Tracking Linear Continuations for Effect Handlers

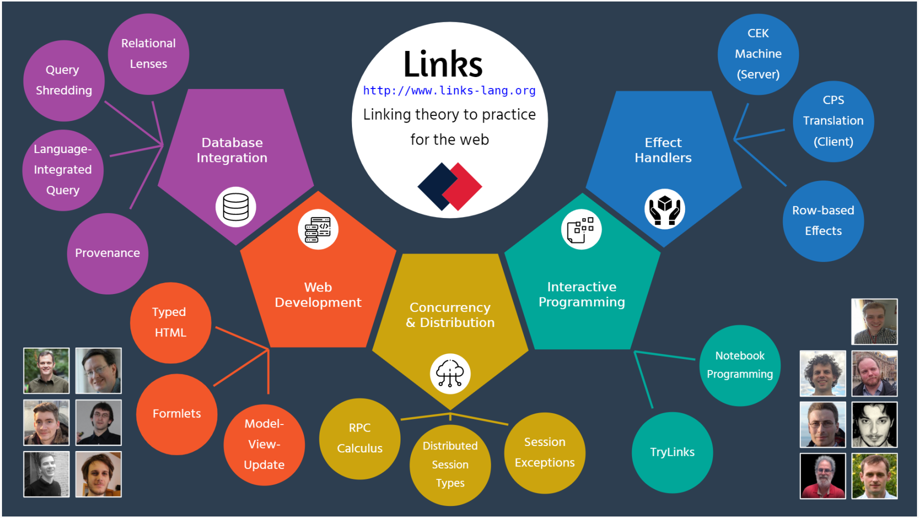
Wenhao Tang

The University of Edinburgh

Huawei Edinburgh Joint Lab Workshop, 6th June 2023

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

Links



Picture by Simon Fowler

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    : ∀ a (b::Session) . (a, !a.b) → b
receive : ∀ a (b::Session) . (?a.b) → (a, b)
fork    : ∀   (a::Session) . (a → ()) → ~a
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
```

Well-Typed Programs in Links Cannot Go Wrong

Linear types prevent us from using the same channel twice.

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

Well-Typed Programs in Links Cannot Go Wrong

Linear types prevent us from using the same channel twice.

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

Even if it is wrapped in a function.

```
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.
```

Well-Typed Programs in Links Cannot Go Wrong

Linear types prevent us from using the same channel twice.

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

Even if it is wrapped in a function.

```
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.
```

Or in a linear function.

```
links> { var ch = fork(receiver);
         var f  = linfun(){ sender(ch) }; f(); f() };
<stdin>:1: Type error: Variable f has linear type `() -@ ()'
but is used 2 times.
```

Effect Handlers in Links

Algebraic effects and handlers provide programmers with advanced control-flow mechanisms.

Effect Handlers in Links

Algebraic effects and handlers provide programmers with advanced control-flow mechanisms.

```
sig choose : ∀ (ρ::Row) . () { Choose: () ⇒ Bool | ρ }→ ()
fun choose() { var i = if (do Choose) 42 else 24; printInt(i) }
```

Effect Handlers in Links

Algebraic effects and handlers provide programmers with advanced control-flow mechanisms.

```
sig choose : ∀ (ρ::Row) . () { Choose: () ⇒ Bool | ρ }→ ()
fun choose() { var i = if (do Choose) 42 else 24; printInt(i) }
```

Handle by invoking the continuation once.

```
links> handle (choose())
         { case <Choose ⇒ r> → r(true) }
42
```

Effect Handlers in Links

Algebraic effects and handlers provide programmers with advanced control-flow mechanisms.

```
sig choose : ∀ (ρ::Row) . () { Choose: () ⇒ Bool | ρ }⟶ ()
fun choose() { var i = if (do Choose) 42 else 24; 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) }
4224
```

Well-Typed Programs in Links CAN Go Wrong ! [1] [2]

We can use the same channel twice by invoking the continuation twice.

```
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.

Well-Typed Programs in Links CAN Go Wrong ! [1,2]

We can use the same channel twice by invoking the continuation twice.

```
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.

Well-Typed Programs in Links CAN Go Wrong ! [1,2]

We can use the same channel twice by invoking the continuation twice.

```
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.
```

The problem is that the continuation has an unlimited type `r : Bool → ()`, which does not reflect the usage of the linear channel `ch`.

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

Our Solution

The previous type-and-effect system does not track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = do Choose; sender(ch)};
fun : ∀ (ρ::Row) . () {Choose:() ⇒ Bool | ρ}→ ()
```

Our Solution

The previous type-and-effect system does not track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = do Choose; sender(ch)};
fun : ∀ (ρ::Row) . () {Choose:() ⇒ Bool | ρ}→ ()
```

Use the operation signatures to track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = lindo Choose; sender(ch)};
fun : ∀ (ρ::Row°) . () {Choose:() =@ Bool | ρ}→ ()
```

Our Solution

The previous type-and-effect system does not track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = do Choose; sender(ch)};
fun : ∀ (ρ::Row) . () {Choose:() ⇒ Bool | ρ}→ ()
```

Use the operation signatures to track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = lindo Choose; sender(ch)};
fun : ∀ (ρ::Row°) . () {Choose:() =@ Bool | ρ}→ ()
```

Linear operations can only be handled by linear handlers.

```
links> handle
       ({ var ch = fork(receiver); var _ = lindo Choose; sender(ch) })
       { case <Choose =@ r> → r(true) }
```

Our Solution

The previous type-and-effect system does not track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = do Choose; sender(ch)};
fun : ∀ (ρ::Row) . () {Choose:() ⇒ Bool | ρ}→ ()
```

Use the operation signatures to track linear continuations.

