT : C[T/\alpha]}$$

By Lemma A.1, we have $V = \Lambda^Y \alpha^K.M$. Reduced by E-TAPP.

Case T-RETURN. In a normal form with respect to E .

Case T-DO. In a normal form with respect to E .

Case

$$\frac{\text{T-SEQ} \quad \begin{array}{l} \Delta; \Gamma_1 \vdash M : A!E \quad \Delta; \Gamma_2, x : A \vdash N : B!E \\ E = \{R\} \quad \Delta \vdash \Gamma_2 : Y \quad \Delta \vdash R : Y \end{array}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow M \mathbf{in} N : B!E}$$

By a case analysis on M .

Subcase $M = (\mathbf{return} N)^E$. Reduced by E-SEQ.

Subcase Otherwise. By the IH, if $M \rightsquigarrow N$, then the original term is reduced by E-LIFT. Otherwise, M is in a normal form with respect to E , which implies the original term is also in a normal form with respect to E .

Case

$$\frac{\text{T-HANDLE} \quad \begin{array}{l} \Delta; \Gamma_1 \vdash H : C \Rightarrow D \quad \Delta; \Gamma_2 \vdash M : C \quad C = A!E' \quad D = B!E \end{array}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle} M \mathbf{with} H : D}$$

By a case analysis on M .

Subcase $M = (\mathbf{return} N)^{E'}$. Reduced by E-RET.

Subcase $M = \mathcal{E}[(\mathbf{do} \ell V)^{E''}]$ with $\ell \notin \text{bl}(\mathcal{E})$ and $(\ell pr \mapsto N) \in H$. The original term is reduced by E-OP.

Subcase Otherwise. By the IH, if $M \rightsquigarrow N$, then the original term is reduced by E-LIFT. Otherwise, M is in a normal form with respect to E' . By Definition 3.1, $M = \mathcal{E}[(\mathbf{do} \ell V)^{E''}]$ for $\ell \in E'$ and $\ell \notin \text{bl}(\mathcal{E})$. By the last subcase, ℓ is also not handled by H . Thus, the original term is also in a normal form with respect to E .

□

A.3 Subject Reduction

LEMMA A.2 (SUBSTITUTION).

1. *Preservation of kinds under type substitution: if $\Delta, \alpha : K' \vdash T : K$ and $\Delta \vdash T' : K'$, then $\Delta \vdash T[T'/\alpha] : K$.*
2. *Preservation of types under type substitution: if $\Delta \vdash T : K$, then $\Delta, \alpha : K; \Gamma \vdash M : C$ implies $\Delta; \Gamma[T/\alpha] \vdash M[T/\alpha] : C[T/\alpha]$, and $\Delta, \alpha : K; \Gamma \vdash V : A$ implies $\Delta; \Gamma[T/\alpha] \vdash V[T/\alpha] : A[T/\alpha]$, and $\Delta, \alpha : K; \Gamma \vdash H : C \Rightarrow D$ implies $\Delta; \Gamma[T/\alpha] \vdash H[T/\alpha] : (C \Rightarrow D)[T/\alpha]$.*

3. *Preservation of types under value substitution:* if $\Delta \vdash \Gamma_1 : Y$, $\Delta; \Gamma_1 \vdash V : A$ and $\Delta \vdash A : Y$, then $\Delta; \Gamma_2, x : A \vdash M : C$ implies $\Delta; \Gamma_1 + \Gamma_2 \vdash M[V/x] : C$, and $\Delta; \Gamma_2, x : A \vdash W : B$ implies $\Delta; \Gamma_1 + \Gamma_2 \vdash W[V/x] : B$, and $\Delta; \Gamma_2, x : A \vdash H : C \Rightarrow D$ implies $\Delta; \Gamma_1 + \Gamma_2 \vdash H[V/x] : C \Rightarrow D$.

PROOF. We apply various structural lemmas like weakening, permutation of contexts, and properties of context splitting in the following proofs.

1. *Preservation of kinds under type substitution.* Straightforward induction on the kinding derivations.
2. *Preservation of types under type substitution.* By Lemma A.2.1 and straightforward mutual induction on the typing derivations.
3. *Preservation of types under value substitution.* By mutual induction on the typing derivations. \square

THEOREM 3.3 (SUBJECT REDUCTION). *If $\Delta; \Gamma \vdash M : C$ and $M \rightsquigarrow N$, then $\Delta; \Gamma \vdash N : C$.*

PROOF. By induction on the typing derivation $\Delta; \Gamma \vdash M : C$.

Case

$$\frac{\text{T-APP} \quad \Delta; \Gamma_1 \vdash V : A \rightarrow^Y C \text{ (1)} \quad \Delta; \Gamma_2 \vdash W : A \text{ (2)}}{\Delta; \Gamma_1 + \Gamma_2 \vdash V W : C}$$

The reduction can only be derived using E-APP, which implies $V = \lambda^Y x^A. N$ and $(\lambda^Y x^A. N) W \rightsquigarrow N[W/x]$. Inversion on (1) gives $\Delta; \Gamma_1, x : A \vdash N : C$ (3). Case analysis on the linearity of A :

Subcase $\Delta \vdash A : \bullet$ (4). Applying Theorem 3.4.1 to (2) gives $\Delta \vdash \Gamma_2 : \bullet$ (5). Applying Lemma A.2.3 to (2), (3), (4) and (5) gives $\Delta; \Gamma_1 + \Gamma_2 \vdash N[W/x] : C$.

Subcase $\Delta \vdash A : \circ$ (4). We always have $\Delta \vdash \Gamma_2 : \circ$ (5). Applying Lemma A.2.3 to (2), (3), (4) and (5) gives $\Delta; \Gamma_1 + \Gamma_2 \vdash N[W/x] : C$.

Case

$$\frac{\text{T-TAPP} \quad \Delta; \Gamma \vdash V : \forall^Y \alpha^K. C \text{ (1)} \quad \Delta \vdash T : K \text{ (2)}}{\Delta; \Gamma \vdash V T : C[T/\alpha]}$$

The reduction can only be derived using E-TAPP, which implies $V = \Lambda^Y \alpha^K. N$ and $(\Lambda^Y \alpha^K. N) T \rightsquigarrow N[T/\alpha]$. Inversion on (1) gives $\Delta, \alpha : K; \Gamma \vdash N : C$ (3). By $\alpha \notin \text{ftv}(\Gamma)$, applying Lemma A.2.2 to (2) and (3) gives $\Delta; \Gamma \vdash N[T/\alpha] : C[T/\alpha]$.

Case T-RETURN. No reduction. M is in a normal form.

Case T-DO. No reduction. M is in a normal form.

Case

$$\frac{\text{T-SEQ} \quad \Delta; \Gamma_1 \vdash M : A! \{R\} \text{ (1)} \quad \Delta; \Gamma_2, x : A \vdash N : B! \{R\} \text{ (2)}}{\Delta \vdash \Gamma_2 : Y \quad \Delta \vdash R : Y} \quad \Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow M \mathbf{in} N : B! \{R\}$$

By a case analysis on the next rule used by reduction:

Subcase E-LIFT. Suppose $M \rightsquigarrow M'$. The IH on (1) gives $\Delta; \Gamma_1 \vdash M' : A! \{R\}$. Then, by T-SEQ we have $\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}^Y x \leftarrow M' \mathbf{in} N : B! \{R\}$.

Subcase E-SEQ. $M = (\mathbf{return} V)^{\{R\}}$. Inversion on (1) gives $\Delta; \Gamma_1 \vdash V : A$ (3). With (2) and (3), our goal follows from a case analysis on the linearity of A similar to the T-APP case.

Case

$$\frac{\text{T-HANDLE} \quad \Delta; \Gamma_1 \vdash M : C \text{ (1)} \quad \Delta; \Gamma_2 \vdash H : C \Rightarrow D \text{ (2)}}{\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle} \ M \ \mathbf{with} \ H : D}$$

By a case analysis on the next rule used by reduction:

Subcase E-LIFT. Suppose $M \rightsquigarrow M'$. The IH on (1) gives $\Delta; \Gamma_1 \vdash M' : C$. Then, by T-HANDLE we have $\Delta; \Gamma_1 + \Gamma_2 \vdash \mathbf{handle} \ M' \ \mathbf{with} \ H : D$.

Subcase E-RET. $M = (\mathbf{return} \ V)^E$ and $(\mathbf{return} \ x \mapsto N) \in H$. Suppose $C = A!E$. Inversion on (1) gives $\Delta; \Gamma_1 \vdash V : A$ (3). Inversion on (2) gives $\Delta; \Gamma_2, x : A \vdash N : D$ (4). With (3) and (4), our goal follows from a case analysis on the linearity of A similar to the T-APP case.

Subcase E-OP. $M = \mathcal{E}[(\mathbf{do} \ \ell \ V)^E]$, $\ell \notin \text{bl}(\mathcal{E})$ and $(\ell \ p \ r \mapsto N) \in H$. Suppose $(\ell : A \rightarrow^Y B) \in E$ and $W = \lambda^Y y^B. \mathbf{handle} \ \mathcal{E}[(\mathbf{return} \ y)^E] \ \mathbf{with} \ H$. The reduction is $\mathbf{handle} \ M \ \mathbf{with} \ H \rightsquigarrow N[V/p, W/r]$. Inversion on (2) gives $\Delta; \Gamma_2, p : A, r : B \rightarrow^Y D \vdash N : D$ (3). By a straightforward induction on (1) similar to the proof of Theorem 3.4.2, it is easy to show that there exists $\Delta \vdash \Gamma_1 = \Gamma_{11} + \Gamma_{12}$ such that $\Delta; \Gamma_{11} \vdash V : A$ (4) and $\Delta; \Gamma_{12}, y : B \vdash \mathcal{E}[(\mathbf{return} \ y)^E] : C$ (5). With (3) and (4), by a case analysis on the linearity of A similar to the T-APP case, we have $\Delta; \Gamma_{11} + (\Gamma_2, r : B \rightarrow^Y D) \vdash N[V/p] : D$ (6). Then by another case analysis on Y :

subcase $Y = \bullet$. By Theorem 3.4.2 we have $\Delta \vdash \Gamma_{12} : \bullet$. Applying T-HANDLE and T-ABS to (5), we have $\Delta; \Gamma_{12} + \Gamma_2 \vdash W : B \rightarrow^Y D$ (7). Applying Theorem 3.4.3 to (2) we have $\Delta \vdash \Gamma_2 : \bullet$. Finally, applying Lemma A.2.3 to (6) and (7), we have $\Delta; \Gamma_{11} + \Gamma_{12} + \Gamma_2 \vdash N[V/p, W/r] : D$.

subcase $Y = \circ$. Applying T-HANDLE and T-ABS to (5), we have $\Delta; \Gamma_{12} + \Gamma_2 \vdash W : B \rightarrow^Y D$ (7). We always have $\Delta \vdash \Gamma_{12} + \Gamma_2 : \circ$. Finally, applying Lemma A.2.3 to (6) and (7), we have $\Delta; \Gamma_{11} + \Gamma_{12} + \Gamma_2 \vdash N[V/p, W/r] : D$.

□

A.4 Linearity Safety of Evaluation

LEMMA A.3 (LINEAR VARIABLES APPEAR EXACTLY ONCE). *If $\Delta; \Gamma, x : A \vdash V : B$ and $\Delta \not\vdash A : \bullet$, then x appears exactly once in V . If $\Delta; \Gamma, x : A \vdash M : C$ and $\Delta \not\vdash A : \bullet$, then x appears exactly once in M .*

PROOF. By the definition of the context splitting relation and straightforward induction on typing derivations. □

LEMMA A.4 (PRESERVATION OF LINEAR SAFETY UNDER SUBSTITUTION). *Given closed and linear safe V and M , if $\vdash V : A$ and $\cdot; x : A \vdash M : C$, then $M[V'/x]$ is linear safe where $(V', _) = \text{tag}(V)$.*

PROOF. Case analysis on the linearity of A .

Case $\vdash A : \bullet$. We have $V' = V$. By the linear safety of V , we have $\mathcal{L}(V) = \emptyset$. The linear safety of $M[V'/x]$ follows from the linear safety of M .

Case $\not\vdash A : \bullet$. By Theorem 3.4, x does not appear in unlimited values, continuations and handlers of M . Thus, V' does not appear in unlimited values, continuations and handlers of $M[V'/x]$. The linear safety of $M[V'/x]$ then directly follows from the linear safety of M and V .

□

THEOREM 3.6 (REDUCTION SAFETY). *For any closed, well-typed and linear safe computation M in $\mathbb{F}_{\text{eff}}^\circ$, if $M \xrightarrow{\mathcal{T}} N$, then N is linear safe and $\mathcal{L}(M) \cup \mathcal{S} = \mathcal{L}(N) \cup \mathcal{T}$.*

PROOF. We proceed by induction on the linearity-aware reduction rules defined in Figure 7. To avoid name conflicts, we consider $\hat{M} \xrightarrow[\tau]{S} \hat{N}$.

