# Soundly Handling Linearity

Wenhao Tang
The University of Edinburgh

TUPLE, the University of Edinburgh, 21 Feb 2024

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

## 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*

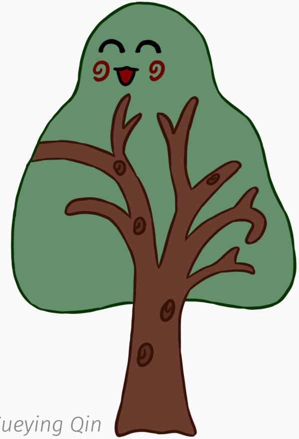
# Linear Types vs Effect Handlers

**linear types**

Rust, Haskell

Idris2, Granule

**effect handlers**



*Picture by Xueying Qin*

# Linear Types vs Effect Handlers

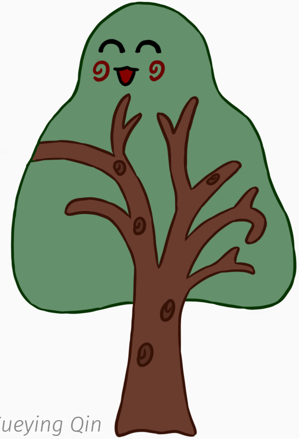
**linear types**

Rust, Haskell

Idris2, Granule

**effect handlers**

OCaml, WebAssembly



*Picture by Xueying Qin*

# Linear Types vs Effect Handlers

**linear types**

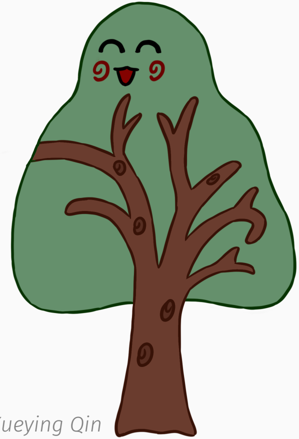
Rust, Haskell

Idris2, Granule

**effect handlers**

OCaml, WebAssembly

Eff, Koka, Frank, Effekt



*Picture by Xueying Qin*

# Linear Types vs Effect Handlers

**linear types**

Rust, Haskell

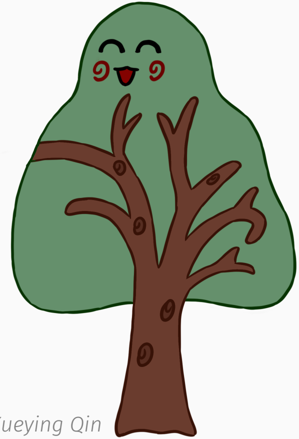
Idris2, Granule

**Links**

**effect handlers**

OCaml, WebAssembly

Eff, Koka, Frank, Effekt



*Picture by Xueying Qin*

# Linear Types vs Effect Handlers

| **linear types** | | **effect handlers** |
|---|---|---|
| Rust, Haskell | | OCaml, WebAssembly |
| Idris2, Granule | **Links** | Eff, Koka, Frank, Effekt |

*Picture by Xueying Qin*

# Linear Types vs Effect Handlers

| linear types | | effect handlers |
|---|---|---|
| Rust, Haskell | | OCaml, WebAssembly |
| Idris2, Granule | **Links** | Eff, Koka, Frank, Effekt |


