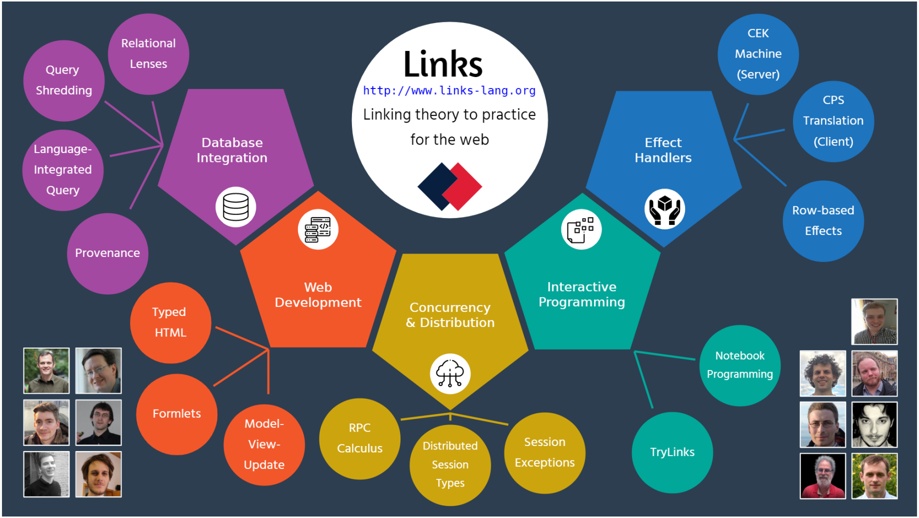
Soundly Handling Linearity

Wenhao Tang
The University of Edinburgh

Seminar, University of Bristol, 3rd Oct 2023

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

Links



Picture by Simon Fowler

Linear Types in Links

Linear types statically guarantee linear resources are used exactly once.

Linear Types in LINKS

Linear types statically guarantee linear resources are used exactly once.

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

Linear types statically guarantee linear resources are used exactly once.

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(c)   { var c' = send(42, c); close(c') }
```

Linear Types in Links

A sender sends an integer.

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

A receiver receives the integer and prints it.

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

Linear Types in Links

A sender sends an integer.

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

A receiver receives the integer and prints it.

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

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

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

Linear types in LINKS are sound

Linear channels cannot be used twice.

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

Unlimited functions cannot capture linear channels.

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

Unlimited functions cannot capture linear channels.

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

Effect Handlers in LINKS

Effect handlers provide advanced mechanisms for manipulating control flow.

Effect Handlers in LINKS

Effect handlers provide advanced mechanisms for manipulating control flow.

Invoke an operation `Choose`.

```
sig choose : () { Choose: () => Bool | _ }~> ()
fun choose() { var i = if (do Choose) 42 else 84; printInt(i) }
```

Effect Handlers in Links

Effect handlers provide advanced mechanisms for manipulating control flow.

Invoke an operation `Choose`.

```
sig choose : () { Choose: () => Bool | _ }~> ()
fun choose() { var i = if (do Choose) 42 else 84; printInt(i) }
```

Handle by invoking the continuation once.

```
links> handle (choose())
       { case <Choose => r> -> r(true) }
42
```

Effect Handlers in Links

Effect handlers provide advanced mechanisms for manipulating control flow.

Invoke an operation `Choose`.

```
sig choose : () { Choose: () => Bool | _ }~> ()
fun choose() { var i = if (do Choose) 42 else 84; printInt(i) }
```

Handle by invoking the continuation once.

```
links> handle (choose())
       { case <Choose => r> -> r(true) }
42
```

Handle by invoking the continuation twice.

```
links> handle (choose())
       { case <Choose => r> -> r(true); r(false) }
4284
```

Well-typed programs in LINKS can go wrong ! [1][2]

A nondeterministic sender sends an integer using the `Choose` operation.

```
sig ndsender : forall r::Row . (!Int.End) { Choose: () => Bool | r}~> ()
fun ndsender(c) {var c' = send(if (do Choose) 42 else 84, c); close(c')}
```

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

Well-typed programs in LINKS can go wrong ! [1][2]

A nondeterministic sender sends an integer using the `Choose` operation.

```
sig ndsender : forall r::Row . (!Int.End) { Choose: () => Bool | r}~> ()
fun ndsender(c) {var c' = send(if (do Choose) 42 else 84, c); close(c')}
```

Use the same channel twice by multi-shot handlers.

```
links> handle ({ var c = fork(receiver); ndsender(c) })
       { case <Choose => r> -> r(true); r(false) };
```

```
42***: 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.