```
links> fun(){var ch = fork(receiver); var _ = lindo Choose; sender(ch)};
fun : ∀ (ρ::Row°) . () {Choose:() =@ Bool | ρ}→ ()
```

Linear operations can only be handled by linear handlers.

```
links> handle
        ({ var ch = fork(receiver); var _ = lindo Choose; sender(ch) })
        { case <Choose =@ r> → r(true) }
```

Now we know the continuation has a linear type `r : Bool -@ ()`.

Core Ideas of Type Checking

Consider two sequenced computations

$$\underbrace{M}_{A\,!\,\{R_1\}}\;;\;\underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

We make sure R_1 is linear if N uses any linear resources.

Core Ideas of Type Checking

Consider two sequenced computations

$$\underbrace{M}_{A\,!\,\{R_1\}} ; \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

We make sure R_1 is linear if N uses any linear resources.

For instance,

$$\underbrace{\mathbf{do}\ \mathit{Choose}}_{\mathit{Bool}\,!\,\{\mathit{Choose}:\text{linear}\}} ; \underbrace{\mathit{sender}(\mathit{ch})}_{()\,!\,\{R_2\}} : B\,!\,\{R\}$$

Core Ideas of Type Checking

We make sure R_1 is linear if N uses any linear resources.

$$\underbrace{M}_{A\,!\,\{R_1\}} ; \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

What's the relationship between R_1, R_2 and R?

Core Ideas of Type Checking

We make sure R_1 is linear if N uses any linear resources.

$$\underbrace{M}_{A\,!\,\{R_1\}} ; \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

What's the relationship between R_1, R_2 and R?

- Row polymorphism: $R_1 = R_2 = R$
- Row subtyping: $R_1 \leqslant R, R_2 \leqslant R$
- Algebraic row subtyping: $R = R_1 \sqcup R_2$

The conventional effect system based on row polymorphism is too coarse for tracking linear continuations, because when N uses linear resources we only need to guarantee that operations in R_1 are linear.

Core Ideas of Type Inference

We make sure R_1 is linear if N uses any linear resources.

$$\underbrace{M}_{A\,!\,\{R_1\}} ; \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

However, we do not always know whether N uses any linear resources during the type inference.

Core Ideas of Type Inference

We make sure R_1 is linear if N uses any linear resources.

$$\underbrace{M}_{A\,!\,\{R_1\}} ; \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

However, we do not always know whether N uses any linear resources during the type inference.

Add linearity annotations to sequencing (as well as operation invocations and handler clauses).

We force R_1 to be linear when

$$\underbrace{M}_{A\,!\,\{R_1\}} ;^{\circ} \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

We force free variables in N to be unlimited when

$$\underbrace{M}_{A\,!\,\{R_1\}} ;^{\bullet} \underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\}$$

Core Ideas of Type Inference

Writing linearity annotations is tedious and harmful.

Core Ideas of Type Inference

Writing linearity annotations is tedious and harmful.

Qualified types / type inference with constraints.

$$\underbrace{M}_{A\,!\,\{R_1\}}\,;\,\underbrace{N}_{B\,!\,\{R_2\}} : B\,!\,\{R\} \mid (N \text{ contains free linear vars} \Rightarrow R_1 \text{ is linear})$$

We can also add the subtyping constraints.

$$B\,!\,\{R\} \mid (N \text{ contains free linear vars} \Rightarrow R_1 \text{ is linear}) \wedge (R_1 \leqslant R) \wedge (R_2 \leqslant R)$$

Conclusion

Our main contributions:

- F°_{eff}: a fine-grained call-by-value variant of *system F* with correct interaction between linear types and effect handlers.

Conclusion

Our main contributions:

- F°_{eff}: a fine-grained call-by-value variant of *system F* with correct interaction between linear types and effect handlers.
- An implementation of F°_{eff} in Links with *ML-style type inference* which requires a fair amount of linearity annotations.

Conclusion

Our main contributions:

- $\mathrm{F}^{\circ}_{\mathrm{eff}}$: a fine-grained call-by-value variant of *system F* with correct interaction between linear types and effect handlers.
- An implementation of $\mathrm{F}^{\circ}_{\mathrm{eff}}$ in Links with *ML-style type inference* which requires a fair amount of linearity annotations.
- $\mathrm{Q}^{\circ\leqslant}_{\mathrm{eff}}$: a *ML-style calculus* with linear types and effect subtyping based on *qualified types*. It requires no syntactic overheads and has better accuracy on tracking linear continuations.

Thank you!