Soundly Handling Linearity

Wenhao Tang[1] Daniel Hillerström[2] Sam Lindley[1] J. Garrett Morris[3]

POPL'24, London, UK, 17th Jan 2024

[1] THE UNIVERSITY of EDINBURGH [2] HUAWEI [3] IOWA

Linear Types vs Effect Handlers

linear types

Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

Rust, Haskell

Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

Rust, Haskell

Idris2, Granule

Picture by Xueying Qin

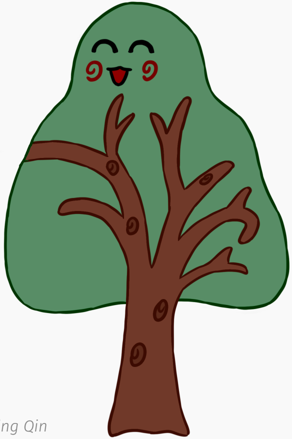
Linear Types vs Effect Handlers

linear types

Rust, Haskell

Idris2, Granule

effect handlers



Picture by Xueying Qin

Linear Types vs Effect Handlers

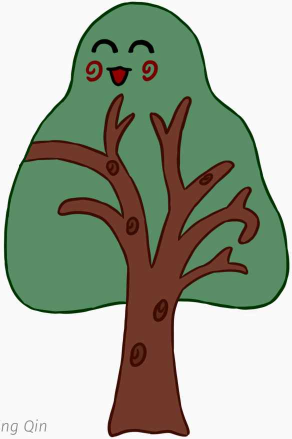
linear types

Rust, Haskell

Idris2, Granule

effect handlers

OCaml, WebAssembly



Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

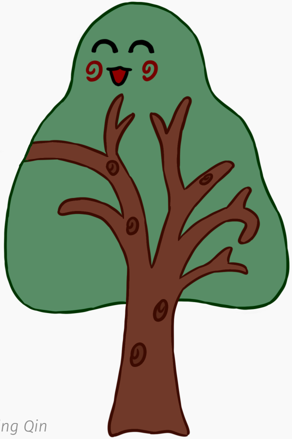
Rust, Haskell

Idris2, Granule

effect handlers

OCaml, WebAssembly

Eff, Koka, Frank, Effekt



Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

Rust, Haskell

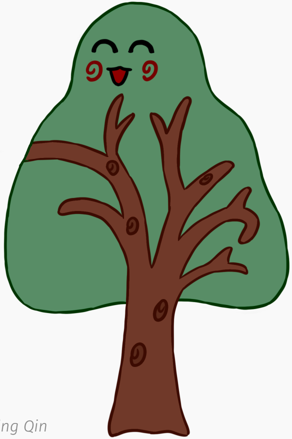
Idris2, Granule

Links

effect handlers

OCaml, WebAssembly

Eff, Koka, Frank, Effekt



Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

Rust, Haskell

Idris2, Granule

Links

effect handlers

OCaml, WebAssembly

Eff, Koka, Frank, Effekt

Picture by Xueying Qin

Linear Types vs Effect Handlers

linear types

Rust, Haskell

Idris2, Granule

Links

effect handlers

OCaml, WebAssembly

Eff, Koka, Frank, Effekt

Picture by Xueying Qin

Linear Types in Links

Links uses linear types for *session types* which characterise communication protocols.

Linear Types in Links

Links uses linear types for *session types* which characterise communication protocols.

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

`!Int.End`: send a value of type `Int`, then `End`

Linear Types in Links

Links uses linear types for *session types* which characterise communication protocols.

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

`!Int.End`: send a value of type `Int`, then `End`

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

`?Int.End`: receive a value of type `Int`, then `End`

Linear Types in Links

Links uses linear types for *session types* which characterise communication protocols.

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

`!Int.End`: send a value of type `Int`, then `End`

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

`?Int.End`: receive a value of type `Int`, then `End`

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

`!Int.End` is dual to `?Int.End`

Well-Typed Programs in Links Cannot Go Wrong ?