*Picture by Xueying Qin*

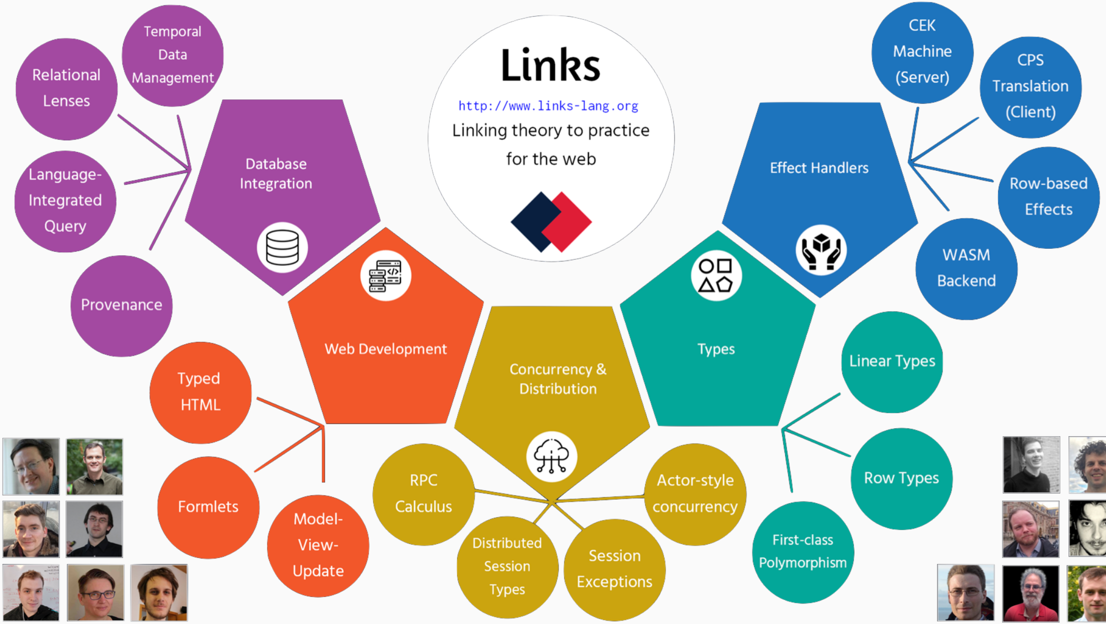
# Overview

In this talk, I will

- give a quick introduction to LINKS
- explain what are *linear types* (and *session types*)
- explain what are *algebraic effects and handlers*
- break LINKS by using them together
- fix LINKS by tracking *control-flow linearity*
- show how to further improve LINKS

Feel free to interrupt me at any time!

# Introduction to LINKS




Picture by Simon Fowler

# Functional Programming in Links

Links is a functional programming language.

# Functional Programming in LINKS

LINKS is a functional programming language.

```
> linx  --set=show_kinds=hide   # start REPL with a clean output of types
Welcome to Links version 0.9.8 (Burghmuirhead)
```

# Functional Programming in LINKS

LINKS is a functional programming language.

```
> linx  --set=show_kinds=hide   # start REPL with a clean output of types
Welcome to Links version 0.9.8 (Burghmuirhead)

links> 1+2+3;
6 : Int
```

# Functional Programming in LINKS

LINKS is a functional programming language.

```
> linx  --set=show_kinds=hide   # start REPL with a clean output of types
Welcome to Links version 0.9.8 (Burghmuirhead)

links> 1+2+3;
6 : Int

links> println("Hello world!");
Hello world!
() : ()
```

# Functional Programming in LINKS

LINKS is a functional programming language.

```
> linx  --set=show_kinds=hide   # start REPL with a clean output of types
Welcome to Links version 0.9.8 (Burghmuirhead)

links> 1+2+3;
6 : Int

links> println("Hello world!");
Hello world!
() : ()

links> fun inc(x) { x+1 };
inc = fun : (Int) -> Int
```

# Parametric Polymorphism in Links

Links supports *parametric polymorphism*.

A polymorphic function can be reused with different types.

## Parametric Polymorphism in Links

Links supports *parametric polymorphism*.

A polymorphic function can be reused with different types.

```
links> fun id(x) { x };
id = fun : (a) -> a        # as usual, the prefix ``forall a'' is omitted
```

# Parametric Polymorphism in LINKS

LINKS supports *parametric polymorphism*.

A polymorphic function can be reused with different types.

```
links> fun id(x) { x };
id = fun : (a) -> a          # as usual, the prefix ``forall a'' is omitted

links> id(42);               # instantiate a to Int
42 : Int
```

# Parametric Polymorphism in LINKS

LINKS supports *parametric polymorphism*.

A polymorphic function can be reused with different types.

```
links> fun id(x) { x };
id = fun : (a) -> a          # as usual, the prefix ``forall a'' is omitted

links> id(42);               # instantiate a to Int
42 : Int

links> id(true);             # instantiate a to Bool
true : Bool
```

# Parametric Polymorphism in Links

Links supports *parametric polymorphism*.