Case

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

The linear safety of \hat{M} gives the linear safety of M and V . The linear safety of \hat{N} follows from Lemma A.4. By inversion on \hat{M} , V has type A . Case analysis on the linearity of A :

Subcase $\vdash A : \bullet$. We have $\text{lin}(V) = \text{false}$ and $\text{tag}(V) = \{V, \emptyset\}$. By the fact that V is closed and linear safe, we have $\mathcal{L}(V) = \emptyset$. Our goal follows from $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(M) = \mathcal{L}(\hat{N}) \cup \emptyset$.

Subcase $\nVdash A : \bullet$. We have $\text{lin}(V) = \text{true}$. By Lemma A.3, x appears in M exactly once. If $V = W^\circ$ for some W , then we have $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(M) \cup \mathcal{L}(V) = \mathcal{L}(M[V/x]) = \mathcal{L}(\hat{N}) \cup \emptyset$. Otherwise, we have $\mathcal{L}(\hat{M}) \cup \{V^\circ\} = \mathcal{L}(M) \cup \mathcal{L}(V) \cup \{V^\circ\} = \mathcal{L}(M) \cup \mathcal{L}(V^\circ) = \mathcal{L}(M[V^\circ/x]) = \mathcal{L}(\hat{N}) \cup \emptyset$.

Case

$$\text{L-TAPP} \quad (\Lambda^Y \alpha^K.M) T \xrightarrow[\emptyset]{\emptyset} M[T/\alpha]$$

The linear safety of \hat{N} directly follows from the linear safety of \hat{M} . We have $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(M) = \mathcal{L}(\hat{N}) \cup \emptyset$.

Case

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

The linear safety of \hat{M} gives the linear safety of N and V . The linear safety of \hat{N} follows from Lemma A.4. Suppose $\vdash V : A$. Our goal follows from a case analysis on the linearity of A similar to the L-APP case.

Case

$$\begin{aligned} \text{L-RET} \quad \mathbf{handle} (\mathbf{return} V)^E \mathbf{with} H &\xrightarrow[\emptyset]{S} N[V'/x], \\ \text{where } (\mathbf{return} x \mapsto N) \in H, (V', \mathcal{S}) &= \text{tag}(V) \end{aligned}$$

The linear safety of \hat{M} gives the linear safety of V , H and N . The linear safety of \hat{N} follows from Lemma A.4. Suppose $\vdash V : A$. Our goal follows from a case analysis on the linearity of A similar to the L-APP case.

Case

$$\begin{aligned} \text{L-OP} \quad \mathbf{handle} \mathcal{E}[(\mathbf{do} \ell V)^E] \mathbf{with} H &\xrightarrow[\emptyset]{S} N[V'/p, W'/r], \\ \text{where } \ell \notin \text{bl}(\mathcal{E}), (\ell p r \mapsto N) \in H, (\ell : A \rightarrow^Y B) \in E, \\ W = \lambda^Y y^B. \mathbf{handle} \mathcal{E}[(\mathbf{return} y)^E] \mathbf{with} H, \\ (V', \mathcal{S}_1) = \text{tag}(V), (W', \mathcal{S}_2) = \text{tag}(W), \mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \end{aligned}$$

The linear safety of \hat{M} gives the linear safety of V , H , N and \mathcal{E} . We need to show the linear safety of W . If $Y = \circ$, the linear safety of W directly follows from the linear safety of \mathcal{E} and H . If $Y = \bullet$, by the linear safety of $\mathcal{E}[(\mathbf{do} \ell V)^E]$ we have $\mathcal{L}(\mathcal{E}) = \emptyset$. By the linear safety of H we have $\mathcal{L}(H) = \emptyset$. Thus, $\mathcal{L}(W) = \emptyset$, which gives us the linear safety of W . The linear safety of \hat{N} follows from Lemma A.4. Then, we prove the equation. By inversion on $(\mathbf{do} \ell V)^E$, we have $\vdash V : A$. Suppose $\vdash W : B \rightarrow^Y C$. By the linear safety of H , we have $\mathcal{L}(H) = \mathcal{L}(N) = \emptyset$. By a case analysis on the linearity of A .

Subcase $\vdash A : \bullet$. We have $\text{lin}(V) = \text{false}$ and $\text{tag}(V) = \{V, \emptyset\}$. By the fact that V is closed and linear safe, we have $\mathcal{L}(V) = \emptyset$. By a case analysis on the linearity of $B \rightarrow^Y C$.

subcase $\vdash B \rightarrow^Y C : \bullet$. We have $\text{lin}(W) = \text{false}$ and $\text{tag}(W) = \{W, \emptyset\}$. By the fact that W is closed and linear safe, we have $\mathcal{L}(W) = \emptyset$. Our goal follows from $\mathcal{L}(\hat{M}) \cup \emptyset = \emptyset = \mathcal{L}(\hat{N}) \cup \emptyset$.

subcase $\vdash B \rightarrow^Y C : \circ$. We have $\text{lin}(W) = \text{true}$ and $\text{tag}(W) = \{W^\circ, \{W^\circ\}\}$. By Lemma A.3, r appears in N exactly once. We have $\mathcal{L}(\hat{M}) \cup \{W^\circ\} = \mathcal{L}(\mathcal{E}) \cup \{W^\circ\} = \mathcal{L}(W^\circ) = \mathcal{L}(\hat{N})$.

Subcase $\varkappa A : \bullet$. We have $\text{lin}(V) = \text{true}$. By Lemma A.3, p appears in N exactly once. If $V = V_1^\circ$ for some V_1 , we have $V^\circ = (V, \emptyset)$. By a case analysis on the linearity of $B \rightarrow^Y C$.

subcase $\vdash B \rightarrow^Y C : \bullet$. We have $\text{lin}(W) = \text{false}$ and $\text{tag}(W) = \{W, \emptyset\}$. By the fact that W is closed and linear safe, we have $\mathcal{L}(W) = \emptyset$. Our goal follows from $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(V) = \mathcal{L}(\hat{N}) \cup \emptyset$.

subcase $\vdash B \rightarrow^Y C : \circ$. We have $\text{lin}(W) = \text{true}$ and $\text{tag}(W) = \{W^\circ, \{W^\circ\}\}$. By Lemma A.3, r appears in N exactly once. We have $\mathcal{L}(\hat{M}) \cup \{W^\circ\} = \mathcal{L}(V) \cup \mathcal{L}(\mathcal{E}) \cup \{W^\circ\} = \mathcal{L}(V) \cup \mathcal{L}(W^\circ) = \mathcal{L}(\hat{N})$.

Otherwise, we have $V^\circ = (V^\circ, \{V^\circ\})$. By a case analysis on the linearity of $B \rightarrow^Y C$.

subcase $\vdash B \rightarrow^Y C : \bullet$. We have $\text{lin}(W) = \text{false}$ and $\text{tag}(W) = \{W, \emptyset\}$. By the fact that W is closed and linear safe, we have $\mathcal{L}(W) = \emptyset$. Our goal follows from $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(V) \cup \{V^\circ\} = \mathcal{L}(V^\circ) = \mathcal{L}(\hat{N}) \cup \emptyset$.

subcase $\vdash B \rightarrow^Y C : \circ$. We have $\text{lin}(W) = \text{true}$ and $\text{tag}(W) = \{W^\circ, \{W^\circ\}\}$. By Lemma A.3, r appears in N exactly once. We have $\mathcal{L}(\hat{M}) \cup \{W^\circ, V^\circ\} = \mathcal{L}(V) \cup \mathcal{L}(\mathcal{E}) \cup \{W^\circ, V^\circ\} = \mathcal{L}(V^\circ) \cup \mathcal{L}(W^\circ) = \mathcal{L}(\hat{N})$.

Case

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

The linear safety of \hat{N} directly follows from the linear safety of \hat{M} . We have $\mathcal{L}(\hat{M}) \cup \emptyset = \mathcal{L}(\mathcal{F}) \cup \mathcal{L}(V^\circ) = \mathcal{L}(\mathcal{F}) \cup \mathcal{L}(V) \cup \{V^\circ\} = \mathcal{L}(\hat{N}) \cup \{V^\circ\}$.

Case

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

The linear safety of \hat{M} gives the linear safety of \mathcal{E} and M . By IH, we have the linear safety of N . The linear safety of \hat{N} follows from the linear safety of \mathcal{E} and N . By IH, we have $\mathcal{L}(M) \cup \mathcal{S} = \mathcal{L}(N) \cup \mathcal{T}$. Our goal follows from $\mathcal{L}(\hat{M}) \cup \mathcal{S} = \mathcal{L}(\mathcal{E}) \cup \mathcal{L}(M) \cup \mathcal{S} = \mathcal{L}(\mathcal{E}) \cup \mathcal{L}(N) \cup \mathcal{T} = \mathcal{L}(\hat{N}) \cup \mathcal{T}$.

□

B FULL SPECIFICATION OF $\mathcal{Q}_{\text{eff}}^\circ$

In this section, we give the full syntax, typing rules, type inference, and constraint solving algorithm of $\mathcal{Q}_{\text{eff}}^\circ$ in Section 5.

B.1 Full Syntax

The full syntax of $\mathcal{Q}_{\text{eff}}^\circ$ is given in Figure 13. Note that we introduce the syntactic category of concrete rows to simplify the presentation of the constraint solving algorithm.

B.2 Full Typing Rules

The full syntax-directed typing rules for $\mathcal{Q}_{\text{eff}}^\circ$ is given in Figure 14. Note that in the qualified effect system of $\mathcal{Q}_{\text{eff}}^\circ$, we only have subtyping between row types and use them in Q-DO, Q-SEQ, Q-HANDLE, and Q-HANDLER. This is different from other type systems with general subtyping, where the subtyping relation is used everywhere. For example, in the Q-APP rule, we require the argument type to be equal to the parameter type of the function, instead of requiring a subtyping

Value types	$A, B ::= \alpha \mid A \rightarrow^Y C$
Computation types	$C, D ::= A! E$
Handler types	$F ::= C \Rightarrow D$
Effect types	$E ::= \{R\}$
Concrete row types	$\text{CRow} \ni K ::= \cdot \mid \ell : A \rightarrow^Y B; K$
Row types	$\text{Row} \ni R ::= \mu \mid K \mid K; R$
Linearity types	$Y ::= \emptyset \mid \bullet \mid \circ$
Types	$\tau ::= A \mid R \mid Y$
Predicates	$\text{Pred} \ni \pi ::= \tau_1 \leq \tau_2 \mid R_1 \leq R_2 \mid R \perp \mathcal{L}$
Qualified types	$\rho ::= A \mid \pi \Rightarrow \rho$
Type schemes	$\text{TySch} \ni \sigma ::= \rho \mid \forall \alpha. \sigma$
Label sets	$\mathcal{L} ::= \emptyset \mid \{\ell\} \uplus \mathcal{L}$
Type contexts	$\text{Env} \ni \Gamma ::= \cdot \mid \Gamma, x : \sigma$
Predicate sets	$\text{PSet} \ni P ::= \cdot \mid P, \pi$
Values	$V, W ::= x \mid \lambda x. M$
Computations	$M, N ::= V W \mid \mathbf{return} V \mid \mathbf{do} \ell V \mid \mathbf{let} x = V \mathbf{in} M$ $\mid \mathbf{let} x \leftarrow M \mathbf{in} N \mid \mathbf{handle} M \mathbf{with} H$
Handlers	$H ::= \{\mathbf{return} x \mapsto M\} \mid \{\ell p r \mapsto M\} \uplus H$

Fig. 13. The syntax of $\mathcal{Q}_{\text{eff}}^\circ$.

relation. Having a full subtyping relation between any types does not help improve the accuracy of tracking control flow linearity; subtyping between effect rows is enough.

B.3 Type Inference Algorithm

The full type inference of $\mathcal{Q}_{\text{eff}}^\circ$ is given in Figure 16. It uses the unification relations $\tau \sim \tau' : \theta$ which states that θ is the principal unifier of types τ and τ' , and $C \sim C' : \theta$ which states that θ is the principal unifier for computation types C and C' . The unification relations are directly defined by the unification function.

$$\frac{\text{U-TYPE}}{\text{unify}(\tau \sim \tau') = \theta}{\tau \sim \tau' : \theta} \qquad \frac{\text{U-COMP}}{\text{unify}(C \sim C') = \theta}{C \sim C' : \theta}$$

Figure 15 gives unification function $\text{unify}(U)$ which takes a set of unification predicates and returns the principal unifiers of them. It is relatively standard [Martelli and Montanari 1982]. The arrow \rightarrow indicates a meta function that might fail. Following Leijen [2008] we explicitly indicate the successful return of a result by return . The auxiliary functions urow and ulin are given and explained in . The unification predicates and predicate sets are defined as follows.

$$\begin{array}{ll} \text{Unification predicates} & \text{UPred} \ni u ::= \tau \sim \tau' \mid C \sim C' \\ \text{Unification sets} & \text{USet} \ni U ::= U, u \end{array}$$

Note that it is possible to postpone the solving of unification constraints to the constraint solving algorithm. We opt for this mixed style presentation for $\mathcal{Q}_{\text{eff}}^\circ$ in order to keep close to the original presentation of qualified types [Jones 1994], and to keep the constraint set cleaner.