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

Well-Typed Programs in LINKS Cannot Go Wrong ?

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

Type error: Variable `c` has linear type ‘`!Int.End`’ but is used 2 times.

Well-Typed Programs in LINKS Cannot Go Wrong ?

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

Type error: Variable `c` has linear type ‘`!Int.End`’ but is used 2 times.

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

Well-Typed Programs in Links Cannot Go Wrong ?

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

```
Type error: Variable c has linear type ‘!Int.End’ but is used 2 times.
```

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

```
Type error: Variable c of linear type ‘!Int.End’ is used in a non-linear function.
```

Well-Typed Programs in Links Cannot Go Wrong ?

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

Type error: Variable `c` has linear type ‘`!Int.End`’ but is used 2 times.

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

Type error: Variable `c` of linear type ‘`!Int.End`’ is used in a non-linear function.

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

Well-Typed Programs in Links Cannot Go Wrong ?

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

Type error: Variable `c` has linear type ‘`!Int.End`’ but is used 2 times.

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

Type error: Variable `c` of linear type ‘`!Int.End`’ is used in a non-linear function.

```
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 us with a flexible way to manipulate control flow.

Effect Handlers in Links

Effect handlers provide us with a flexible way to manipulate control flow.

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

`Choose: () => Bool` takes no parameter and returns a boolean value

Effect Handlers in Links

Effect handlers provide us with a flexible way to manipulate control flow.

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

`Choose: () => Bool` takes no parameter and returns a boolean value

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

one-shot handler

Effect Handlers in Links

Effect handlers provide us with a flexible way to manipulate control flow.

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

`Choose: () => Bool` takes no parameter and returns a boolean value

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

one-shot handler

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

multi-shot handler

Well-Typed Programs in LINKS Can Go Wrong !

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

Well-Typed Programs in Links Can Go Wrong !

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

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

Well-Typed Programs in Links Can Go Wrong !

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

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

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

multi-shot handler

Well-Typed Programs in Links Can Go Wrong !

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

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

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

multi-shot handler

