Soundly Handling Linearity

Wenhao Tang
The University of Edinburgh

EHOP Workshop, 22th July 2023

(Joint work with Daniel Hillerström, Sam Lindley, and J. Garrett Morris)

Linear Types in Links

Links uses linear types for session types:

- `!A.S` : send a value of type `A`, then continue as `S`
- `?A.S` : receive a value of type `A`, then continue as `S`
- `End` : no communication

Linear Types in Links

Links uses linear types for session types:

- `!A.S` : send a value of type `A`, then continue as `S`
- `?A.S` : receive a value of type `A`, then continue as `S`
- `End` : no communication

Primitive operations on session-typed channels:

```
send    : forall (a::Any) (b::Session) . (a, !a.b) -> b
receive : forall (a::Any) (b::Session) . (?a.b) -> (a, b)
fork    : forall          (b::Session) . (b -> ()) -> ~b
close   : End -> ()
```

Linear Types in Links

A sender sends an integer.

```
sig sender      : (!Int.End) ~> ()
fun sender(ch)  { var ch' = send(42, ch); close(ch') }
```

Linear Types in Links

A sender sends an integer.

```
sig sender      : (!Int.End) ~> ()
fun sender(ch)  { var ch' = send(42, ch); close(ch') }
```

A receiver receives the integer and prints it.

```
sig receiver    : (?Int.End) ~> ()
fun receiver(ch) { var (i, ch') = receive(ch); close(ch'); printInt(i) }
```

Linear Types in Links

A sender sends an integer.

```
sig sender       : (!Int.End) ~> ()
fun sender(ch)   { var ch' = send(42, ch); close(ch') }
```

A receiver receives the integer and prints it.

```
sig receiver     : (?Int.End) ~> ()
fun receiver(ch) { var (i, ch') = receive(ch); close(ch'); printInt(i) }
```

Fork the receiver and pass the dual channel to the sender.

```
links> { var ch = fork(receiver); sender(ch) };
42
```

Linear types in LINKS are sound ?

Linear channels cannot be used twice.

```
links> { var ch = fork(receiver); sender(ch); sender(ch); };
Type error: Variable ch has linear type `!Int.End'
but is used 2 times.
```

Linear types in LINKS are sound ?

Linear channels cannot be used twice.

```
links> { var ch = fork(receiver); sender(ch); sender(ch); };
Type error: Variable ch has linear type `!Int.End'
but is used 2 times.
```

Unlimited functions cannot capture linear channels.

```
links> { var ch = fork(receiver);
         var f  = fun(){ sender(ch) }; f(); f() };
Type error: Variable ch of linear type `!Int.End'
is used in a non-linear function literal.
```

Linear types in Links are sound ?

Linear channels cannot be used twice.

```
links> { var ch = fork(receiver); sender(ch); sender(ch); };
Type error: Variable ch has linear type `!Int.End'
but is used 2 times.
```

Unlimited functions cannot capture linear channels.

```
links> { var ch = fork(receiver);
         var f  = fun(){ sender(ch) }; f(); f() };
Type error: Variable ch of linear type `!Int.End'
is used in a non-linear function literal.
```

Linear functions cannot be used twice.

```
links> { var ch = fork(receiver);
         var f  = linfun(){ sender(ch) }; f(); f() };
Type error: Variable f has linear type `() -@ ()'
but is used 2 times.
```

No, well-typed programs in LINKS can go wrong ! [1] [2]

We can use the same channel twice by multi-shot handlers.

```
links> handle
       ({ var ch = fork(receiver); var _ = do Choose; sender(ch) })
       { case <Choose => r> -> r(true); r(false) }
```

[1] https://github.com/links-lang/links/issues/544

[2] Emrich and Hillerström, “Broken Links (Presentation)”, 2020.

No, well-typed programs in LINKS can go wrong ! [1][2]

We can use the same channel twice by multi-shot handlers.

```
links> handle
        ({ var ch = fork(receiver); var _ = do Choose; sender(ch) })
        { case <Choose => r> -> r(true); r(false) }
```

```
***: Internal Error in evalir.ml (Please report as a bug): NotFound
chan_3 (in Hashtbl.find) while interpreting.
```

[1] https://github.com/links-lang/links/issues/544
[2] Emrich and Hillerström, "Broken Links (Presentation)", 2020.