$P \mid \Gamma \vdash V : A$	$P \mid \Gamma \vdash M : C$	$P \mid \Gamma \vdash H : C \Rightarrow D$
$\frac{\text{Q-VAR} \quad \begin{array}{c} P \vdash \Gamma \leq \bullet \\ (P \Rightarrow A) \sqsubseteq \sigma \end{array}}{P \mid \Gamma, x : \sigma \vdash x : A}$	$\frac{\text{Q-ABS} \quad \begin{array}{c} P \mid \Gamma, x : A \vdash M : C \\ P \vdash \Gamma \leq Y \end{array}}{P \mid \Gamma \vdash \lambda x. M : A \rightarrow^Y C}$	$\frac{\text{Q-APP} \quad \begin{array}{c} P \mid \Gamma_1, \Gamma \vdash V : A \rightarrow^Y C \\ P \mid \Gamma_2, \Gamma \vdash W : A \quad P \vdash \Gamma \leq \bullet \end{array}}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash V W : C}$
$\frac{\text{Q-LET} \quad \begin{array}{c} Q \mid \Gamma_1, \Gamma \vdash V : A \quad \sigma = \text{gen}((\Gamma_1, \Gamma), Q \Rightarrow A) \\ P \mid \Gamma_2, \Gamma, x : \sigma \vdash M : C \quad P \vdash \Gamma \leq \bullet \end{array}}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{let} \ x = V \ \mathbf{in} \ M : C}$	$\frac{\text{Q-RETURN} \quad P \mid \Gamma \vdash V : A}{P \mid \Gamma \vdash \mathbf{return} \ V : A! \{R\}}$	
$\frac{\text{Q-DO} \quad \begin{array}{c} P \mid \Gamma \vdash V : A_\ell \\ P \vdash \{\ell : A_\ell \rightarrow^Y B_\ell\} \leq R \end{array}}{P \mid \Gamma \vdash \mathbf{do} \ \ell \ V : B_\ell! \{R\}}$	$\frac{\text{Q-SEQ} \quad \begin{array}{c} P \mid \Gamma_1, \Gamma \vdash M : A! \{R_1\} \quad P \mid \Gamma_2, \Gamma, x : A \vdash N : B! \{R_2\} \\ P \vdash R_1 \leq R \quad P \vdash R_2 \leq R \quad P \vdash \Gamma_2 \leq R_1 \quad P \vdash \Gamma \leq \bullet \end{array}}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B! \{R\}}$	
$\frac{\text{Q-HANDLE} \quad \begin{array}{c} P \mid \Gamma_1, \Gamma \vdash H : A! \{R_1\} \Rightarrow D \\ P \mid \Gamma_2, \Gamma \vdash M : A! \{R\} \\ P \vdash \Gamma \leq \bullet \quad P \vdash R \leq R_1 \end{array}}{P \mid \Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{handle} \ M \ \mathbf{with} \ H : D}$	$\frac{\text{Q-HANDLER} \quad \begin{array}{c} H = \{\mathbf{return} \ x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_i \\ C = A! \{(\ell_i : A_i \rightarrow^{Y_i} B_i)_i; R_1\} \quad D = B! \{R_2\} \\ P \mid \Gamma, x : A \vdash M : D \\ [P \mid \Gamma, p_i : A_i, r_i : B_i \rightarrow^{Y_i} D + N_i : D]_i \\ P \vdash \Gamma \leq \bullet \quad P \vdash R_1 \leq R_2 \quad P \vdash R_1 \perp \{\ell_i\}_i \end{array}}{P \mid \Gamma \vdash H : C \Rightarrow D}$	

Fig. 14. Syntax-directed Typing Rules for Q_{eff}°

B.4 Constraint Solving Algorithm

The constraint solving algorithm of Q_{eff}° is given in Figure 17.

The function `ulin` unifies two linearity types. The function `ulab` unifies the signatures of shared labels of two concrete rows. The function `urow` wraps `ulab`. The function `trlin` computes the transitive closure of linearity constraints.

The function `srow`(θ, P, Q) solves row constraints. It takes the current substitution θ and the currently solved predicate set P , and solves the predicates in Q . The basic idea is to transform the row subtyping predicates to forms of $\mu \leq R$ and row lacking predicates to forms of $\mu \perp \mathcal{L}$, which we call *solved forms*. It does a case analysis on the first predicate in Q . For instance, consider the most complicated case $K_1 ; \mu_1 \leq K_2 ; \mu_2$. It first unifies the common labels of K_1 and K_2 . When K_1 is a subset of K_2 , we can directly transform it to the solved form; otherwise, we allocate a fresh row variable to substitute μ_2 and transform it to the solved form. Note that we also need to move all previously solved predicates to the unsolved predicate set, because the row variable μ_2 is substituted, which might turn some predicates in solved forms to unsolved forms.

The main function `solve` sequentially solves row constraints using `srow` and linearity constraints using `trlin`. Note that we use `factorise` to factorise the output predicate set to transform the linearity constraints into the simplest form (i.e., only between value type variables, row variables, and linearity), which is suitable for computing the transitive closure using `trlin`.

```

unify : USet  $\rightarrow$  Subst
unify( $\cdot$ ) = return  $\iota$ 
unify( $\alpha \sim \alpha, U$ ) = unify( $U$ )
unify( $\alpha \sim \tau, U$ ) =
  assert  $\alpha \notin \text{ftv}(\tau)$ 
  let  $\theta = [\tau/\alpha]$ 
  unify( $\theta U$ ) $\theta$ 
unify( $\tau \sim \alpha, U$ ) =
  unify( $\alpha \sim \tau, U$ )
unify( $A! \{R\} \sim A'! \{R'\}, U$ ) =
  unify( $A \sim A', R \sim R', U$ )
unify( $(A \rightarrow^Y C) \sim (A' \rightarrow^{Y'} C'), U$ ) =
  unify( $A \sim A', C \sim C', Y \sim Y', U$ )
unify( $Y \sim Y', U$ ) =
  let  $\theta = \text{ulin}(Y, Y')$ 
  unify( $\theta U$ ) $\theta$ 
unify( $K_1 \sim K_2, U$ ) =
  let  $(K'_1, K'_2, \theta) = \text{urow}(K_1, K_2)$ 
  assert  $\text{set}(K'_1) = \text{set}(K'_2)$ 
  unify( $\theta U$ ) $\theta$ 
unify( $K_1; \mu_1 \sim K_2, U$ ) =
  let  $(K'_1, K'_2, \theta) = \text{urow}(K_1, K_2)$ 
  assert  $\text{set}(K'_1) \subseteq \text{set}(K_2)$ 
  assume fresh  $\mu$ 
  let  $\theta' = [((K'_2 \setminus K'_1); \mu) / \mu_1]$ 
  unify( $\theta' \theta U$ ) $\theta' \theta$ 
unify( $K_2 \sim K_1; \mu_1, U$ ) =
  unify( $K_1; \mu_1 \sim K_2, U$ )
unify( $K_1; \mu_1 \sim K_2; \mu_2, U$ ) =
  let  $(K'_1, K'_2, \theta) = \text{urow}(K_1, K_2)$ 
  assume fresh  $\mu$ 
  let  $\theta' = [((K'_2 \setminus K'_1); \mu) / \mu_1,$ 
              $((K'_1 \setminus K'_2); \mu) / \mu_2]$ 
  unify( $\theta' \theta U$ ) $\theta' \theta$ 

```

Fig. 15. Unification of $\mathcal{Q}_{\text{eff}}^\circ$.

$\Gamma \vdash V : A \dashv \theta, P, \Sigma$	$\Gamma \vdash M : C \dashv \theta, P, \Sigma$	$\Gamma \vdash H : C \Rightarrow D \dashv \theta, P, \Sigma$
--	--	--

<p>Q-VAR^W</p> $\frac{\begin{array}{l} (x : \forall \bar{\alpha}. P \Rightarrow A) \in \Gamma \\ \bar{\beta} \text{ fresh} \quad \theta = [\bar{\beta}/\bar{\alpha}] \end{array}}{\Gamma \vdash x : \theta A \dashv \theta, \theta P, \{x\}}$	<p>Q-LET^W</p> $\frac{\begin{array}{l} \Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \quad \sigma = \text{gen}(\theta_1 \Gamma, P_1 \Rightarrow A) \\ \theta_1 \Gamma, x : \sigma \vdash M : C \dashv \theta_2, P_2, \Sigma_2 \\ Q = \text{un}(\theta_2 \theta_1 \Gamma _{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : \sigma) _{\Sigma_2^c}) \end{array}}{\Gamma \vdash \mathbf{let} \ x = V \ \mathbf{in} \ M : C \dashv \theta_2 \theta_1, P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)}$
<p>Q-ABS^W</p> $\frac{\begin{array}{l} \alpha, \phi \text{ fresh} \quad \Gamma, x : \alpha \vdash M : C \dashv \theta, P, \Sigma \\ Q = \text{leq}(\theta \Gamma _{\Sigma}, \phi) \cup \text{un}(\theta(x : \alpha) _{\Sigma^c}) \end{array}}{\Gamma \vdash \lambda x. M : \theta \alpha \rightarrow^\phi C \dashv \theta, P \cup Q, \Sigma \setminus x}$	<p>Q-APP^W</p> $\frac{\begin{array}{l} \Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \quad \theta_1 \Gamma \vdash W : B \dashv \theta_2, P_2, \Sigma_2 \\ \alpha, \mu, \phi \text{ fresh} \quad \theta_2 A \sim (B \rightarrow^\phi \alpha! \mu) : \theta_3 \\ P = \theta_3(\theta_2 P_1 \cup P_2) \quad Q = \text{un}(\theta_3 \theta_2 \theta_1 \Gamma _{\Sigma_1 \cap \Sigma_2}) \end{array}}{\Gamma \vdash V W : \theta_3(\alpha! \mu) \dashv \theta_3 \theta_2 \theta_1, P \cup Q, \Sigma_1 \cup \Sigma_2}$
<p>Q-SEQ^W</p> $\frac{\begin{array}{l} \Gamma \vdash M : A! \{R_1\} \dashv \theta_1, P_1, \Sigma_1 \quad \theta_1 \Gamma, x : A \vdash N : B! \{R_2\} \dashv \theta_2, P_2, \Sigma_2 \quad \mu \text{ fresh} \\ Q = \text{un}(\theta_2 \theta_1 \Gamma _{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : A) _{\Sigma_2^c}) \cup \text{leq}(\theta_2 \theta_1 \Gamma _{\Sigma_2}, \theta_2 R_1) \cup \text{sub}(\theta_2 R_1, \mu) \cup \text{sub}(R_2, \mu) \end{array}}{\Gamma \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B! \mu \dashv \theta_2 \theta_1, \theta_2 P_1 \cup P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)}$	<p>Q-DO^W</p> $\frac{\begin{array}{l} \Gamma \vdash V : A \dashv \theta_1, P, \Sigma \quad A \sim A_\ell : \theta_2 \\ \mu, \phi \text{ fresh} \quad Q = \text{sub}((\ell : A_\ell \rightarrow^\phi B_\ell), \mu) \end{array}}{\Gamma \vdash \mathbf{do} \ \ell \ V : B_\ell! \{\mu\} \dashv \theta_2 \theta_1, \theta_2 P \cup Q, \Sigma}$
<p>Q-RETURN^W</p> $\frac{\Gamma \vdash V : A \dashv \theta, P, \Sigma \quad \mu \text{ fresh}}{\Gamma \vdash \mathbf{return} \ V : A! \{\mu\} \dashv \theta, P, \Sigma}$	<p>Q-HANDLE^W</p> $\frac{\begin{array}{l} \Gamma \vdash H : A! \{R_1\} \Rightarrow D \dashv \theta_1, P_1, \Sigma_1 \quad \theta_1 \Gamma \vdash M : A'! \{R\} \dashv \theta_2, P_2, \Sigma_2 \\ \theta_2 A \sim A' : \theta_3 \quad P = \theta_3(\theta_2 P_1 \cup P_2) \quad Q = \text{sub}(\theta_3 R, \theta_3 \theta_2 R_1) \cup \text{un}(\theta_3 \theta_2 \theta_1 \Gamma _{\Sigma_1 \cap \Sigma_2}) \end{array}}{\Gamma \vdash \mathbf{handle} \ M \ \mathbf{with} \ H : \theta_3 \theta_2 D \dashv \theta_3 \theta_2 \theta_1, P \cup Q, \Sigma_1 \cup \Sigma_2}$
<p>Q-HANDLER^W</p> $\frac{\begin{array}{l} \alpha, \phi_i, \mu \text{ fresh} \quad \Gamma, x : \alpha \vdash M : D \dashv \theta_0, P_0, \Sigma_0 \\ [\theta_{i-1}(\Gamma, p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D) \vdash N_i : D_i \dashv \theta'_i, P_i, \Sigma_i \quad D_i \sim \theta'_i \theta_{i-1} D : \theta''_i \quad \theta_i = \theta''_i \theta'_i \theta_{i-1}]_{i=1}^n \\ C = \theta_n(\alpha! \{(\ell_i : A_{\ell_i} \rightarrow^{\phi_i} B_{\ell_i})_i; \mu\}) \quad B! \{R\} = \theta_n D \\ \Sigma = (\Sigma_0 \setminus \{x\}) \cup (\cup_{i=1}^n (\Sigma_i \setminus \{p_i, r_i\})) \quad P = (\cup_{i=0}^n \theta_n P_i) \cup \text{un}(\theta_n \Gamma _{\Sigma}) \cup \text{sub}(\mu, R) \cup \text{lack}(\mu, \{\ell_i\}_i) \\ Q = \text{un}(\theta_n(x : \alpha) _{\Sigma_0^c}) \cup (\cup_{i=1}^n \text{un}(\theta_n(p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D))) \end{array}}{\Gamma \vdash \{\mathbf{return} \ x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_{i=1}^n : C \Rightarrow \theta_n D \dashv \theta_n, P \cup Q, \Sigma}$	<p>$\text{leq}(\Gamma, \tau) = \text{factorise}(\Gamma \leq \tau) \quad \text{sub}(R_1, R_2) = \text{factorise}(R_1 \leq R_2)$ $\text{un}(\Gamma) = \text{leq}(\Gamma, \bullet) \quad \text{lack}(R, \mathcal{L}) = \text{factorise}(R \perp \mathcal{L})$</p>