A polymorphic function can be reused with different types.

```
links> fun id(x) { x };
id = fun : (a) -> a         # as usual, the prefix ``forall a'' is omitted

links> id(42);              # instantiate a to Int
42 : Int

links> id(true);            # instantiate a to Bool
true : Bool

links> id("Hello world!"); # instantiate a to String
"Hello world!" : String
```

# Linear Types

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

Linear values cannot be discarded or duplicated, while unlimited values can.

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

Linear values cannot be discarded or duplicated, while unlimited values can.

Linear types statically guarantee this property.

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

Linear values cannot be discarded or duplicated, while unlimited values can.

Linear types statically guarantee this property.

```
links> typename Channel = !Int.End;  # an alias for a primitive linear type
Channel = !(Int).End
```

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

Linear values cannot be discarded or duplicated, while unlimited values can.

Linear types statically guarantee this property.

```
links> typename Channel = !Int.End;  # an alias for a primitive linear type
Channel = !(Int).End

links> fun dupLin(ch:Channel) { (ch, ch) };
Type error: Variable ch has linear type `Channel' but is used 2 times.
```

# Linear types restrict the usage of values

Some resources like file handles and communication channels are *linear*.

Linear values cannot be discarded or duplicated, while unlimited values can.

Linear types statically guarantee this property.

```
links> typename Channel = !Int.End;  # an alias for a primitive linear type
Channel = !(Int).End

links> fun dupLin(ch:Channel) { (ch, ch) };
Type error: Variable ch has linear type `Channel' but is used 2 times.

links> fun discardLin(ch:Channel) { 42 };
Type error: Variable ch has linear type `Channel' but is used 0 times.
```

# How does LINKS determine the linearity of types?

LINKS knows the linearity of primitive types by default.

# How does LINKS determine the linearity of types?

LINKS knows the linearity of primitive types by default.

- `Channel` is linear
- `Int`, `Bool` and `String` are unlimited

## How does Links determine the linearity of types?

Links knows the linearity of primitive types by default.

- `Channel` is linear
- `Int`, `Bool` and `String` are unlimited

Links knows the linearity of (most of) data types by looking at their components.

# How does LINKS determine the linearity of types?

LINKS knows the linearity of primitive types by default.

- `Channel` is linear
- `Int`, `Bool` and `String` are unlimited

LINKS knows the linearity of (most of) data types by looking at their components.

- `(Int, Channel)` is linear
- `(Int, Bool, String)` is unlimited

# How does Links determine the linearity of types?

Links knows the linearity of primitive types by default.

- `Channel` is linear
- `Int`, `Bool` and `String` are unlimited

Links knows the linearity of (most of) data types by looking at their components.

- `(Int, Channel)` is linear
- `(Int, Bool, String)` is unlimited

Links requires functions to be explicitly annotated with their linearity.

# How does LINKS determine the linearity of types?

LINKS knows the linearity of primitive types by default.

- `Channel` is linear
- `Int`, `Bool` and `String` are unlimited

LINKS knows the linearity of (most of) data types by looking at their components.

- `(Int, Channel)` is linear
- `(Int, Bool, String)` is unlimited

LINKS requires functions to be explicitly annotated with their linearity.

```
links> fun inc(x) { x+1 };
inc = fun : (Int) -> Int
links> linfun inc(x) { x+1 };
inc = fun : (Int) -@ Int       # called ``lollipop'' 🍭
```

# How does LINKS determine the linearity of type variables?

LINKS tells the linearity of type variables by their *kinds*.

```
> linx  # start REPL with kinds output
Welcome to Links version 0.9.8 (Burghmuirhead)
```

# How does LINKS determine the linearity of type variables?

LINKS tells the linearity of type variables by their *kinds*.

```
> linx  # start REPL with kinds output
Welcome to Links version 0.9.8 (Burghmuirhead)

links> fun id(x) { x };
id = fun : (a::Any) -> a::Any
a::Any can be instantiated to any types
```

# How does Links determine the linearity of type variables?

Links tells the linearity of type variables by their *kinds*.

```
> linx  # start REPL with kinds output
Welcome to Links version 0.9.8 (Burghmuirhead)

links> fun id(x) { x };
id = fun : (a::Any) -> a::Any
a::Any can be instantiated to any types

links> fun dup(x) { (x, x) };
dup = fun : (a) -> (a, a)
a::Unl (omitted by default) must be instantiated to unlimited types
```