No, well-typed programs in LINKS can go wrong ! [1] [2]

We can use the same channel twice by multi-shot handlers.

```
links> handle
       ({ var ch = fork(receiver); var _ = do Choose; sender(ch) })
       { case <Choose => r> -> r(true); r(false) }


***: Internal Error in evalir.ml (Please report as a bug): NotFound
 chan_3 (in Hashtbl.find) while interpreting.
```

We fix this by extending the linear type system and effect system to track *control flow linearity*, in addition to value linearity.

---

[1] https://github.com/links-lang/links/issues/544
[2] Emrich and Hillerström, "Broken Links (Presentation)", 2020.

Value Linearity in F°_{eff}

Value linearity restricts the *use* of values.

Value linearity characterises whether values contain linear resources.

Value Linearity in $\mathsf{F}^{\circ}_{\text{eff}}$

Value linearity restricts the *use* of values.

Value linearity characterises whether values contain linear resources.

$\mathsf{F}^{\circ}_{\text{eff}}$ tracks the value linearity with kinds.

$$\begin{aligned} &\mathit{Int} &&: \mathsf{Type}^{\bullet} \\ &\mathit{File} &&: \mathsf{Type}^{\circ} \\ &(\mathit{File}, \mathit{Int}) &&: \mathsf{Type}^{\circ} \\ &A \to^{\circ} C &&: \mathsf{Type}^{\circ} \end{aligned}$$

Value Linearity in $\mathsf{F}^{\circ}_{\text{eff}}$

Value linearity restricts the *use* of values.

Value linearity characterises whether values contain linear resources.

$\mathsf{F}^{\circ}_{\text{eff}}$ tracks the value linearity with kinds.

$$\begin{array}{ll} \mathit{Int} & : \mathsf{Type}^{\bullet} \\ \mathit{File} & : \mathsf{Type}^{\circ} \\ (\mathit{File}, \mathit{Int}) & : \mathsf{Type}^{\circ} \\ A \to^{\circ} C & : \mathsf{Type}^{\circ} \end{array}$$

Functions are annotated with their value linearity.

$$\lambda^{\bullet} f. (\lambda^{\circ} s. \mathbf{let}\ f' \leftarrow \mathit{write}\,(s, f)\ \mathbf{in}\ \mathit{close}\,f') : \mathit{File} \to^{\bullet} (\mathit{String} \to^{\circ} ())$$

Value Linearity in F°_{eff}

Value linearity restricts the *use* of values.

Value linearity characterises whether values contain linear resources.

F°_{eff} tracks the value linearity with kinds.

$$\begin{aligned} &\mathit{Int} && : \mathsf{Type}^{\bullet} \\ &\mathit{File} && : \mathsf{Type}^{\circ} \\ &(\mathit{File}, \mathit{Int}) && : \mathsf{Type}^{\circ} \\ &A \to^{\circ} C && : \mathsf{Type}^{\circ} \end{aligned}$$

Functions are annotated with their value linearity.

$$\lambda^{\bullet} f.\, (\lambda^{\circ} s.\, \mathbf{let}\ f' \leftarrow \mathit{write}\,(s, f)\ \mathbf{in}\ \mathit{close}\, f') : \mathit{File} \to^{\bullet} (\mathit{String} \to^{\circ} ())$$

It is always safe to use unlimited values just once. We have the subkinding relation $\vdash \mathsf{Type}^{\bullet} \leq \mathsf{Type}^{\circ}$.

Multi-shot handlers abuse linear resources

We get the same problem as LINKS if we only track value linearity in the presence of multi-shot handlers.

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

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

Control Flow Linearity in F°_{eff}

Ctrl flow linearity restricts how many times control may enter a local context.

Ctrl flow linearity characterises whether a local context captures linear resources.

Control Flow Linearity in F°_{eff}

Ctrl flow linearity restricts how many times control may enter a local context.

Ctrl flow linearity characterises whether a local context captures linear resources.

The continuation (context) of *Choose* is control flow linear.