Fig. 16. Type inference of $\mathcal{Q}_{\text{eff}}^{\circ}$

```

srow : (Subst × PSet × PSet)
      → (Subst × PSet)
srow(θ, P, ·) = return (θ, P)
srow(θ, P, (τ1 ≤ τ2, Q)) =
  srow(θ, (P, τ1 ≤ τ2), Q)
srow(θ, P, (K1 ≤ K2, Q)) =
  let (K'1, K'2, θ') = urow(K1, K2)
  assert set(K'1) ⊆ set(K'2)
  srow(θ'θ, θ'P, θ'Q)
srow(θ, P, (K1 ; μ ≤ K2 ; μ, Q)) =
  let (K'1, K'2, θ') = urow(K1, K2)
  assert set(K'1) ⊆ set(K'2)
  srow(θ'θ, θ'P, θ'Q)
srow(θ, P, (K1 ; μ ≤ K2, Q)) =
  let (K'1, K'2, θ') = urow(K1, K2)
  assert set(K'1) ⊆ set(K'2)
  srow(θ'θ, (θ'P, μ ≤ (K'2 \ K'1)), θ'Q)
srow(θ, P, (K1 ≤ K2 ; μ2, Q)) =
  let (K'1, K'2, θ') = urow(K1, K2)
  if set(K'1) ⊆ set(K'2)
  then srow(θ'θ, θ'P, θ'Q)
  else assume fresh μ
       let θ'' = [((K'1 \ K'2) ; μ) / μ2]θ'
       srow(θ''θ, ·, θ''(Q, P))
srow(θ, P, (K1 ; μ1 ≤ K2 ; μ2, Q)) =
  let (K'1, K'2, θ') = urow(K1, K2)
  if set(K'1) ⊆ set(K'2)
  then srow(θ'θ, (θ'P, μ1 ≤ (K'2 \ K'1) ; μ2), θ'Q)
  else assume fresh μ
       let θ'' = [((K'1 \ K'2) ; μ) / μ2]θ'
       srow(θ''θ, μ1 ≤ (K'2 \ K'1) ; μ, θ''(Q, P))
srow(θ, P, (K ⊥ L, Q)) =
  assert dom(K) ∩ L = ∅
  srow(θ, P, Q)
srow(θ, P, (K ; μ ⊥ L, Q)) =
  assert dom(K) ∩ L = ∅
  srow(θ, (P, μ ⊥ L), Q)

urow : (CRow × CRow)
      → (CRow × CRow × Subst)
urow(K, K') =
  let θ = ulab(K, K')
  return (θK, θK', θ)

ulab : (CRow × CRow) → Subst
ulab(·, K) = return ι
ulab(K, ·) = return ι
ulab((ℓ : Y1 ; K1), (ℓ : Y2 ; K2)) =
  let θ = ulin(Y1, Y2)
  let θ' = ulab(θK1, θK2)
  return θ'θ
ulab((ℓ : Y ; K1), K2) = ulab(K1, K2)
ulab(K1, (ℓ : Y ; K2)) = ulab(K1, K2)

ulin : (Lin × Lin) → Subst
ulin(Y, Y) = return ι
ulin(•, ◦) = fail
ulin(◦, •) = fail
ulin(φ, Y) = return [Y/φ]
ulin(Y, φ) = return [Y/φ]

trlin : (PSet × PSet) → PSet
trlin(P, ·) = P
trlin(P, (R1 ≤ R2, Q)) = trlin(P, Q)
trlin(P, (τ1 ≤ τ2, Q)) = trlin(P ∪ P'', Q)
  where
    P' = P ∪ {τ1 ≤ τ1, τ2 ≤ τ2}
    P'' = {τ'1 ≤ τ'2 | {τ'1 ≤ τ1, τ2 ≤ τ'2} ⊆ P'}

solve : PSet → (Subst × PSet)
solve(P) =
  let (θ, Q) = srow(ι, ·, P)
  let Q' = trlin(·, factorise(Q))
  assert (◦ ≤ •) ∉ Q'
  return (θ, Q)

```

Fig. 17. Constraint solving of Q_{eff}° .

C PROOFS OF $\mathcal{Q}_{\text{eff}}^\circ$

In this section, we prove the theorems in Section 5.

C.1 Correctness of Factorisation

We first prove some useful properties of the entailment relations.

THEOREM C.1 (PROPERTIES OF THE ENTAILMENT RELATION). *The entailment relation between predicate sets satisfies the following properties:*

- *Monotonicity.* If $Q \subseteq P$, then $P \vdash Q$.
- *Transitivity.* If $P_1 \vdash P_2$ and $P_2 \vdash P_3$, then $P_1 \vdash P_3$.
- *Closure property.* If $P \vdash Q$, then $\theta P \vdash \theta Q$.
- *Weakening.* If $P \vdash Q$, then $P, P' \vdash Q$.

PROOF.

Monotonicity. Directly follows from P-SUBSUME and P-PREDSET.

Transitivity. By P-PREDSET, we only need to prove that if $P_1 \vdash P_2$ and $P_2 \vdash \pi$, then $P_1 \vdash \pi$. By straightforward induction on $P_2 \vdash \pi$.

Closure property. By P-PREDSET, we only need to prove that if $P \vdash \pi$ then $\theta P \vdash \theta \pi$. By straightforward induction on $P \vdash \pi$.

Weakening. By P-PREDSET, we only need to prove that if $P \vdash \pi$ then $P, P' \vdash \pi$. By straightforward induction on $P \vdash \pi$.

□

LEMMA C.2 (INVERSE CLOSURE PROPERTY). *If $P \vdash \theta(\sigma \leq \tau)$, then there exists $P' \vdash \sigma \leq \tau$ such that $P \vdash \theta P'$.*

PROOF. By induction on the entailment relations.

Case

$$\frac{\text{P-QUANTIFIER} \quad P \vdash [\tau'/\alpha]\theta(\sigma \leq \tau) \text{ (1) for some } \tau'}{P \vdash \theta((\forall \alpha.\sigma) \leq \tau)}$$

Assume that $\alpha \notin \text{dom}(\theta)$ and $\alpha \notin \text{ftv}(\tau)$ without loss of generality. We can commute $[\tau'/\alpha]$ and θ in (1). By the IH on (1), there exists $P' \vdash [\tau'/\alpha](\sigma \leq \tau)$ such that $P \vdash \theta P'$. By P-QUANTIFIER, we have $P' \vdash (\forall \alpha.\sigma) \leq \tau$.

Case

$$\frac{\text{P-QUALIFIER} \quad P \vdash \theta\pi \text{ (1)} \quad P \vdash \theta(\rho \leq \tau) \text{ (2)}}{P \vdash \theta((\pi \Rightarrow \rho) \leq \tau)}$$

By the IH on (1), there exists $P_1 \vdash \pi$ such that $P \vdash \theta P_1$. By the IH on (2), there exists $P_2 \vdash \rho \leq \tau$ such that $P \vdash \theta P_2$. By P-QUALIFIER, we have $P_1 \cup P_2 \vdash (\pi \Rightarrow \rho) \leq \tau$. By P-PREDSET, we have $P \vdash \theta(P_1 \cup P_2)$.

Case For all other cases of $P \vdash \theta\pi$, just take $P' = \pi$.

□

THEOREM 5.3 (CORRECTNESS OF FACTORISATION). *If $\text{factorise}(P) = Q$, then $Q \vdash P$ and $P \vdash Q$. If $\text{factorise}(\Gamma \leq \tau) = Q$, then $Q \vdash \Gamma \leq \tau$ and for any $P \vdash \Gamma \leq \tau$, there exists θ such that $P \vdash \theta Q$.*

PROOF. The first part of the theorem is kind of obvious because $\text{factorise}(P)$ is almost directly defined from the entailment rules in Figure 10. We prove the auxiliary lemma that if $\text{factorise}(\pi) =$

Q , then $Q \vdash \pi$ and $\pi \vdash Q$. Both directions follow from straightforward induction on the definition of `factorise`. Note that in the proof of $\pi \vdash Q$, we apply the bottom-up direction of the two-way rules `P-FUN` and `P-ROW`. Then, given $\text{factorise}(P) = \bigcup_{\pi \in P} \text{factorise}(\pi)$, by the lemma we have $\text{factorise}(\pi) \vdash \pi$ for all $\pi \in P$, which then give $\bigcup_{\pi \in P} \text{factorise}(\pi) \vdash P$ by `P-PREDSET` and the *weakening* of Theorem C.1. We also have that $\pi \vdash \text{factorise}(\pi)$ for all $\pi \in P$, which then give $P \vdash \bigcup_{\pi \in P} \text{factorise}(\pi)$ by `P-PREDSET` and the *weakening* of Theorem C.1.

For the second part of the theorem, we prove the auxiliary lemma that if $\text{factorise}(\sigma \leq \tau) = Q$, then $Q \vdash \sigma \leq \tau$ and for any $P \vdash \sigma \leq \tau$, there exists θ such that $P \vdash \theta Q$. The $Q \vdash \sigma \leq \tau$ follows from straightforward induction on the definition of `factorise`. The other direction is more involved. We proceed by induction on the definition of `factorise`.

Case

$$\text{factorise}((\forall \alpha. \sigma) \leq \tau) = \text{factorise}([\beta/\alpha]\sigma \leq \tau) \text{ (1) for some fresh } \beta$$

Suppose $\text{factorise}((\forall \alpha. \sigma) \leq \tau) = Q$. We want to show that for any $P \vdash (\forall \alpha. \sigma) \leq \tau$ (2), there exists θ such that $P \vdash \theta Q$. By (2) and `P-QUANTIFIER`, there exists $\theta_1 = [\tau'/\alpha]$ such that $P \vdash \theta_1 \sigma \leq \tau$. Let $\theta_2 = [\tau'/\beta]$. We have $P \vdash \theta_2 [\beta/\alpha]\sigma \leq \tau$. By Lemma C.2, there exists P' such that $P' \vdash [\beta/\alpha]\sigma \leq \tau$ (3) and $P \vdash \theta_2 P'$. By (3) and the IH on (1), there exists θ_3 such that $P' \vdash \theta_3 Q$. Then, by the closure property and transitivity of Theorem C.1, we have $P \vdash \theta_2 \theta_3 Q$.

Case

$$\text{factorise}((\pi \Rightarrow \sigma) \leq \tau) = \text{factorise}(\pi) \text{ (1)} \cup \text{factorise}(\sigma \leq \tau) \text{ (2)}$$

Suppose $\text{factorise}(\pi) = Q_1$ and $\text{factorise}(\sigma \leq \tau) = Q_2$. For any $P \vdash (\pi \Rightarrow \sigma) \leq \tau$, by `P-QUALIFIER`, we have $P \vdash \pi$ and $P \vdash \sigma \leq \tau$. By the IH on (1), there exists θ_1 such that $P \vdash \theta_1 Q_1$. By the IH on (2), there exists θ_2 such that $P \vdash \theta_2 Q_2$. Note that $\text{dom}(\theta_1) \cap \text{dom}(\theta_2) = \emptyset$. Thus, we have $P \vdash \theta_1 \theta_2 (Q_1 \cup Q_2)$.