```
42***: Internal Error in evalir.ml : NotFound chan_3 while interpreting.
```

Well-Typed Programs in LINKS Can Go Wrong !

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

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

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

multi-shot handler

This has been a long-standing bug in LINKS: `github.com/links-lang/links/issues/544`.

Well-Typed Programs in LINKS Can Go Wrong !

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

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

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

multi-shot handler

This has been a long-standing bug in LINKS: `github.com/links-lang/links/issues/544`.

Core idea: track ***control-flow linearity*** in addition to ***value linearity***.

Value Linearity

Value linearity restricts the ***use*** of values.

Value Linearity

Value linearity restricts the ***use*** of values.

Value linearity characterises whether ***values*** contain linear resources.

Value Linearity in $\mathrm{F}^{\circ}_{\mathrm{eff}}$ (F-eff-pop)

Value linearity restricts the ***use*** of values.

Value linearity characterises whether ***values*** contain linear resources.

$$Y ::= \underset{\text{linear}}{\circ} \mid \underset{\text{unlimited}}{\bullet}$$

Value Linearity in $\mathsf{F}^{\circ}_{\mathsf{eff}}$ (F-eff-pop)

Value linearity restricts the ***use*** of values.

Value linearity characterises whether ***values*** contain linear resources.

$$Y ::= \underset{\text{linear}}{\circ} \mid \underset{\text{unlimited}}{\bullet}$$

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

$$\begin{array}{rr} \mathtt{Int} : \mathsf{Type}^{\bullet} & !\mathtt{Int}.\mathsf{End} : \mathsf{Type}^{\circ} \\ (!\mathtt{Int}.\mathsf{End}, \mathtt{Int}) : \mathsf{Type}^{\circ} & A \to^{\circ} C : \mathsf{Type}^{\circ} \end{array}$$

Value Linearity in $\mathsf{F}^{\circ}_{\textbf{eff}}$ (F-eff-pop)

Value linearity restricts the ***use*** of values.

Value linearity characterises whether ***values*** contain linear resources.

$$Y ::= \underset{\text{linear}}{\circ} \;\big|\; \underset{\text{unlimited}}{\bullet}$$

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

$$\begin{array}{rr} \texttt{Int} : \mathsf{Type}^{\bullet} & \mathsf{!Int.End} : \mathsf{Type}^{\circ} \\ (\mathsf{!Int.End}, \texttt{Int}) : \mathsf{Type}^{\circ} & A \to^{\circ} C : \mathsf{Type}^{\circ} \end{array}$$

Functions are annotated with value linearity.

$$\mathit{sender} = \underset{\text{unlimited fun}}{\lambda^{\bullet}}\, \underset{\text{linear var}}{c^{\mathsf{!Int.End}}} \,.\, \underset{\text{linear fun}}{\lambda^{\circ}}\, \underset{\text{unlimited var}}{i^{\mathsf{Int}}} . \mathbf{let}\ c' \leftarrow \mathsf{send}\,(i, c)\ \mathbf{in}\ \mathsf{close}\, c'$$

Multi-Shot Handlers Break Value Linearity

$$\mathit{ndsender}_{\text{✗}} \ : \ !\mathtt{Int}.\mathtt{End} \longrightarrow^{\bullet} () \,!\, \{\mathtt{Choose} : () \twoheadrightarrow \mathtt{Bool}\}$$

$$\mathit{ndsender}_{\text{✗}} = \lambda^{\bullet} c.$$

$$\begin{array}{l}
\mathbf{let}\ b \leftarrow \mathbf{do}\ \mathsf{Choose}\,()\ \mathbf{in} \\
\mathbf{let}\ s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
\mathbf{let}\ c' \leftarrow \mathsf{send}\,(s, c)\ \mathbf{in}\ \mathtt{close}\ c'
\end{array}$$

linear variable c is captured in the continuation of Choose

Multi-Shot Handlers Break Value Linearity

$$\begin{aligned}
&\mathit{ndsender}_{\text{✗}} \;:\; \mathtt{!Int.End} \to^{\bullet} ()\,!\,\{\mathsf{Choose} : () \twoheadrightarrow \mathsf{Bool}\} \\
&\mathit{ndsender}_{\text{✗}} = \lambda^{\bullet} c. \\
&\qquad \mathbf{let}\ b \leftarrow \mathbf{do}\ \mathsf{Choose}\,()\ \mathbf{in} \\
&\qquad \mathbf{let}\ s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
&\qquad \mathbf{let}\ c' \leftarrow \mathsf{send}\,(s, c)\ \mathbf{in}\ \mathtt{close}\ c'
\end{aligned}$$

linear variable c is captured in the continuation of Choose