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

Control Flow Linearity in F°_{eff}

F°_{eff} tracks the control flow linearity at the granularity of operations ($\mathit{Choose} : () \twoheadrightarrow^{Y} \mathit{Bool}$), which represents the control flow linearity of their continuations.

Control Flow Linearity in $\mathsf{F}^{\circ}_{\mathrm{eff}}$

$\mathsf{F}^{\circ}_{\mathrm{eff}}$ tracks the control flow linearity at the granularity of operations ($\mathit{Choose} : () \twoheadrightarrow^{Y} \mathit{Bool}$), which represents the control flow linearity of their continuations.

Let-bindings ($\mathbf{let}^{Y}\ x \leftarrow M\ \mathbf{in}\ N$) are annotated with the control flow linearity of the local context (i.e., $\mathbf{let}^{Y}\ x \leftarrow \_\ \mathbf{in}\ N$).

Control Flow Linearity in F°_{eff}

F°_{eff} tracks the control flow linearity at the granularity of operations ($\mathit{Choose} : () \twoheadrightarrow^{Y} \mathit{Bool}$), which represents the control flow linearity of their continuations.

Let-bindings ($\textbf{let}^{Y}\, x \leftarrow M \textbf{ in } N$) are annotated with the control flow linearity of the local context (i.e., $\textbf{let}^{Y}\, x \leftarrow \_ \textbf{ in } N$).

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

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

Ill-typed!

Linear effect rows can be used as unlimited ones

F°_{eff} lifts the control flow linearity of operations to effect rows.

$$\begin{aligned}
(\mathit{Choose} : () \twoheadrightarrow^{\circ} \mathit{Bool}) &\quad : \mathrm{Row}^{\circ} \\
(\mathit{Choose} : () \twoheadrightarrow^{\bullet} \mathit{Bool}) &\quad : \mathrm{Row}^{\bullet} \\
(L_1 : \circ ; L_2 : \circ ; L_3 : \bullet) &\quad : \mathrm{Row}^{\bullet}
\end{aligned}$$

Linear effect rows can be used as unlimited ones

F°_{eff} lifts the control flow linearity of operations to effect rows.

$$\begin{array}{ll}(\mathit{Choose} : () \twoheadrightarrow^{\circ} \mathit{Bool}) & : \mathsf{Row}^{\circ} \\ (\mathit{Choose} : () \twoheadrightarrow^{\bullet} \mathit{Bool}) & : \mathsf{Row}^{\bullet} \\ (L_1 : \circ ; L_2 : \circ ; L_3 : \bullet) & : \mathsf{Row}^{\bullet}\end{array}$$

It is always safe to use control-flow-linear operations in an unlimited context. We have the subkinding relation $\vdash \mathsf{Row}^{\circ} \leq \mathsf{Row}^{\bullet}$. For instance,

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

Control flow linearity is dual to value linearity!

Control Flow Linearity in Links

The original Links does not track control flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : forall (ρ::Row) . (End) {L:() => () | ρ}~> ()
```

Control Flow Linearity in LINKS

The original LINKS does not track control flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : forall (ρ::Row) . (End) {L:() => () | ρ}~> ()
```

We use **xlin** to claim that the current context is control flow linear, and **lindo** to invoke linear operations.

```
links> fun(ch:End) {xlin; lindo L; close(ch)};
fun : forall (ρ::Row(Lin)) . () {L:() =@ () | ρ}~> ()
```

Control Flow Linearity in Links

The original Links does not track control flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : forall (ρ::Row) . (End) {L:() => () | ρ}~> ()
```

We use **xlin** to claim that the current context is control flow linear, and **lindo** to invoke linear operations.

```
links> fun(ch:End) {xlin; lindo L; close(ch)};
fun : forall (ρ::Row(Lin)) . () {L:() =@ () | ρ}~> ()
```

Linear operations can only be handled by linear handlers.