# Session Types

# Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

## Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type
```

# Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End         # define an alias of a session type

sig sender       : (Channel) ~> ()  # Channel = !Int.End
```

## Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);     # c:!Int.End : send a value of type Int, then End
```

## Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);     # c:!Int.End : send a value of type Int, then End
  close(c')                 # c':End     : no further communication
}
```

# Session Types in Links

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);     # c:!Int.End : send a value of type Int, then End
  close(c')                 # c':End     : no further communication
}

sig receiver    : (~Channel) ~> () # dual of Channel = ?Int.End
```

# Session Types in Links

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);     # c:!Int.End : send a value of type Int, then End
  close(c')                 # c':End     : no further communication
}

sig receiver    : (~Channel) ~> () # dual of Channel = ?Int.End
fun receiver(c) {
  var (i, c') = receive(c); # c:?Int.End : receive a value of type Int, then End
```

# Session Types in Links

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End        # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);     # c:!Int.End : send a value of type Int, then End
  close(c')                 # c':End     : no further communication
}

sig receiver    : (~Channel) ~> () # dual of Channel = ?Int.End
fun receiver(c) {
  var (i, c') = receive(c); # c:?Int.End : receive a value of type Int, then End
  close(c');                # c':End     : no further communication
```

# Session Types in LINKS

Session types characterise communication protocols. Session types are linear.

```
typename Channel = !Int.End         # define an alias of a session type

sig sender      : (Channel) ~> ()  # Channel = !Int.End
fun sender(c)   {
  var c' = send(42, c);      # c:!Int.End : send a value of type Int, then End
  close(c')                  # c':End     : no further communication
}

sig receiver    : (~Channel) ~> () # dual of Channel = ?Int.End
fun receiver(c) {
  var (i, c') = receive(c); # c:?Int.End : receive a value of type Int, then End
  close(c');                # c':End     : no further communication
  println(intToString(i))
}
```

# Connect Sender and Receiver

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

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

# Well-typed programs in Links CANNOT go wrong ?

Let's try to hack Links by duplicating a linear channel!

# Well-typed programs in LINKS CANNOT go wrong ?

Let's try to hack LINKS by duplicating a linear channel!

```
links> { var c = fork(receiver); sender(c); sender(c); }; # simply use c twice
```

## Well-typed programs in Links CANNOT go wrong ?

Let's try to hack Links by duplicating a linear channel!

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

## Well-typed programs in LINKS CANNOT go wrong ?

Let's try to hack LINKS by duplicating a linear channel!

```
links> { var c = fork(receiver); sender(c); sender(c); }; # simply use c twice
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() }; # capture c in a function
```

# Well-typed programs in LINKS CANNOT go wrong ?

Let's try to hack LINKS by duplicating a linear channel!

```
links> { var c = fork(receiver); sender(c); sender(c); }; # simply use c twice
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() }; # capture c in a function
Type error: Variable c of linear type `!Int.End' is used in a non-linear function.
```

# Well-typed programs in LINKS CANNOT go wrong ?

Let's try to hack LINKS by duplicating a linear channel!

```
links> { var c = fork(receiver); sender(c); sender(c); }; # simply use c twice
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() }; # capture c in a function
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() }; # capture c in a linear function
```

# Well-typed programs in LINKS CANNOT go wrong ?

Let's try to hack LINKS by duplicating a linear channel!