Well-typed programs in LINKS can go wrong ! [1,2]

A nondeterministic sender sends an integer using the Choose operation.

```
sig ndsender : forall r::Row . (!Int.End) { Choose: () => Bool | r}~> ()
fun ndsender(c) {var c' = send(if (do Choose) 42 else 84, c); close(c')}
```

Use the same channel twice by multi-shot handlers.

```
links> handle ({ var c = fork(receiver); ndsender(c) })
       { case <Choose => r> -> r(true); r(false) };
42***: Internal Error in evalir.ml (Please report as a bug):
NotFound chan_3 (in Hashtbl.find) while interpreting.
```

Our solution: 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 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.

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

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.

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

Functions are annotated with their value linearity.

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

Unlimited values can be used as linear values

It is always safe to use unlimited values just once.

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

With the *subkinding* relation $\vdash \mathbf{Type}^\bullet \leq \mathbf{Type}^\circ$, we can instantiate α to *Int*.

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

Multi-shot handlers abuse linear resources

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

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

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

Control-Flow Linearity in F°_{eff}

CFL restricts how many times control may enter a local context.

CFL characterises whether a local context captures linear resources.

Control-Flow Linearity in F°_{eff}

CFL restricts how many times control may enter a local context.

CFL characterises whether a local context captures linear resources.

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

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

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

Linearity $Y ::= \circ \mid \bullet$

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

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

Linearity $Y ::= \circ \mid \bullet$

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

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

Control-Flow Linearity in F°_{eff}

Linearity $Y ::= \circ \mid \bullet$

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

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

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

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

Ill-typed as r is given a linear function type!

Linear effect rows can be used as unlimited ones

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

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

Linear effect rows can be used as unlimited ones

$\mathsf{F}^{\circ}_{\text{eff}}$ lifts the control-flow linearity of operations to effect rows.

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

It is always safe to use control-flow-linear operations in an unlimited context.

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

With the *subkinding* relation $\vdash \mathrm{Row}^{\circ} \leq \mathrm{Row}^{\bullet}$, we have

$$\begin{aligned}
&\mathit{tossCoin}\ \{\mathit{Choose} : \bullet\}\ (\lambda^{\bullet}().(\mathbf{do}\ \ \mathit{Choose}\,())^{\{\mathit{Choose}:\bullet\}}) \\
&\mathit{tossCoin}\ \{\mathit{Choose} : \circ\}\ (\lambda^{\bullet}().(\mathbf{do}\ \ \mathit{Choose}\,())^{\{\mathit{Choose}:\circ\}})
\end{aligned}$$

Control flow linearity is *“dual”* to value linearity!

Control-Flow Linearity in Links

Previously, Links does not track control-flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : (End) {L:() => () | _}~> ()
```

Control-Flow Linearity in LINKS

Previously, LINKS does not track control-flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : (End) {L:() => () | _}~> ()
```

By default, CFL is unlimited. We use the keyword **`xlin`** to switch CFL to linear, and **`lindo`** to invoke control-flow-linear operations.

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

Control-Flow Linearity in Links

Previously, Links does not track control-flow linearity.

```
links> fun(ch:End) {do L; close(ch)};
fun : (End) {L:() => () | _}~> ()
```

By default, CFL is unlimited. We use the keyword **xlin** to switch CFL to linear, and **lindo** to invoke control-flow-linear operations.

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

Control-flow-linear operations can only be handled by one-shot handlers.

```
links> fun(ch:End) { handle ({xlin; lindo L; close(ch)})
                     {case <L =@ r> -> xlin; r(())} };
fun : (End) {L{_::Lin}|_::Lin}~> ()
```

Nondeterministic sender, again

```
sig receiver : (?Int.End) { |_::Lin}~> ()
fun receiver(c) { xlin; var (i, c') = receive(c); close(c'); printInt(i) }
sig ndsender : (!Int.End) {Choose: () => Bool | _::Lin}~> ()
fun ndsender(c) {xlin; close(send(if (lindo Choose) 42 else 84, c))}
```

Nondeterministic sender, again

```
sig receiver : (?Int.End) { |_::Lin}~> ()
fun receiver(c) { xlin; var (i, c') = receive(c); close(c'); printInt(i) }
sig ndsender : (!Int.End) {Choose: () => Bool | _::Lin}~> ()
fun ndsender(c) {xlin; close(send(if (lindo Choose) 42 else 84, c))}
```

```
links> handle ({ xlin; var c = fork(receiver); ndsender(c) })
       { case <Choose => r> -> r(true); r(false) };
  Type error: ... =@ does not match => ...
```

```
links> handle ({ xlin; var c = fork(receiver); ndsender(c) })
       { case <Choose =@ r> -> r(true); r(false) };
  Type error: ... linear function r is used 2 times ...
```

```
links> handle ({ xlin; var c = fork(receiver); ndsender(c) })
       { case <Choose =@ r> -> r(true) };
42
```