```
links> fun(ch:End) {
  handle ({ xlin; lindo L; close(ch) }) { case <L =@ r> -> xlin; r(()) }
}
fun : forall (θ:Presence(Lin)) (row:Row(Lin)) . (End) {L{θ} | ρ}~> ()
```

xlin is a modality ?

`xlin` creates a linear scope.

xlin is a modality ?

`xlin` creates a linear scope.

$\Box A$: A linear type A

$\Box\ell$: A control-flow-linear operation ℓ

$\Box(A\,!\,\{\ell_1\,;\ell_2\}) = \Box A\,!\,\Box\{\ell_1\,;\ell_2\} = \Box A\,!\,\{\Box\ell_1\,;\Box\ell_2\}$

xlin is a modality ?

`xlin` creates a linear scope.

$\Box A$: A linear type A

$\Box \ell$: A control-flow-linear operation ℓ

$\Box(A \,!\, \{\ell_1 \,;\, \ell_2\}) = \Box A \,!\, \Box\{\ell_1 \,;\, \ell_2\} = \Box A \,!\, \{\Box\ell_1 \,;\, \Box\ell_2\}$

$$\textsc{T-Box}\quad \frac{\Gamma, \text{✂} \vdash V : A}{\Gamma \vdash \mathrm{box}\, V : \Box A}
\qquad
\textsc{T-Unbox}\quad \frac{\Gamma \vdash V : \Box A}{\Gamma, \text{✂}, \Gamma' \vdash \mathrm{unbox}\, V : A}
\qquad
\textsc{T-Var}\quad \frac{}{\Gamma, x : A, \Gamma' \vdash x : A}$$

$$\textsc{T-BoxC}\quad \frac{\Gamma, \text{✂} \vdash M : A \,!\, E}{\Gamma \vdash \mathrm{box}\, M : \Box A \,!\, \Box E}$$

The handler rule guarantees that $\Box\ell$ is handled by resuming exactly once.

Problems with Subkinding-based Linear Types

Linear types in $\mathsf{F}^{\circ}_{\text{eff}}$ (and LINKS) can be annoying.

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

Problems with Subkinding-based Linear Types

Linear types in $\mathsf{F}^{\circ}_{\mathrm{eff}}$ (and LINKS) can be annoying.

$$\mathit{verboseId} \;:\; \forall \mu^{\mathrm{Row}^{Y_1}}\ \alpha^{\mathrm{Type}^{Y_2}}.\, \alpha \rightarrow^{Y_0} \alpha \,!\, \{\mathit{Print} : \mathit{String} \twoheadrightarrow^{Y_3} () \,;\, \mu\}$$
$$\mathit{verboseId} = \Lambda \mu^{\mathrm{Row}^{Y_1}}\ \alpha^{\mathrm{Type}^{Y_2}}.\, \lambda^{Y_0} x.\, \mathbf{let}^{Y_4}\ () \leftarrow \mathbf{do}\ \mathit{Print}\ \text{"}\mathit{idiscalled}\text{"}\ \mathbf{in}\ x$$

We have ten different types for *verboseId*, none of which is the most general.

$$\forall \mu^{\bullet}\ \alpha^{\bullet}.\alpha \rightarrow^{\bullet} \alpha \,!\, \{\mathit{Print} : \bullet \,;\, \mu\} \qquad \forall \mu^{\bullet}\ \alpha^{\bullet}.\alpha \rightarrow^{\circ} \alpha \,!\, \{\mathit{Print} : \bullet \,;\, \mu\}$$
$$\forall \mu^{\bullet}\ \alpha^{\bullet}.\alpha \rightarrow^{\bullet} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\} \qquad \forall \mu^{\bullet}\ \alpha^{\bullet}.\alpha \rightarrow^{\circ} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\}$$
$$\forall \mu^{\circ}\ \alpha^{\bullet}.\alpha \rightarrow^{\bullet} \alpha \,!\, \{\mathit{Print} : \bullet \,;\, \mu\} \qquad \forall \mu^{\circ}\ \alpha^{\bullet}.\alpha \rightarrow^{\circ} \alpha \,!\, \{\mathit{Print} : \bullet \,;\, \mu\}$$
$$\forall \mu^{\circ}\ \alpha^{\bullet}.\alpha \rightarrow^{\bullet} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\} \qquad \forall \mu^{\circ}\ \alpha^{\bullet}.\alpha \rightarrow^{\circ} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\}$$
$$\forall \mu^{\circ}\ \alpha^{\circ}.\alpha \rightarrow^{\bullet} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\} \qquad \forall \mu^{\circ}\ \alpha^{\circ}.\alpha \rightarrow^{\circ} \alpha \,!\, \{\mathit{Print} : \circ \,;\, \mu\}$$

Qualified Linear Types in Q°_{eff}

We can restore principal types by abstracting over linearity and introducing constraints on linearity.

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