Case

$$\text{factorise}(\pi) = Q$$

By the first part of the theorem which has been proved, we have $\pi \vdash Q$. For any $P \vdash \pi$, by the transitivity of Theorem C.1, we have $P \vdash Q$.

With this lemma, our goal follows from a similar analysis to the proof of the first part of the theorem since `P-CONTEXT` and `P-PREDSET` are both conjunction rules. \square

C.2 Principal Unifier

We have the following lemmas for the unification function in Figure 15 and its auxiliary functions.

LEMMA C.3 (PRINCIPAL AUXILIARY UNIFIERS). *Given K_1 and K_2 , let $K'_1 = (K_1|_{\text{dom}(K_1) \cap \text{dom}(K_2)})$ and $K'_2 = (K_2|_{\text{dom}(K_1) \cap \text{dom}(K_2)})$. If $\text{ulab}(K_1, K_2) = \theta$, then for any $\theta' K'_1 = \theta' K'_2$, there exists θ'' such that $\theta' = \theta'' \theta$; if it fails, then K'_1 and K'_2 cannot be unified.*

PROOF. By straightforward induction on the definition of `urow`, `ulab` and `ulin`. \square

LEMMA C.4 (PRINCIPAL UNIFIERS). *If $A \sim B : \theta$, then for any $\theta' A = \theta' B$, there exists θ'' such that $\theta' = \theta'' \theta$; if it fails, then A and B cannot be unified. The same applies to computation types.*

PROOF. By straightforward induction on the definition of `unify(U)`. \square

C.3 Soundness and Completeness of Type Inference

We prove the soundness and completeness of type inference as well as auxiliary lemmas.

LEMMA C.5 (CLOSURE PROPERTY OF TYPING). *If $P \mid \Gamma \vdash V : A$, then $\theta P \mid \theta\Gamma \vdash V : \theta A$. The same applies to computation and handler typing.*

PROOF. By the closure property of Theorem C.1 and straightforward induction on the typing derivations. \square

LEMMA C.6 (WEAKENING OF PREDICATES). *If $P \mid \Gamma \vdash V : A$, then $P, P' \mid \Gamma \vdash V : A$. The same applies to computation and handler typing.*

PROOF. By the weakening property of Theorem C.1 and straightforward induction on the typing derivations. \square

LEMMA C.7 (EXTRA IS UNLIMITED). *If $P \mid \Gamma \vdash V : A$, then $P' \mid \Gamma, x : \sigma \vdash V : A$ for any $P' \vdash P$ and $P' \vdash \sigma \leq \bullet$. The same applies to computation and handler typing.*

PROOF. By straightforward induction on the typing derivations. \square

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

PROOF. By mutual induction on the type inference derivations $\Gamma \vdash V : A \dashv \theta, P, \Sigma$, $\Gamma \vdash M : C \dashv \theta, P, \Sigma$, and $\Gamma \vdash H : C \Rightarrow D \dashv \theta, P, \Sigma$.

Case

$$\frac{\text{Q-VAR}^W \quad (x : \forall \bar{\alpha}. P \Rightarrow A) \in \Gamma \quad \bar{\beta} \text{ fresh} \quad \theta = [\bar{\beta}/\bar{\alpha}] \quad (1)}{\Gamma \vdash x : \theta A \dashv \theta, \theta P, \{x\}}$$

By (1), we have $\theta P \Rightarrow \theta A \sqsubseteq \theta(\forall \bar{\alpha}. P \Rightarrow A)$. Our goal then follows from Q-VAR.

Case

$$\frac{\text{Q-LET}^W \quad \begin{array}{l} \Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \quad (1) \quad \sigma = \text{gen}(\theta_1\Gamma, P_1 \Rightarrow A) \\ \theta_1\Gamma, x : \sigma \vdash M : C \dashv \theta_2, P_2, \Sigma_2 \quad (2) \\ Q = \text{un}(\theta_2\theta_1\Gamma|_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : \sigma)|_{\Sigma_2^c}) \end{array}}{\Gamma \vdash \mathbf{let} \ x = V \ \mathbf{in} \ M : C \dashv \theta_2\theta_1, P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)}$$

By the IH on (1), we have $P_1 \mid \theta_1\Gamma|_{\Sigma_1} \vdash V : A$. By Lemma C.5, we have $\theta_2P_1 \mid \theta_2\theta_1\Gamma \vdash V : \theta_2A$ (3). By the IH on (2), we have $P_2 \mid \theta_2(\theta_1\Gamma, x : \sigma)|_{\Sigma_2} \vdash M : \theta_2C$ (4). Let $\sigma' = \text{gen}(\theta_2\theta_1\Gamma, \theta_2P_1 \Rightarrow \theta_2A)$. Notice that θ_2 is generated by the type inference judgement (2), which cannot substitute any variables bound by σ (i.e., variables in $\text{ftv}(P_1 \Rightarrow A) \setminus \text{ftv}(\theta_1\Gamma)$). Thus, we have $\theta_2\sigma = \sigma'$. Let $\Sigma'_2 = \Sigma_2 \setminus x$, $\Gamma_1 = (\theta_2\theta_1\Gamma)|_{\Sigma_1 \setminus \Sigma'_2}$, $\Gamma_2 = (\theta_2\theta_1\Gamma)|_{\Sigma'_2 \setminus \Sigma_1}$, $\Gamma' = (\theta_2\theta_1\Gamma)|_{\Sigma_1 \cap \Sigma'_2}$. By (3) and (4), we have $\theta_2P_1 \mid \Gamma_1, \Gamma' \vdash V : \theta_2A$ (5) and $P_2 \mid \Gamma_2, \Gamma', (x : \sigma')|_{\Sigma_2} \vdash M : \theta_2C$. By Lemma C.7 we have $P_2 \cup \text{un}((x : \sigma')|_{\Sigma_2^c}) \mid \Gamma_2, \Gamma', x : \sigma' \vdash M : \theta_2C$ (6). By Theorem 5.3, we have $Q \vdash \Gamma' \leq \bullet$. Our goal follows from Q-LET, (5), (6) and Lemma C.6.

Case

$$\frac{\text{Q-ABS}^W \quad \begin{array}{l} \alpha, \phi \text{ fresh} \quad \Gamma, x : \alpha \vdash M : C \dashv \theta, P, \Sigma \quad (1) \\ Q = \text{leq}(\theta\Gamma|_{\Sigma}, \phi) \cup \text{un}(\theta(x : \alpha)|_{\Sigma^c}) \end{array}}{\Gamma \vdash \lambda x. M : \theta\alpha \dashv \phi \ C \dashv \theta, P \cup Q, \Sigma \setminus x}$$

By the IH on (1), we have $P \mid (\theta\Gamma, x : \theta\alpha)|_{\Sigma} \vdash M : C$. Our goal follows from Lemma C.7, Theorem 5.3 and Q-ABS.

Case

$$\begin{array}{c} \text{Q-APP}^W \\ \Gamma \vdash V : A \dashv \theta_1, P_1, \Sigma_1 \text{ (1)} \quad \theta_1\Gamma \vdash W : B \dashv \theta_2, P_2, \Sigma_2 \text{ (2)} \\ \alpha, \mu, \phi \text{ fresh} \quad \theta_2 A \sim (B \rightarrow^{\phi} \alpha! \mu) : \theta_3 \text{ (3)} \\ P = \theta_3(\theta_2 P_1 \cup P_2) \quad Q = \text{un}(\theta_3 \theta_2 \theta_1 \Gamma|_{\Sigma_1 \cap \Sigma_2}) \\ \hline \Gamma \vdash V W : \theta_3(\alpha! \mu) \dashv \theta_3 \theta_2 \theta_1, P \cup Q, \Sigma_1 \cup \Sigma_2 \end{array}$$

By the IH on (1), we have $P_1 \mid \theta_1\Gamma \vdash V : A$. By the IH on (2), we have $P_2 \mid \theta_2\theta_1\Gamma \vdash W : B$. By Lemma C.5, we have $\theta_3\theta_2P_1 \mid \theta_3\theta_2\theta_1\Gamma \vdash V : \theta_3\theta_2A$ (4) and $\theta_3P_2 \mid \theta_3\theta_2\theta_1\Gamma \vdash W : \theta_3B$ (5). By (3), we have $\theta_3\theta_2A = \theta_3(B \rightarrow^{\phi} \alpha! \mu)$. Let $\Gamma_1 = (\theta_3\theta_2\theta_1\Gamma)|_{\Sigma_1 \setminus \Sigma_2}$, $\Gamma_2 = (\theta_3\theta_2\theta_1\Gamma)|_{\Sigma_2 \setminus \Sigma_1}$, $\Gamma' = (\theta_3\theta_2\theta_1\Gamma)|_{\Sigma_1 \cap \Sigma_2}$. By (4) and (5), we have $\theta_3\theta_2P_1 \mid \Gamma_1, \Gamma' \vdash V : \theta_3\theta_2A$ (6) and $\theta_3P_2 \mid \Gamma_2, \Gamma' \vdash W : \theta_3B$ (7). By Theorem 5.3, we have $Q \vdash \Gamma' \leq \bullet$. Our goal follows from Q-APP, (6), (7), and Lemma C.6.

Case

$$\begin{array}{c} \text{Q-RETURN}^W \\ \Gamma \vdash V : A \dashv \theta, P, \Sigma \text{ (1)} \quad \mu \text{ fresh} \\ \hline \Gamma \vdash \text{return } V : A! \{\mu\} \dashv \theta, P, \Sigma \end{array}$$

Our goal follows from the IH on (1) and Q-RETURN.

Case

$$\begin{array}{c} \text{Q-DO}^W \\ \Gamma \vdash V : A \dashv \theta_1, P, \Sigma \text{ (1)} \quad A \sim A_{\ell} : \theta_2 \\ \mu, \phi \text{ fresh} \quad Q = \text{sub}((\ell : A_{\ell} \rightarrow^{\phi} B_{\ell}), \mu) \\ \hline \Gamma \vdash \text{do } \ell V : B_{\ell}! \{\mu\} \dashv \theta_2 \theta_1, \theta_2 P \cup Q, \Sigma \end{array}$$

Our goal follows from the IH on (1), Q-DO, Theorem 5.3, and Lemma C.5.

Case

$$\begin{array}{c} \text{Q-SEQ}^W \\ \Gamma \vdash M : A! \{R_1\} \dashv \theta_1, P_1, \Sigma_1 \text{ (1)} \quad \theta_1\Gamma, x : A \dashv N : B! \{R_2\} \dashv \theta_2, P_2, \Sigma_2 \text{ (2)} \quad \mu \text{ fresh} \\ Q = \text{un}(\theta_2\theta_1\Gamma|_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : A)|_{\Sigma_2^c}) \cup \text{leq}(\theta_2\theta_1\Gamma|_{\Sigma_2}, \theta_2R_1) \cup \text{sub}(\theta_2R_1, \mu) \cup \text{sub}(R_2, \mu) \\ \hline \Gamma \vdash \text{let } x \leftarrow M \text{ in } N : B! \mu \dashv \theta_2 \theta_1, \theta_2 P_1 \cup P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x) \end{array}$$

Similar to the Q-LET^W and Q-APP^W cases. Let $\Sigma'_2 = \Sigma_2 \setminus x$, $\Gamma_1 = (\theta_2\theta_1\Gamma)|_{\Sigma_1 \setminus \Sigma'_2}$, $\Gamma_2 = (\theta_2\theta_1\Gamma)|_{\Sigma'_2 \setminus \Sigma_1}$, $\Gamma' = (\theta_2\theta_1\Gamma)|_{\Sigma_1 \cap \Sigma'_2}$. By the IH on (1) and Lemma C.5, we have $\theta_2P_1 \mid \Gamma_1, \Gamma' \vdash M : \theta_2(A! \{R_1\})$ (3). By the IH on (2), we have $P_2 \mid \Gamma_2, \Gamma', (x : A)|_{\Sigma_2} \dashv N : B! \{R_2\}$ (4). Our goal follows from Q-SEQ, (3), (4), Theorem 5.3, Lemma C.6 and Lemma C.7.

Case

$$\begin{array}{c} \text{Q-HANDLE}^W \\ \Gamma \vdash H : A! \{R_1\} \rightrightarrows D \dashv \theta_1, P_1, \Sigma_1 \text{ (1)} \quad \theta_1\Gamma \vdash M : A'! \{R\} \dashv \theta_2, P_2, \Sigma_2 \text{ (2)} \\ \theta_2A \sim A' : \theta_3 \quad P = \theta_3(\theta_2P_1 \cup P_2) \quad Q = \text{sub}(\theta_3R, \theta_3\theta_2R_1) \cup \text{un}(\theta_3\theta_2\theta_1\Gamma|_{\Sigma_1 \cap \Sigma_2}) \\ \hline \Gamma \vdash \text{handle } M \text{ with } H : \theta_3\theta_2D \dashv \theta_3\theta_2\theta_1, P \cup Q, \Sigma_1 \cup \Sigma_2 \end{array}$$

By a similar proof to the Q-APP case, our goal follows from the IHs on (1) and (2), Theorem 5.3, Lemma C.5 and Lemma C.6.