Implementation Details

LINKS also adapts a Row-based effect system. Effect types of sequenced computations are unified. For instance,

```
f(42); g(); h("Hello, world!")
```

Implementation Details

LINKS also adapts a Row-based effect system. Effect types of sequenced computations are unified. For instance,

```
f(42); g(); h("Hello, world!")
```

Informally, we introduce the concept *effect scope* to mean the maximal scope where computations have the same effect types. There are only two cases that new effect scopes are created:

- Function bodies (closures) hold their own effect scopes.
- Computations being handled (the `M` in **`handle`** `M {...}`) have their own effect scopes, but also share unhandled effects with outside.

Implementation Details

LINKS also adapts a Row-based effect system. Effect types of sequenced computations are unified. For instance,

```
f(42); g(); h("Hello, world!")
```

Informally, we introduce the concept *effect scope* to mean the maximal scope where computations have the same effect types. There are only two cases that new effect scopes are created:

- Function bodies (closures) hold their own effect scopes.
- Computations being handled (the `M` in **`handle`** `M {...}`) have their own effect scopes, but also share unhandled effects with outside.

`xlin` requires all operations in the current effect scope to be linear.

(Bonus) xlin is a modality ?

Intuition: **`xlin`** creates a linear scope.

(Bonus) xlin is a modality ?

Intuition: **`xlin`** creates a linear scope.

Typing rules for the Fitch-style modal lambda calculus λ_{IK}:

$$\frac{✂ \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x : A}$$

$$\frac{\Gamma, ✂ \vdash M : A}{\Gamma \vdash \mathbf{box}\, M : \Box A}$$

$$\frac{\Gamma \vdash M : \Box A \qquad ✂ \notin \Gamma'}{\Gamma, ✂, \Gamma' \vdash \mathbf{unbox}\, M : A}$$

(Bonus) xlin is a modality ?

TLDR: No, it isn't.

(Bonus) xlin is a modality ?

TLDR: No, it isn't.

$\square A$: a linear type A

(Bonus) xlin is a modality ?

TLDR: No, it isn't.

$\Box A$: a linear type A

If we only consider where linear variables can be used

$$\text{T-VAR}\;\frac{✂ \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x : A}
\qquad
\text{T-BOX(CT)}\;\frac{\Gamma, [✂] \vdash V : A}{\Gamma \vdash \mathbf{box}\, V : \Box A}
\qquad
\text{T-UNBOX(4)}\;\frac{\Gamma \vdash V : \Box A}{\Gamma, ✂, \Gamma' \vdash \mathbf{unbox}\, V : A}$$

(Bonus) xlin is a modality ?

TLDR: No, it isn't.

$\square A$: a linear type A

If we only consider where linear variables can be used

T-Var
$$\frac{✂ \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x : A}$$

T-Box(CT)
$$\frac{\Gamma, [✂] \vdash V : A}{\Gamma \vdash \mathbf{box}\ V : \square A}$$

T-Unbox(4)
$$\frac{\Gamma \vdash V : \square A}{\Gamma, ✂, \Gamma' \vdash \mathbf{unbox}\ V : A}$$

However, it doesn't work well for operations :(

The main problem is that closures should create new scopes.

(Bonus) CFL with modalities

We may still formalise **`xlin`** with modalities.

(Bonus) CFL with modalities

We may still formalise **`xlin`** with modalities.

Consider CBPV. Value linearity is a property of values, while CFL is a property of computations (effects). $\Box A$ and $\Box E$ for unlimited values and effects.

$$\frac{✂ \notin \Gamma'}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma, ✂ \vdash V : A}{\Gamma \vdash \mathbf{box}\, V : \Box A} \qquad \frac{\Gamma \vdash V : \Box A}{\Gamma, \Gamma' \vdash \mathbf{unbox}\, V : A}$$

$$\frac{\Gamma \vdash M : C \dashv E}{\Gamma \vdash \mathbf{thunk}\, M : {\downarrow^E} C} \qquad \frac{\Gamma \vdash V : {\downarrow^E} C}{\Gamma \vdash \mathbf{force}\, V : C \dashv E} \qquad \frac{\Gamma \vdash M : C \dashv \Box E \qquad ✂ \notin \Gamma'}{\Gamma, ✂, \Gamma' \vdash \mathbf{unbox}\, M : C \dashv E}$$

$$\frac{\Gamma \vdash M : {\uparrow} A \dashv E_1 \qquad \Gamma, ✂, x : A \vdash N : C \dashv E_2}{\Gamma \vdash \mathbf{let\ box}\ x \leftarrow M\ \mathbf{in}\ N : C \dashv (\Box E_1) \cup E_2}$$

$$\frac{\Gamma \vdash M : {\uparrow} A \dashv E_1 \qquad \Gamma, x : A \vdash N : C \dashv E_2}{\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C \dashv (\mathsf{lin}(E_1)) \cup E_2}$$

Restriction of Subkinding-based Linear Types

Linear types in $\mathsf{F}^{\circ}_{\text{eff}}$ (and LINKS) can be annoying due to annotations and lack of principal types.