$$\lambda^{\bullet} c.\mathbf{handle}\ (\mathit{ndsender}_{\text{✗}}\ c)\ \mathbf{with}\ \{\mathsf{Choose}\ \_\ r \mapsto r\ \mathtt{true}\,;\, r\ \mathtt{false}\}$$

well-typed but duplicates the channel endpoint c

Control-Flow Linearity

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-Flow Linearity

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-flow linearity characterises whether a ***local context*** captures linear resources.

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.
Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\text{✗}} : \mathsf{Int.End} \rightarrow^{\bullet} () \,!\, \{ \ \mathsf{Choose} : () \twoheadrightarrow \mathsf{Bool} \ \}$$

$$\begin{aligned}
&\mathit{ndsender}_{\text{✗}} = \lambda^{\bullet} c. \\
&\qquad \mathbf{let} \quad b \leftarrow \mathbf{do}\ \mathsf{Choose}\ ()\ \mathbf{in} \\
&\qquad \mathbf{let} \quad s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
&\qquad \mathbf{let} \quad c' \leftarrow \mathsf{send}\ (s, c)\ \mathbf{in}\ \mathsf{close}\ c'
\end{aligned}$$

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\text{✗}} : \mathsf{Int.End} \longrightarrow^{\bullet} ()\,!\,\{\, \overset{\text{control-flow-linear operation}}{\underset{\text{CFL of continuation}}{\mathsf{Choose} : () \twoheadrightarrow^{\circ} \mathsf{Bool}}} \,\}$$

$$\mathit{ndsender}_{\text{✗}} = \lambda^{\bullet} c.$$

$$\begin{aligned}
&\textbf{let} \quad b \leftarrow \textbf{do}\ \mathsf{Choose}\,()\ \textbf{in} \\
&\textbf{let} \quad s \leftarrow \textbf{if}\ b\ \textbf{then}\ 42\ \textbf{else}\ 84\ \textbf{in} \\
&\textbf{let} \quad c' \leftarrow \mathsf{send}\,(s, c)\ \textbf{in}\ \mathsf{close}\,c'
\end{aligned}$$

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\text{✗}} : \mathtt{Int.End} \longrightarrow^{\bullet} ()\,!\,\{\underset{\text{CFL of continuation}}{\overset{\text{control-flow-linear operation}}{\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}}}\}$$

$$\mathit{ndsender}_{\text{✗}} = \lambda^{\bullet} c.$$

$$\begin{aligned}
&\underset{\text{CFL of local context}}{\mathbf{let}^{\circ}} && b \leftarrow \mathbf{do}\ \mathtt{Choose}\ ()\ \mathbf{in} \\
&\mathbf{let} && s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
&\mathbf{let} && c' \leftarrow \mathtt{send}\,(s, c)\ \mathbf{in}\ \mathtt{close}\ c'
\end{aligned}$$

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\checkmark} : \mathsf{Int.End} \to^{\bullet} () \,!\, \{\underbrace{\mathsf{Choose} : () \twoheadrightarrow^{\circ} \mathsf{Bool}}_{\text{CFL of continuation}}\}$$

$$\begin{aligned}
&\mathit{ndsender}_{\checkmark} = \lambda^{\bullet} c. \\
&\quad \underbrace{\mathbf{let}^{\circ}}_{\text{CFL of local context}} \quad b \leftarrow \mathbf{do}\ \mathsf{Choose}\ ()\ \mathbf{in} \\
&\quad \underbrace{\mathbf{let}^{\circ}}_{\textbf{CFL of local context}} \quad s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
&\quad \mathbf{let} \quad c' \leftarrow \mathsf{send}\ (s, c)\ \mathbf{in}\ \mathsf{close}\ c'
\end{aligned}$$

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.

Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\checkmark} : \mathtt{Int.End} \to^{\bullet} ()\,!\,\{\underset{\text{CFL of continuation}}{\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}}\}$$

$$\begin{aligned}
&\mathit{ndsender}_{\checkmark} = \lambda^{\bullet} c. \\
&\quad \underset{\text{CFL of local context}}{\mathbf{let}^{\circ}}\ b \leftarrow \mathbf{do}\ \mathsf{Choose}\ ()\ \mathbf{in} \\
&\quad \underset{\text{CFL of local context}}{\mathbf{let}^{\circ}}\ s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in} \\
&\quad \underset{\textbf{CFL of local context}}{\mathbf{let}^{\bullet}}\ c' \leftarrow \mathsf{send}\,(s, c)\ \mathbf{in}\ \mathtt{close}\ c'
\end{aligned}$$

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

Control-flow linearity (CFL) restricts how many times control may ***enter*** a local context.
Control-flow linearity characterises whether a ***local context*** captures linear resources.