Case

$$\begin{array}{l}
\text{Q-HANDLER}^W \\
\alpha, \phi_i, \mu \text{ fresh} \quad \Gamma, x : \alpha \vdash M : D \dashv \theta_0, P_0, \Sigma_0 \quad (1) \\
[\theta_{i-1}(\Gamma, p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D) \vdash N_i : D_i \dashv \theta'_i, P_i, \Sigma_i \quad (2) \\
D_i \sim \theta'_i \theta_{i-1} D : \theta''_i \quad \theta_i = \theta''_i \theta'_i \theta_{i-1}]_{i=1}^n \\
C = \theta_n(\alpha! \{(\ell_i : A_{\ell_i} \rightarrow^{\phi_i} B_{\ell_i})_i; \mu\}) \quad B! \{R\} = \theta_n D \\
\Sigma = (\Sigma_0 \setminus \{x\}) \cup (\cup_{i=1}^n (\Sigma_i \setminus \{p_i, r_i\})) \\
P = (\cup_{i=0}^n \theta_n P_i) \cup \text{un}(\theta_n \Gamma|_{\Sigma}) \cup \text{sub}(\mu, R) \cup \text{lack}(\mu, \{\ell_i\}_i) \\
Q = \text{un}(\theta_n(x : \alpha)|_{\Sigma_0^c}) \cup (\cup_{i=1}^n \text{un}(\theta_n(p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D))) \\
\hline
\Gamma \vdash \{\text{return } x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_{i=1}^n : C \Rightarrow \theta_n D \dashv \theta_n, P \cup Q, \Sigma
\end{array}$$

The type inference for handlers is the most complicated, but there is nothing really new about the proof compared to previous cases. By the IH on (1), we have $P_0 \mid \theta_0(\Gamma, x : \alpha)|_{\Sigma_0} \vdash M : D$. By the IH on (2), we have $P_i \mid \theta'_i \theta_{i-1}(\Gamma, p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D)|_{\Sigma_i} \vdash N_i : D_i$. By Lemma C.5, we have $\theta_n P_0 \mid \theta_n(\Gamma, x : \alpha)|_{\Sigma_0} \vdash M : \theta_n D$ and $\theta_n P_i \mid \theta_n(\Gamma, p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D)|_{\Sigma_i} \vdash N_i : \theta_n D_i$. By Lemma C.7, we have

$$\theta_n P_0 \cup \text{un}(\theta_n \Gamma|_{\Sigma}) \cup \text{un}(\theta_n(x : \alpha)|_{\Sigma_0^c}) \mid \theta_n(\Gamma|_{\Sigma}, x : \alpha) \vdash M : \theta_n D \quad (3)$$

and

$$\theta_n P_i \cup \text{un}(\theta_n \Gamma|_{\Sigma}) \cup \text{un}(\theta_n(p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D)) \mid \theta_n(\Gamma|_{\Sigma}, p_i : A_{\ell_i}, r_i : B_{\ell_i} \rightarrow^{\phi_i} D) \vdash N_i : \theta_n D_i \quad (4)$$

By Theorem 5.3, we have $P \cup Q \vdash \{\mu \leq R, \mu \perp \{\ell_i\}_i\}$. Our goal follows from Q-HANDLER, (3), (4), and Lemma C.6. \square

LEMMA C.8 (MORE GENERAL CONTEXTS). *If $P \mid \Gamma, x : \sigma \vdash V : A$ and $\sigma \sqsubseteq \sigma'$, then $P \mid \Gamma, x : \sigma' \vdash V : A$. The same applies to computation and handler typing.*

PROOF. By straightforward induction on the typing derivation. \square

LEMMA C.9 (ZERO IS UNLIMITED). *If $P \mid \Gamma, x : \sigma \vdash V : A$ and x does not appear in V , then $P \vdash \sigma \leq \bullet$. The same applies to computation and handler typing.*

PROOF. By straightforward induction on the typing derivation. \square

LEMMA C.10 (CLOSURE PROPERTY OF FACTORISATION). *If $\text{factorise}(P) = Q$, then $\text{factorise}(\theta P) = \theta Q$. If $\text{factorise}(\Gamma \leq \tau) = Q$, then $\text{factorise}(\theta(\Gamma \leq \tau)) = \theta Q$.*

PROOF. By the closure property of Theorem C.1 and straightforward induction on the definition of factorise. \square

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.*

PROOF. By mutual induction on the syntax-directed typing derivations $P \mid \Gamma \vdash V : A$, $P \mid \Gamma \vdash M : C$, and $P \mid \Gamma \vdash H : C \Rightarrow D$.

Case

$$\begin{array}{l}
\text{Q-VAR} \\
P \vdash \Gamma \leq \bullet \quad P \Rightarrow A \sqsubseteq \forall \bar{\alpha}. Q \Rightarrow B \\
\hline
P \mid \theta(\Gamma, x : \forall \bar{\alpha}. Q \Rightarrow B) \vdash x : A
\end{array}$$

By $P \Rightarrow A \sqsubseteq \sqrt{\alpha}.Q \Rightarrow B$, there exists θ_1 such that $A = \theta_1 B$ and $P \vdash \theta_1 Q$. By Q-VAR^W, we have the following derivation

$$\frac{\text{Q-VAR}^W \quad \overline{\beta} \text{ fresh} \quad \theta' = [\overline{\beta}/\overline{\alpha}]}{\Gamma, x : \sqrt{\alpha}.Q \Rightarrow B \vdash x : \theta' B \dashv \theta', \theta' Q, \{x\}}$$

Let $\theta'' = \theta\theta_1[\overline{\alpha}/\overline{\beta}]$, we have $A = \theta_1 B = \theta''\theta' B$, $P \vdash \theta_1 Q = \theta''\theta' Q$, and $(\theta = \theta''\theta')|_{\Gamma}$.

Case

$$\frac{\text{Q-LET} \quad \begin{array}{l} P_1 \mid \theta(\Gamma_1, \Gamma) \vdash V : A \text{ (1)} \quad \sigma = \text{gen}(\theta(\Gamma_1, \Gamma), P_1 \Rightarrow A) \\ P_2 \mid \theta(\Gamma_2, \Gamma), x : \sigma \vdash M : C \text{ (2)} \quad P_2 \vdash \theta\Gamma \leq \bullet \end{array}}{P_2 \mid \theta(\Gamma_1, \Gamma_2, \Gamma) \vdash \text{let } x = V \text{ in } M : C}$$

By the IH on (1), we have $\Gamma_1, \Gamma \vdash V : A' \dashv \theta_1, P'_1, \Sigma_1$ and there exists θ'_1 such that $A = \theta'_1 A'$, $P_1 \vdash \theta'_1 P'_1$, and $(\theta = \theta'_1 \theta_1)|_{\Gamma_1, \Gamma}$. By context weakening, we have $\Gamma_1, \Gamma_2, \Gamma \vdash V : A' \dashv \theta_1, P'_1, \Sigma_1$ (3). We also have $\sigma = \text{gen}(\theta(\Gamma_1, \Gamma_2, \Gamma), P_1 \Rightarrow A)$. Let $\sigma' = \text{gen}(\theta_1(\Gamma_1, \Gamma_2, \Gamma), P'_1 \Rightarrow A')$. By $(\theta = \theta'_1 \theta_1)|_{\Gamma_1, \Gamma}$, it is easy to see that $\sigma \sqsubseteq \theta'_1 \sigma'$. Then by (2) and Lemma C.8, we have $P_2 \mid \theta(\Gamma_2, \Gamma), x : \theta'_1 \sigma' \vdash M : C$, which further implies $P_2 \mid \theta_3 \theta'_1 \theta_1(\Gamma_2, \Gamma, x : \sigma') \vdash M : C$ (4) for some θ_3 with $\theta = \theta_3 \theta'_1 \theta_1$. By the IH on (4), we have $\theta_1(\Gamma_2, \Gamma, x : \sigma') \vdash M : C' \dashv \theta_2, P'_2, \Sigma_2$ and there exists θ'_2 such that $C = \theta'_2 C'$, $P_2 \vdash \theta'_2 P'_2$, and $(\theta_3 \theta'_1 = \theta'_2 \theta_2)|_{\Gamma_2, \Gamma}$. By context weakening and $\theta_1 \sigma' = \sigma'$, we have $\theta_1(\Gamma_1, \Gamma_2, \Gamma), x : \sigma' \vdash M : C' \dashv \theta_2, P'_2, \Sigma_2$ (5). Let $Q = \text{un}(\theta_2 \theta_1(\Gamma_1, \Gamma_2, \Gamma)|_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2(x : \sigma)|_{\Sigma_2^c})$. By Q-LET^W, (3) and (5), we have

$$\Gamma_1, \Gamma_2, \Gamma \vdash \text{let } x = V \text{ in } M : C' \dashv \theta_2 \theta_1, P'_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)$$

With $\theta' = \theta'_2 \theta_2$, we have $(\theta = \theta' \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma_3}$. By $\Sigma_1 \cap \Sigma_2 \subseteq \text{dom}(\Gamma)$, Lemma C.9, Lemma C.10 and Theorem 5.3, there exists θ_p such that $P_2 \vdash \theta_p \theta' Q$ (6). Let $\theta'' = \theta_p \theta'$. Our goal follows from $(\theta = \theta'' \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma_3}$, $C = \theta'' C'$, and $P_2 \vdash \theta''(P'_2 \cup Q)$.

Case

$$\frac{\text{Q-ABS} \quad \begin{array}{l} P \mid \theta\Gamma, x : A \vdash M : C \text{ (1)} \quad P \vdash \theta\Gamma \leq Y \end{array}}{P \mid \theta\Gamma \vdash \lambda x. M : A \rightarrow^Y C}$$

Take a fresh variable α and let $\theta_1 = \theta[A/\alpha]$. By (1), we have $P \mid \theta_1(\Gamma, x : \alpha) \vdash M : C$ (2). By the IH on (2), we have $\Gamma, x : \alpha \vdash M : C' \dashv \theta', P', \Sigma$ (3) and there exists θ'' such that $C = \theta'' C'$, $P \vdash \theta'' P'$, and $(\theta_1 = \theta'' \theta')|_{\Gamma, x : \alpha}$. Let $Q = \text{leq}(\theta' \Gamma|_{\Sigma}, \phi) \cup \text{un}(\theta'(x : \alpha)|_{\Sigma^c})$. By Q-ABS^W and (3), taking a fresh variable ϕ , we have

$$\Gamma \vdash \lambda x. M : \theta' \alpha \rightarrow^{\phi} C' \dashv \theta', P' \cup Q, \Sigma \setminus x$$

With $\theta_2 = \theta''[Y/\phi]$, we have $(\theta = \theta_2 \theta')|_{\Gamma, x : \alpha}$. By $P \vdash \theta\Gamma \leq Y$, Lemma C.9, Lemma C.10, and Theorem 5.3, there exists θ_p such that $P \vdash \theta_p \theta_2 Q$. Let $\theta_3 = \theta_p \theta_2$. Our goal follows from $(\theta = \theta_3 \theta')|_{\Gamma, x : \alpha}$, $A \rightarrow^Y C = \theta_3(\theta' \alpha \rightarrow^{\phi} C')$ and $P \vdash \theta_3(P' \cup Q)$.