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

Restriction of Subkinding-based Linear Types

Linear types in $\mathsf{F}^{\circ}_{\text{eff}}$ (and Links) can be annoying due to annotations and lack of principal types.

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

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

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

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

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

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

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

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

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

The order of linearity is given by $\bullet \leq \circ$.

$\alpha \leq \phi$: the linearity of the value type α is less than the linearity variable ϕ

$\alpha \leq \mu$: the linearity of the value type α is less than the control-flow linearity of the row type μ

Restriction of Row-based Effect Types

Effect row types of sequenced computations must be unified.

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

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 Q°_{eff}

We support row subtyping again by qualified types.

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

$\mu \leqslant \mu'$: the row type μ is a subrow of the row type μ'

Qualified Effect Types in Q°_{eff}

We support row subtyping again by qualified types.

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

$\mu \leqslant \mu'$: the row type μ is a subrow of the row type μ'

Q°_{eff} has a full type inference which infers principal types and a deterministic constraint solver. It does not require any type or linearity annotations.

Qualified Effect Types in Q°_{eff}

We support row subtyping again by qualified types.

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

$\mu \leqslant \mu'$: the row type μ is a subrow of the row type μ'

Q°_{eff} has a full type inference which infers principal types and a deterministic constraint solver. It 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.

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

$\mu \leqslant \mu'$: the row type μ is a subrow of the row type μ'

Q°_{eff} has a full type inference which infers principal types and a deterministic constraint solver. It 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?

(Bonus) Algebraic Subtyping for Effects

Use algebraic subtyping.

(Bonus) Algebraic Subtyping for Effects

Use algebraic subtyping.

The core idea of algebraic subtyping is to encode subtyping constraints with union and intersection directly in types. For instance,

$$\forall \alpha\, \beta\, \gamma.(\alpha \leqslant \gamma, \beta \leqslant \gamma) \Rightarrow (\alpha, \beta) \to \gamma$$

is transformed to

$$\forall \alpha\, \beta.(\alpha, \beta) \to \alpha \sqcup \beta$$

Algebraic subtyping for row types is quite standard. Informally,

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

$R_1 \sqcup R_2$: the union of row types R_1 and R_2

(Bonus) Algebraic Subtyping for Linearity

Algebraic subtyping for linear types is more interesting. Informally,

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

$\to^{\alpha}$: a function type whose linearity is *at least* the linearity of α

$\alpha \vee \beta$: the union of the linearity of value types α and β

$\alpha \wedge \bullet$: α with linearity that is the intersection of α and $\bullet$

(Bonus) Algebraic Subtyping for Linearity

Algebraic subtyping for linear types is more interesting. Informally,

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

$\to^{\alpha}$: a function type whose linearity is *at least* the linearity of α

$\alpha \vee \beta$: the union of the linearity of value types α and β

$\alpha \wedge \bullet$: α with linearity that is the intersection of α and $\bullet$

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

More in the paper: `https://arxiv.org/abs/2307.09383`

- $\mathrm{F}^{\circ}_{\mathrm{eff}}$: a *system F*-style calculus with subkinding-based linear types and row-based effect types. Core calculus of LINKS (to some extent). Metatheory: type soundness + runtime linearity safety.
- $\mathrm{Q}^{\circ}_{\mathrm{eff}}$: an *ML*-style calculus with linear types and effect types both based on *qualified types*. Full type inference with principal types. Deterministic constraint solving. Better accuracy enabled by effect subtyping.

Conclusion

More in the paper: `https://arxiv.org/abs/2307.09383`

- $\mathrm{F}^{\circ}_{\mathrm{eff}}$: a *system F*-style calculus with subkinding-based linear types and row-based effect types. Core calculus of LINKS (to some extent). Metatheory: type soundness + runtime linearity safety.
- $\mathrm{Q}^{\circ}_{\mathrm{eff}}$: an *ML*-style calculus with linear types and effect types both based on *qualified types*. Full type inference with principal types. Deterministic constraint solving. Better accuracy enabled by effect subtyping.

Potential future work:

- CFL with modalities.
- Algebraic subtyping for linearity (and effects).
- Shallow handlers.

Thank you!

Takeaway: consider tracking control-flow linearity when having both linear types and effect handlers!