$$\mathit{ndsender}_{\checkmark} \;:\; \mathsf{Int.End} \to^{\bullet} ()\,!\,\{\underbrace{\mathsf{Choose} : () \twoheadrightarrow^{\circ} \mathsf{Bool}}_{\text{CFL of continuation}}\}$$

$$\begin{aligned}
&\mathit{ndsender}_{\checkmark} = \lambda^{\bullet} c.\\
&\qquad \underbrace{\mathbf{let}^{\circ}}_{\text{CFL of local context}}\; b \leftarrow \mathbf{do}\ \mathsf{Choose}\ ()\ \mathbf{in}\\
&\qquad \underbrace{\mathbf{let}^{\circ}}_{\text{CFL of local context}}\; s \leftarrow \mathbf{if}\ b\ \mathbf{then}\ 42\ \mathbf{else}\ 84\ \mathbf{in}\\
&\qquad \underbrace{\mathbf{let}^{\bullet}}_{\textbf{CFL of local context}}\; c' \leftarrow \mathsf{send}\,(s, c)\ \mathbf{in}\ \mathsf{close}\ c'
\end{aligned}$$

$$\lambda^{\bullet} c.\mathbf{handle}\ (\mathit{ndsender}_{\checkmark}\ c)\ \mathbf{with}\ \{\mathsf{Choose}\ \_\ r \mapsto r\ \mathtt{true}\,;\, r\ \mathtt{false}\}$$

ill-typed since r is now a linear function because **Choose** is control-flow linear

Subkinding of Linearity

The control-flow linearity of operations are lifted to the kind of effect rows.

Subkinding of Linearity

The control-flow linearity of operations are lifted to the kind of effect rows.

$$\begin{aligned}
&(\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) && : \mathsf{Row}^{\circ} \\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}) && : \mathsf{Row}^{\bullet} \\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}, \mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) && : \mathsf{Row}^{\bullet}
\end{aligned}$$

Subkinding of Linearity

The control-flow linearity of operations are lifted to the kind of effect rows.

$$\begin{aligned}
(\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) \quad &: \mathrm{Row}^{\circ} \\
(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}) \quad &: \mathrm{Row}^{\bullet} \\
(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}, \mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) \quad &: \mathrm{Row}^{\bullet}
\end{aligned}$$

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

$$\vdash \mathrm{Row}^{\circ} \leq \mathrm{Row}^{\bullet}$$

Subkinding of Linearity

The control-flow linearity of operations are lifted to the kind of effect rows.

$$\begin{aligned}
&(\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) &&: \mathsf{Row}^{\circ}\\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}) &&: \mathsf{Row}^{\bullet}\\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}, \mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) &&: \mathsf{Row}^{\bullet}
\end{aligned}$$

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

$$\vdash \mathsf{Row}^{\circ} \leq \mathsf{Row}^{\bullet}$$

It is always safe to use unlimited values just once.

$$\vdash \mathsf{Type}^{\bullet} \leq \mathsf{Type}^{\circ}$$

Subkinding of Linearity

The control-flow linearity of operations are lifted to the kind of effect rows.

$$\begin{aligned}
&(\mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) && : \mathsf{Row}^{\circ} \\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}) && : \mathsf{Row}^{\bullet} \\
&(\mathtt{Print} : () \twoheadrightarrow^{\bullet} \mathtt{Bool}, \mathtt{Choose} : () \twoheadrightarrow^{\circ} \mathtt{Bool}) && : \mathsf{Row}^{\bullet}
\end{aligned}$$

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

$$\vdash \mathsf{Row}^{\circ} \leq \mathsf{Row}^{\bullet}$$

It is always safe to use unlimited values just once.

$$\vdash \mathsf{Type}^{\bullet} \leq \mathsf{Type}^{\circ}$$

Control-flow linearity is ***dual*** to value linearity!

Value linearity is about values, and control-flow linearity is about contexts.

Tracking Control-Flow Linearity in Links

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

sig ndsender    : (!Int.End) {Choose: () => Bool | _ }~> ()
fun ndsender(c) { close(send(if (do Choose) 42 else 84, c)) }

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

42***: Internal Error in evalir.ml : NotFound chan_3 while interpreting.
```