Case

$$\frac{\text{Q-APP} \quad \begin{array}{l} P \mid \theta(\Gamma_1, \Gamma) \vdash V : B \rightarrow^Y C \text{ (1)} \quad P \mid \theta(\Gamma_2, \Gamma) \vdash W : B \text{ (2)} \quad P \vdash \theta\Gamma \leq \bullet \end{array}}{P \mid \theta(\Gamma_1, \Gamma_2, \Gamma) \vdash V W : C}$$

By the IH on (1), we have $\Gamma_1, \Gamma \vdash V : A' \dashv \theta_1, P_1, \Sigma_1$ (3) and there exists θ'_1 such that $B \rightarrow^Y C = \theta'_1 A'$, $P \vdash \theta'_1 P_1$, and $(\theta = \theta'_1 \theta_1)|_{\Gamma_1, \Gamma}$. Let $\theta = \theta' \theta'_1 \theta_1$ where θ' only substitutes type variables only appearing in Γ_2 . By the IH on (2), we have $\theta_1(\Gamma_2, \Gamma) \vdash W : B' \dashv \theta_2, P_2, \Sigma_2$ (4)

and there exists θ'_2 such that $B = \theta'_2 B'$, $P \vdash \theta'_2 P_2$, and $(\theta' \theta'_1 = \theta'_2 \theta_2)|_{\Gamma_2, \Gamma}$ (5). Take fresh variables α, μ, ϕ . By $B \rightarrow^Y C = \theta'_1 A'$, the unification $\theta_2 A' \sim B' \rightarrow^\phi \alpha! \mu : \theta_3$ succeeds. By Lemma C.4 and (5), there exists θ_4 such that $\theta_4 \theta_3 (\theta_2 A') = \theta_4 \theta_3 (B' \rightarrow^\phi \alpha! \mu) = B \rightarrow^Y C$. Let $P_3 = \theta_3 (\theta_2 P_1 \cup P_2)$ and $Q = \text{un}(\theta_3 \theta_2 \theta_1 (\Gamma_1, \Gamma_2, \Gamma)|_{\Sigma_1 \cap \Sigma_2})$. By Q-APP^W, (3), (4) and context weakening, we have $\Gamma_1, \Gamma_2, \Gamma \vdash V W : \theta_3 (\alpha! \mu) \dashv \theta_3 \theta_2 \theta_1, P_3 \cup Q, \Sigma_1 \cup \Sigma_2$. With $\theta'' = \theta_4 \theta'_2 \theta'_1$, we have $(\theta = \theta'' \theta_3 \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma}$. By $\Sigma_1 \cap \Sigma_2 \subseteq \text{dom}(\Gamma)$, $P \vdash \theta \Gamma \leq \bullet$, Lemma C.10, and Theorem 5.3, we have $P \vdash \theta_p \theta'' Q$. Let $\theta_5 = \theta_p \theta''$. Our goal follows from $C = \theta_5 \theta_3 (\alpha! \mu)$, $P \vdash \theta_5 (P_3 \cup Q)$ and $(\theta = \theta_5 \theta_3 \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma}$.

Case

$$\frac{\text{Q-RETURN} \quad P \mid \theta \Gamma \vdash V : A \ (1)}{P \mid \theta \Gamma \vdash \mathbf{return} \ V : A! \{R\}}$$

Our goal follows from the IH on (1).

Case

$$\frac{\text{Q-DO} \quad P \mid \theta \Gamma \vdash V : A_\ell \ (1) \quad P \vdash \{\ell : A_\ell \rightarrow^Y B_\ell\} \leq R}{P \mid \theta \Gamma \vdash \mathbf{do} \ \ell \ V : B_\ell! \{E\}}$$

Similar to previous cases. Our goal follows from the IH on (1), Lemma C.4, and Theorem 5.3.

Case

$$\frac{\text{Q-SEQ} \quad \begin{array}{l} P \mid \theta(\Gamma_1, \Gamma) \vdash M : A! \{R_1\} \ (1) \quad P \mid \theta(\Gamma_2, \Gamma), x : A \vdash N : B! \{R_2\} \ (2) \\ P \vdash R_1 \leq R \quad P \vdash R_2 \leq R \quad P \vdash \theta \Gamma_2 \leq R_1 \quad P \vdash \theta \Gamma \leq \bullet \end{array}}{P \mid \theta(\Gamma_1, \Gamma_2, \Gamma) \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B! \{R\}}$$

By the IH on (1), we have $\Gamma_1, \Gamma \vdash M : A'! \{R'_1\} \dashv \theta_1, P_1, \Sigma_1$ (4) and there exists θ'_1 such that $A! \{R_1\} = \theta'_1 (A'! \{R'_1\})$, $P \vdash \theta'_1 P_1$, and $(\theta = \theta'_1 \theta_1)|_{\Gamma_1, \Gamma}$. Let $\theta = \theta' \theta'_1 \theta_1$ where θ' substitutes type variables only appearing in Γ_2 . By (2), we have $P \mid \theta' \theta'_1 \theta_1 (\Gamma_2, \Gamma, x : A') \vdash N : B! \{R_2\}$ (3). By the IH on (3), we have $\theta_1 (\Gamma_2, \Gamma, x : A') \vdash N : B'! \{R'_2\} \dashv \theta_2, P_2, \Sigma_2$ (5) and there exists θ'_2 such that $B! \{R_2\} = \theta'_2 (B'! \{R'_2\})$, $P \vdash \theta'_2 P_2$ and $(\theta' \theta'_1 = \theta'_2 \theta_2)|_{\Gamma_2, \Gamma}$. Take a fresh variable μ . Let $Q = \text{un}(\theta_2 \theta_1 (\Gamma_1, \Gamma_2, \Gamma)|_{\Sigma_1 \cap \Sigma_2}) \cup \text{un}(\theta_2 (x : A)|_{\Sigma_2}) \cup \text{leq}(\theta_2 \theta_1 (\Gamma_1, \Gamma_2, \Gamma)|_{\Sigma_2}, \theta_2 R_1) \cup \text{sub}(\theta_2 R_1, \mu) \cup \text{sub}(R_2, \mu)$. By Q-SEQ^W, (4), (5), and context weakening, we have $\Gamma_1, \Gamma_2, \Gamma \vdash \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N : B'! \{R_2\} \dashv \theta_2 \theta_1, \theta_2 P_1 \cup P_2 \cup Q, \Sigma_1 \cup (\Sigma_2 \setminus x)$. With $\theta'' = [R/\mu] \theta'_2 \theta'_1$, we have $(\theta = \theta'' \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma}$. By $\Sigma_1 \cap \Sigma_2 \subseteq \text{dom}(\Gamma)$, $P \vdash \theta \Gamma \leq \bullet$, Lemma C.9, $\Gamma_2 = \Gamma|_{\Sigma_2}$, $P \vdash \theta \Gamma_2 \leq R_1$, $P \vdash R_1 \leq R$, $P \vdash R_2 \leq R$, Lemma C.10 and Theorem 5.3, there exists θ_p such that $P \vdash \theta_p \theta'' Q$. Let $\theta_3 = \theta_p \theta''$. Our goal follows from $B! \{R_2\} = \theta_3 (B'! \{\mu\})$, $P \vdash \theta_3 (\theta_2 P_1 \cup P_2)$, and $(\theta = \theta_3 \theta_2 \theta_1)|_{\Gamma_1, \Gamma_2, \Gamma}$.

Case

$$\frac{\text{Q-HANDLE} \quad \begin{array}{l} P \mid \theta(\Gamma_1, \Gamma) \vdash H : A! \{R_1\} \Rightarrow D \ (1) \quad P \mid \theta(\Gamma_2, \Gamma) \vdash M : A! \{R\} \ (2) \\ P \vdash \theta \Gamma \leq \bullet \quad P \vdash R \leq R_1 \ (3) \end{array}}{P \mid \theta(\Gamma_1, \Gamma_2, \Gamma) \vdash \mathbf{handle} \ M \ \mathbf{with} \ H : D}$$

By a similar proof to the Q-APP case, our goal follows from the IHs on (1) and (2), Lemma C.10, Theorem 5.3, and Lemma C.4. The only difference is the subtyping constraint $\text{sub}(\theta_3 R, \theta_3 \theta_2 R_1)$ used by Q-HANDLE^W, which follows from (3).

Case

Q-HANDLER

$$\begin{array}{c}
C = A! \{(\ell_i : A_{\ell_i} \twoheadrightarrow^{Y_i} B_i)_i; R_1\} \quad D = B! \{R_2\} \\
P \mid \theta\Gamma, x : A \vdash M : D \text{ (1)} \quad [P \mid \theta\Gamma, p_i : A_i, r_i : B_i \twoheadrightarrow^{Y_i} D \vdash N_i : D]_i \text{ (2)} \\
P \vdash \theta\Gamma \leq \bullet \quad P \vdash R_1 \leq R_2 \quad P \vdash R_1 \perp \{\ell_i\}_i \\
\hline
P \mid \theta\Gamma \vdash \{\mathbf{return} \ x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_{i=1}^n : C \Rightarrow D
\end{array}$$

The typing rule for handler is the most complicated one, but there is actually nothing new of the proof compared to previous cases for other rules. For each typing derivation on the handler clauses, we do a similar proof to the Q-Abs case. Take fresh variables α, ϕ_i , and μ . First, by (1) we have $P \mid \theta[A/\alpha](\Gamma, x : \alpha) \vdash M : D$. By the IH on it, we have $\Gamma, x : \alpha \vdash M : D' \dashv \theta_0, P_0, \Sigma_0$ (3) and there exists θ'_0 such that $D = \theta'_0 D'$, $P \vdash \theta'_0 P_0$ and $(\theta[A/\alpha] = \theta'_0 \theta_0)|_{\Gamma, x, \alpha}$. Let $\theta_0^a = \theta_0$ and $\theta_0^b = \theta'_0$. We have $(\theta_0^b \theta_0^a = \theta)|_{\Gamma}$.

By the typing derivation on the first handler clause in (2), we have $P \mid \theta_0^b [Y_1/\phi_1] \theta_0^a (\Gamma, p_1 : A_1, r_1 : B_1 \twoheadrightarrow^{\phi_1} D) \vdash N_1 : D$. By the IH on it, we have $\theta_0^a (\Gamma, p_1 : A_1, r_1 : B_1 \twoheadrightarrow^{\phi_1} D) \vdash N_1 : D_1 \dashv \theta_1, P_1, \Sigma_1$ and θ'_1 such that $D = \theta'_1 D_1$, $P \vdash \theta'_1 P_1$ and $(\theta_0^b [Y_1/\phi_1] = \theta'_1 \theta_1)|_{(\Gamma, p_1 : A_1, r_1 : B_1 \twoheadrightarrow^{\phi_1} D)}$. By $D = \theta'_1 D_1$, the unification $D_1 \sim \theta'_1 \theta_1 D' : \theta_1^x$ succeeds. By Lemma C.4, there exists θ_1^y such that $\theta_1^y D_1 = D$. Set $\theta_1^a = \theta_x \theta_1 \theta_0^a$ and $\theta_1^b = \theta'_1 \theta_1^y$. We have $(\theta_1^b \theta_1^a = \theta)|_{\Gamma}$.

Repeating the above process for every i from 2 to n , we have $\theta_{i-1}^a (\Gamma, p_i : A_i, r_i : B_i \twoheadrightarrow^{\phi_i} D) \vdash N_i : D_i \dashv \theta_i, P_i, \Sigma_i$ (4) and $(\theta_i^b \theta_i^a = \theta)|_{\Gamma}$. Let

$$\begin{aligned}
C' &= \theta_n^a (\alpha! \{(\ell_i : A_{\ell_i} \twoheadrightarrow^{\phi_i} B_i)_i; \mu\}) \\
B'! \{R\} &= \theta_n^a D' \\
\Sigma &= (\Sigma_0 \setminus \{x\}) \cup (\cup_{i=1}^n (\Sigma_i \setminus \{p_i, r_i\})) \\
P' &= (\cup_{i=0}^n \theta_n^a P_i) \cup \text{un}(\theta_n^a \Gamma|_{\Sigma}) \cup \text{sub}(\mu, R) \cup \text{lack}(\mu, \{\ell_i\}_i) \\
Q' &= \text{un}(\theta_n^a (x : \alpha)|_{\Sigma_0^c}) \cup (\cup_{i=1}^n \text{un}(\theta_n^a (p_i : A_{\ell_i}, r_i : B_{\ell_i} \twoheadrightarrow^{\phi_i} D)))
\end{aligned}$$

By Q-HANDLER^W, (3), and (4), we have $\Gamma \vdash \{\mathbf{return} \ x \mapsto M\} \uplus \{\ell_i \ p_i \ r_i \mapsto N_i\}_{i=1}^n : C' \Rightarrow \theta_n^a D' \dashv \theta_n^a, P' \cup Q', \Sigma$. With $\theta' = \theta_n^b [R_1/\mu]$, we have $(\theta = \theta' \theta_n^a)|_{\Gamma}$. By Lemma C.9, Lemma C.10, and Theorem 5.3 there exists θ_p such that $P \vdash \theta_p \theta' (P \cup Q)$. Let $\theta'' = \theta_p \theta'$. Our goal follows from $C \Rightarrow D = \theta'' (C' \Rightarrow \theta_n^a D')$, $P \vdash \theta'' (P \cup Q)$, and $(\theta = \theta'' \theta_n^a)|_{\Gamma}$. \square

C.4 Correctness of Constraint Solving

LEMMA C.11. *If $\text{urow}(K_1, K_2)$ returns (K'_1, K'_2, θ) , then $\llbracket K_1 \leq K_2 \rrbracket_{\text{sat}} = \llbracket K'_1 \leq K'_2 \rrbracket_{\text{sat}} \theta$; if it fails, then $K_1 \leq K_2$ is not satisfiable.*

PROOF. By Lemma C.3, the substitution θ returned by $\text{urow}(K_1, K_2)$ is the principal unifier that unifies the linearity types of the same labels in K_1 and K_2 , which is a necessary condition for any solution of $K_1 \leq K_2$. \square

LEMMA C.12. *If $\text{factorise}(P) = Q$, then $\llbracket P \rrbracket_{\text{sat}} = \llbracket Q \rrbracket_{\text{sat}}$.*

PROOF. By Theorem 5.3, we have $P \vdash Q$ and $Q \vdash P$. For any $\theta \in \llbracket P \rrbracket_{\text{sat}}$, we have $\cdot \vdash \theta P$. By the closure property of Theorem C.1, we have $\theta P \vdash \theta Q$. By the transitivity of Theorem C.1, we have $\cdot \vdash \theta Q$, which implies $\theta \in \llbracket Q \rrbracket_{\text{sat}}$. Symmetrically, for any $\theta \in \llbracket Q \rrbracket_{\text{sat}}$, we can prove $\theta \in \llbracket P \rrbracket_{\text{sat}}$. Finally, we have $\llbracket P \rrbracket_{\text{sat}} = \llbracket Q \rrbracket_{\text{sat}}$. \square

THEOREM 5.4 (CORRECTNESS OF CONSTRAINT SOLVING). *For any constraint set P generated by the type inference of $\text{Q}_{\text{eff}}^{\circ}$, $\text{solve}(P)$ always terminates.*

- If it fails, then P is not satisfiable.
- If it returns (θ, Q) , then P is satisfiable and $\llbracket P \rrbracket_{sat} = \llbracket Q \rrbracket_{sat} \theta$.