Problems with Row-based Effect Types

Effect row types of sequenced computations must be unified.

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

We can only have $R_1 = R_2 = R$, which overly restricts that operations invoked in h must be control flow linear.

Qualified Effect Types in $\mathrm{Q}^{\circ}_{\mathrm{eff}}$

We support row subtyping again by qualified types.

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

Qualified types is expressive. $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ has a full type inference with constraint solving which does not require any type or linearity annotations.

Qualified Effect Types in Q°_{eff}

We support row subtyping again by qualified types.

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

Qualified types is expressive. Q°_{eff} has a full type inference with constraint solving which does not require any type or linearity annotations.

Interesting interaction between row constraints and linearity constraints: $\mu_1 \leqslant \mu_2$ and $\circ \preceq \mu_2$ implies $\circ \preceq \mu_1$.

Qualified Effect Types in Q°_{eff}

We support row subtyping again by qualified types.

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

Qualified types is expressive. Q°_{eff} has a full type inference with constraint solving which does not require any type or linearity annotations.

Interesting interaction between row constraints and linearity constraints: $\mu_1 \leqslant \mu_2$ and $\circ \preceq \mu_2$ implies $\circ \preceq \mu_1$.

But having explicit constraint sets in types is still a pain?

Algebraic Subtyping for Linear Types and Effect Types

Use algebraic subtyping.

Algebraic Subtyping for Linear Types and Effect Types

Use algebraic subtyping.

Algebraic subtyping for row types is standard. Informally,

$$\frac{\Gamma \vdash M : A \,!\, R_1 \qquad N : B \,!\, R_2}{\Gamma \vdash M; N : B \,!\, R_1 \sqcup R_2}$$

Algebraic Subtyping for Linear Types and Effect Types

Use algebraic subtyping.

Algebraic subtyping for row types is standard. Informally,

$$\frac{\Gamma \vdash M : A \,!\, R_1 \qquad N : B \,!\, R_2}{\Gamma \vdash M; N : B \,!\, R_1 \sqcup R_2}$$

Algebraic subtyping for linear types is more interesting. Informally,

$$\begin{aligned} &\lambda x.\lambda y.\lambda z.(x, y, z) &&: \alpha \to \beta \to^{\alpha} \gamma \to^{\alpha \vee \beta} (\alpha, \beta, \gamma) \\ &\lambda x.(x, x) &&: \alpha \wedge \bullet \to (\alpha, \alpha) \end{aligned}$$

Algebraic Subtyping for Linear Types and Effect Types

Use algebraic subtyping.

Algebraic subtyping for row types is standard. Informally,

$$\frac{\Gamma \vdash M : A \,!\, R_1 \qquad N : B \,!\, R_2}{\Gamma \vdash M; N : B \,!\, R_1 \sqcup R_2}$$

Algebraic subtyping for linear types is more interesting. Informally,

$$\begin{aligned} &\lambda x.\lambda y.\lambda z.(x, y, z) &&: \alpha \to \beta \to^{\alpha} \gamma \to^{\alpha \vee \beta} (\alpha, \beta, \gamma) \\ &\lambda x.(x, x) &&: \alpha \wedge \bullet \to (\alpha, \alpha) \end{aligned}$$

It is easy to extend it with control flow linearity. Informally,

$$\begin{aligned} &\mathit{verboseId} \;:\; \alpha \to \alpha \,!\, \{\mathit{Print} : \phi \vee \alpha \,; \mu\} \\ &\mathit{verboseId} = \lambda x.\, \mathbf{do}\ \mathit{Print}\ \text{"idiscalled"} \,; x \end{aligned}$$

Conclusion

- Track control flow linearity when combining linear types with effect handlers.
- Row subtyping is necessary to have a more fine-grained tracking of control flow linearity.

Thank you!