Tracking Control-Flow Linearity in LINKS

```
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> -> xlin; r(true); r(false) };
```

Type Error: Variable `r` has linear type but is used 2 times.

Tracking Control-Flow Linearity in Links

```
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> -> xlin; r(true); r(false) };
```

Type Error: Variable `r` has linear type but is used 2 times.

Now close the issue! `github.com/links-lang/links/issues/544`

Beyond $\mathrm{F}^{\circ}_{\mathbf{eff}}$ and Links

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

$$\mathit{verboseId} = \lambda x.\, \mathbf{do}\ \texttt{Print}\ \text{"42"}\,;\, x$$

Beyond $\mathsf{F}^{\circ}_{\mathbf{eff}}$ and Links

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

$$\mathit{verboseId} = \lambda x.\, \mathbf{do}\ \mathtt{Print}\ \text{"42"}\, ;\, x$$

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

Beyond $\mathsf{F}^{\circ}_{\mathbf{eff}}$ and LINKS : $\mathrm{Q}^{\circ}_{\mathbf{eff}}$ (Q-eff-pop)

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

$$\mathit{verboseId} = \lambda x.\, \mathbf{do}\ \texttt{Print}\ \texttt{"42"}\, ;\, x$$

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

$\mathrm{Q}^{\circ}_{\mathsf{eff}}$ restores principal types by *qualified types*.

$$\forall \alpha\, \mu\, \phi\, \phi'.\, \alpha \leq \phi \Rightarrow \alpha \rightarrow^{\phi'} \alpha\, !\, \{\texttt{Print} : \phi ; \mu\}$$

Beyond $\mathrm{F}^{\circ}_{\mathrm{eff}}$ and LINKS : $\mathrm{Q}^{\circ}_{\mathrm{eff}}$ (Q-eff-pop)

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

$$\mathit{verboseId} = \lambda x.\,\mathbf{do}\ \mathtt{Print}\ \text{"42"}\,;\,x$$

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

$\mathrm{Q}^{\circ}_{\mathrm{eff}}$ restores principal types by *qualified types*.

$$\forall \alpha\,\mu\,\phi\,\phi'.\ \alpha \leq \phi \Rightarrow \alpha \to^{\phi'} \alpha\,!\,\{\mathtt{Print} : \phi;\mu\}$$

$\mathrm{Q}^{\circ}_{\mathrm{eff}}$ also supports effect subtyping, making CFL more precise.

More in the Paper

F°_{eff} *system-F* style

subkinding-based linear types [Mazurak et al. 2010]

row-based effect types [Hillerström and Lindley 2016]

implementation in Links

metatheory (type soundness and runtime linearity safety)

Q°_{eff} *ML* style

***qualified* linear types** based on Quill [Morris 2016]

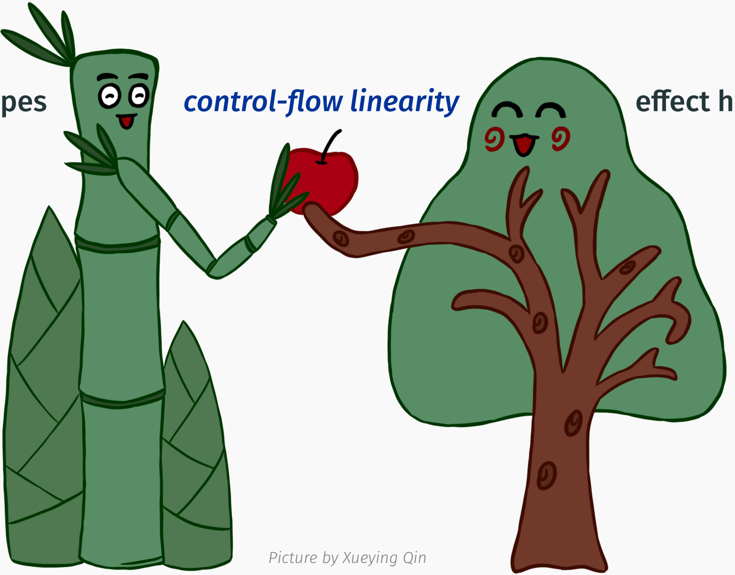
***qualified* effect types** based on Rose [Morris and McKinna 2019]

type inference with principal types

deterministic constraint solving

metatheory (soundness and completeness of type inference)

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



linear types

control-flow linearity

effect handlers

Picture by Xueying Qin