PROOF. The termination of `trlin` and `factorise` is obvious. It may be not very obvious that `srow` always terminates since the `srow` (θ, P, Q) moves the solved predicates in P to the set of unsolved constraints Q in some cases. Note that only row subtyping constraints of forms $K_1 \leq K_2; \mu_2$ and $K_1; \mu_1 \leq K_2; \mu_2$ might require resolving previously solved constraints because they substitute row variables. In both cases, when $\text{set}(K'_1) \not\subseteq \text{set}(K'_2)$, we substitute μ_2 with $(K'_1 \setminus K'_2); \mu$. Notice that the number of labels used in the whole predicate set is finite, and the `srow` fails when there are duplicated labels in the same row, which implies that this kind of substitution terminates. Finally, we can conclude that `srow` terminates.

For the correctness, the idea is to show that every step preserves solutions. We first show `srow` preserves solutions by proving a lemma that if `srow` (θ, P, Q) returns $(\theta' \theta, Q')$, then we have $\llbracket P \cup Q \rrbracket_{sat} = \llbracket Q' \rrbracket_{sat} \theta'$; if it fails, then $P \cup Q$ is not satisfiable. We prove by induction on the definition of `srow`.

Case

$$\text{srow}(\theta, P, \cdot) = \text{return}(\theta, P)$$

Our goal follows from $\llbracket P \rrbracket_{sat} = \llbracket P \rrbracket_{sat} \iota$.

Case

$$\text{srow}(\theta, P, (\tau_1 \leq \tau_2, Q)) = \text{srow}(\theta, (P, \tau_1 \leq \tau_2), Q) \quad (1)$$

Our goal follows from the IH on (1) and $\llbracket P \cup (\tau_1 \leq \tau_2, Q) \rrbracket_{sat} = \llbracket (P, \tau_1 \leq \tau_2) \cup Q \rrbracket_{sat}$.

Case

$$\begin{aligned} \text{srow}(\theta, P, (K_1 \leq K_2, Q)) = \\ \text{let } (K'_1, K'_2, \theta') = \text{urow}(K_1, K_2) \quad (2) \\ \text{assert } \text{set}(K'_1) \subseteq \text{set}(K'_2) \quad (3) \\ \text{srow}(\theta' \theta, \theta' P, \theta' Q) \quad (1) \end{aligned}$$

Obviously (3) fails when $K'_1 \leq K'_2$ is not satisfiable. Our goal follows from the IH on (1), Lemma C.11 on (2), and $\llbracket P \cup (K_1 \leq K_2, Q) \rrbracket_{sat} = \llbracket \theta' P \cup \theta' Q \rrbracket_{sat} \theta'$.

Case

$$\begin{aligned} \text{srow}(\theta, P, (K_1; \mu \leq K_2; \mu, Q)) = \\ \text{let } (K'_1, K'_2, \theta') = \text{urow}(K_1, K_2) \quad (2) \\ \text{assert } \text{set}(K'_1) \subseteq \text{set}(K'_2) \quad (3) \\ \text{srow}(\theta' \theta, \theta' P, \theta' Q) \quad (1) \end{aligned}$$

Obviously (3) fails when $K'_1 \leq K'_2$ is not satisfiable. Our goal follows from the IH on (1), Lemma C.11 on (2), and $\llbracket P \cup (K_1; \mu \leq K_2; \mu, Q) \rrbracket_{sat} = \llbracket \theta' P \cup \theta' Q \rrbracket_{sat} \theta'$.

Case

$$\begin{aligned} \text{srow}(\theta, P, (K_1; \mu \leq K_2, Q)) = \\ \text{let } (K'_1, K'_2, \theta') = \text{urow}(K_1, K_2) \quad (2) \\ \text{assert } \text{set}(K'_1) \subseteq \text{set}(K'_2) \quad (3) \\ \text{srow}(\theta' \theta, (\theta' P, \mu \leq (K'_2 \setminus K'_1)), \theta' Q) \quad (1) \end{aligned}$$

Obviously (3) fails when $K'_1 \leq K'_2$ is not satisfiable. Our goal follows from the IH on (1), Lemma C.11 on (2), and $\llbracket P \cup (K_1; \mu \leq K_2, Q) \rrbracket_{sat} = \llbracket (\theta' P, \mu \leq (K'_2 \setminus K'_1)) \cup \theta' Q \rrbracket_{sat} \theta'$.

Case

$$\begin{aligned}
& \text{srow}(\theta, P, (K_1 \leq K_2; \mu_2, Q)) = \\
& \quad \text{let } (K'_1, K'_2, \theta') = \text{urow}(K_1, K_2) \\
& \quad \text{assume fresh } \mu \\
& \quad \text{if } \text{set}(K'_1) \subseteq \text{set}(K'_2) \\
& \quad \quad \text{then } \text{srow}(\theta' \theta, \theta' P, \theta' Q) \text{ (1)} \\
& \quad \quad \text{else let } \theta'' = [((K'_1 \setminus K'_2); \mu) / \mu_2] \theta' \\
& \quad \quad \quad \text{srow}(\theta'' \theta, \cdot, \theta''(Q, P)) \text{ (2)}
\end{aligned}$$

For the true branch, our goal follows from the IH on (2), Lemma C.3, and

$$\llbracket P \cup (K_1 \leq K_2; \mu_2, Q) \rrbracket_{\text{sat}} = \llbracket \theta' P \cup \theta' Q \rrbracket_{\text{sat}} \theta'$$

. For the false branch, our goal follows from the IH on (2), Lemma C.3, and

$$\llbracket P \cup (K_1 \leq K_2; \mu_2, Q) \rrbracket_{\text{sat}} = \llbracket \theta''(Q, P) \rrbracket_{\text{sat}} \theta''$$

. Both of the above equations follow from the fact that in order to solve $K_1 \leq K_2; \mu_2$, it is necessary to unify the linearity types of the same labels in K_1 and K_2 , and instantiate μ_2 with at least other labels only in K_1 (no instantiation needed when $\text{set}(K'_1) \subseteq \text{set}(K'_2)$).

Case

$$\begin{aligned}
& \text{srow}(\theta, P, (K_1; \mu_1 \leq K_2; \mu_2, Q)) = \\
& \quad \text{let } (K'_1, K'_2, \theta') = \text{urow}(K_1, K_2) \\
& \quad \text{assume fresh } \mu \\
& \quad \text{if } \text{set}(K'_1) \subseteq \text{set}(K'_2) \\
& \quad \quad \text{then } \text{srow}(\theta' \theta, (\theta' P, \mu_1 \leq (K'_2 \setminus K'_1); \mu_2), \theta' Q) \text{ (1)} \\
& \quad \quad \text{else let } \theta'' = [((K'_1 \setminus K'_2); \mu) / \mu_2] \theta' \\
& \quad \quad \quad \text{srow}(\theta'' \theta, \mu_1 \leq (K'_2 \setminus K'_1); \mu, \theta''(Q, P)) \text{ (2)}
\end{aligned}$$

For the true branch of if, our goal follows from the IH on (1), Lemma C.3, and

$$\llbracket P \cup (K_1; \mu_1 \leq K_2; \mu_2, Q) \rrbracket_{\text{sat}} = \llbracket (\theta' P, \mu_1 \leq (K'_2 \setminus K'_1); \mu_2) \cup \theta' Q \rrbracket_{\text{sat}} \theta'$$

For the false branch of if, our goal follows from the IH on (1), Lemma C.3, and

$$\llbracket P \cup (K_1; \mu_1 \leq K_2; \mu_2, Q) \rrbracket_{\text{sat}} = \llbracket (\mu_1 \leq (K'_2 \setminus K'_1); \mu) \cup \theta''(Q, P) \rrbracket_{\text{sat}} \theta''$$

Both of the above two equations follow from the fact that in order to solve $K_1; \mu_1 \leq K_2; \mu_2$, it is necessary to unify the linearity types of the same labels in K_1 and K_2 , and instantiate μ_2 with at least other labels only in K_1 (no instantiation needed when $\text{set}(K'_1) \subseteq \text{set}(K'_2)$).

Case

$$\begin{aligned}
& \text{srow}(\theta, P, (K \perp \mathcal{L}, Q)) = \\
& \quad \text{assert } \text{dom}(K) \cap \mathcal{L} = \emptyset \text{ (2)} \\
& \quad \text{srow}(\theta, P, Q) \text{ (1)}
\end{aligned}$$

Obviously (2) fails when $K \perp \mathcal{L}$ is not satisfiable. Our goal follows the IH on (1).

Case

$$\begin{aligned}
& \text{srow}(\theta, P, (K; \mu \perp \mathcal{L}, Q)) = \\
& \quad \text{assert } \text{dom}(K) \cap \mathcal{L} = \emptyset \text{ (2)} \\
& \quad \text{srow}(\theta, (P, \mu \perp \mathcal{L}), Q) \text{ (1)}
\end{aligned}$$

Obviously (2) fails when $K; \mu \perp \mathcal{L}$ is not satisfiable. Our goal follows the IH on (1).

Then, we can conclude that if $\text{srow } t, \cdot, P$ returns (θ, Q) , then we have $\llbracket P \rrbracket_{\text{sat}} = \llbracket Q \rrbracket_{\text{sat}} \theta$; if it fails, then P is not satisfiable. Moreover, in Q , row subtyping constraints are all in the forms of $\mu \leq K$ and $\mu \leq K; \mu'$.

By Lemma C.12, we have $\llbracket Q \rrbracket_{\text{sat}} = \llbracket \text{factorise}(Q) \rrbracket_{\text{sat}}$. Moreover, in $\text{factorise}(Q)$, linearity constraints are all in atomic forms, which means they are only between type variables, row variables, and linearity types Y .

Let $Q'' = \text{factorise}(Q)$. For $\text{trlin}(\cdot, Q'') = Q'$, we want to show that $\llbracket Q' \rrbracket_{\text{sat}} = \llbracket \text{factorise}(Q) \rrbracket_{\text{sat}}$. Notice that $\text{trlin}(\cdot, Q'')$ essentially computes the transitive closure of the linearity constraints in Q'' . Obviously we have $\llbracket Q'' \rrbracket_{\text{sat}} \subseteq \llbracket Q' \rrbracket_{\text{sat}}$. For the other direction, we need to show that for any $\{\tau_1 \leq \tau_2, \tau_2 \leq \tau_3\} \subseteq Q'$ and $\theta \in \llbracket \tau_1 \leq \tau_2, \tau_2 \leq \tau_3 \rrbracket_{\text{sat}}$, we have $\cdot \vdash \theta(\tau_1 \leq \tau_3)$. Notice that the type inference of Q_{eff}° only generates linearity constraints of forms $\Gamma \leq \tau$, which means rows only appear on the RHS. Thus, after factorisation, $\theta\tau_2$ can only be A or Y . The $\cdot \vdash \theta(\tau_1 \leq \tau_3)$ follows from a straightforward case analysis on $\theta\tau_2$.

Finally, if $\circ \leq \bullet \in Q'$, then Q' is obviously not satisfiable. Otherwise, we have a trivial solution by substituting all row variables with the empty row \cdot , value variables with $()$, and linearity variables with \bullet . We also have $\llbracket P \rrbracket_{\text{sat}} = \llbracket Q' \rrbracket_{\text{sat}} \theta$, which further implies the trivial solution of Q' also gives a solution of P . These results also hold for Q since $\llbracket Q' \rrbracket_{\text{sat}} = \llbracket \text{factorise}(Q) \rrbracket_{\text{sat}} = \llbracket Q \rrbracket_{\text{sat}}$. \square