```
links> { var c = fork(receiver); sender(c); sender(c); }; # simply use c twice
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() }; # capture c in a function
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() }; # capture c in a linear function
Type error: Variable f has linear type `() -@ ()' but is used 2 times.
```

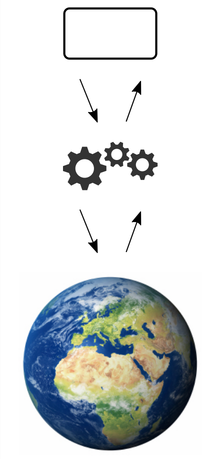
# Algebraic Effects and Handlers

# Effects

Programs must interact with their environment.

# Effects

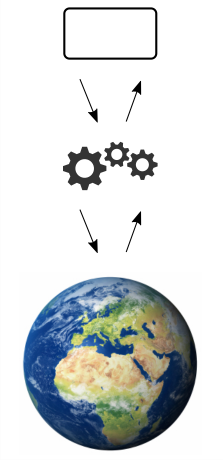
Programs must interact with their environment.



Picture from Sam Lindley

# Effects

Programs must interact with their environment.



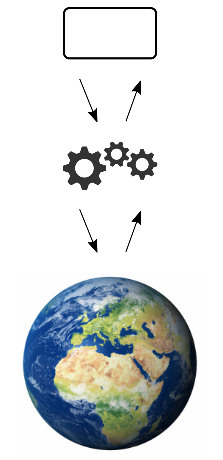
Picture from Sam Lindley

Effects are pervasive

- input/output
  user interaction
- concurrency
  web applications
- distribution
  cloud computing
- exceptions
  fault tolerance
- choice
  backtracking search

# Effects

Programs must interact with their environment.



Picture from Sam Lindley

Effects are pervasive

- input/output
  - user interaction
- concurrency
  - web applications
- distribution
  - cloud computing
- exceptions
  - fault tolerance
- choice
  - backtracking search

Typically ad hoc and hard-wired

# Algebraic Effects and Handlers

*Composable* and *customisable* user-defined interpretation of effects in general.

# Algebraic Effects and Handlers

*Composable* and *customisable* user-defined interpretation of effects in general.

Growing industrial interest

| | | |
|---|---|---|
| GitHub | `semantic` | Code analysis library ($>$ 25 million repositories) |
| | React | JavaScript UI library ($>$ 2 million websites) |
| Uber | Pyro | Statistical inference (10% ad spend saving) |

*Table from Sam Lindley*

# Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)
```

## Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()
```

# Defining Println

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()

links> handle (do Println("Hello world!")) {  # user-defined algebraic operation
```

# Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()

links> handle (do Println("Hello world!")) {  # user-defined algebraic operation
         case <Println(s) => r> ->            # s = "Hello world!", r = continuation
```

# Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()

links> handle (do Println("Hello world!")) {  # user-defined algebraic operation
         case <Println(s) => r> ->            # s = "Hello world!", r = continuation
           println(s);                        # print the parameter
```

# Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()

links> handle (do Println("Hello world!")) {  # user-defined algebraic operation
         case <Println(s) => r> ->              # s = "Hello world!", r = continuation
           println(s);                          # print the parameter
           r(())                                # resume the continuation
```

# Defining `Println`

The built-in `println` function in LINKS always prints its argument.

```
> linx --enable-handlers
Welcome to Links version 0.9.8 (Burghmuirhead)

links> println("Hello world!");
Hello world!
() : ()

links> handle (do Println("Hello world!")) {  # user-defined algebraic operation
         case <Println(s) => r> ->            # s = "Hello world!", r = continuation
           println(s);                        # print the parameter
           r(())                              # resume the continuation
       }
Hello world!
() : ()
```

# Customising `Println`

One of the advantages of algebraic effects and handlers is that we can give different interpretations of the same operation without changing its syntax.

# Customising Println

One of the advantages of algebraic effects and handlers is that we can give different interpretations of the same operation without changing its syntax.

```
links> handle (do Println("Hello world!")) {
         case <Println(s) => r> ->
           println("Print twice: " ^^ s ^^ " " ^^ s); r(())
       };
Print twice: Hello world! Hello world!
() : ()
```

## Customising `Println`

One of the advantages of algebraic effects and handlers is that we can give different interpretations of the same operation without changing its syntax.

```
links> handle (do Println("Hello world!")) {
         case <Println(s) => r> ->
           println("Print twice: " ^^ s ^^ " " ^^ s); r(())
       };
Print twice: Hello world! Hello world!
() : ()

links> handle (do Println("Hello world!")) {
         case <Println(s) => r> ->
           println("I don't want to print :("); r(())
       };
I don't want to print :(
() : ()
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }


links> handle (ndprinter())
       { case <Choose => r> -> r(true) };           # one-shot handler
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }


links> handle (ndprinter())
       { case <Choose => r> -> r(true) };           # one-shot handler
       # fun r(b) { var i = if (b) then 42 else 84; printInt(i) }
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }


links> handle (ndprinter())
       { case <Choose => r> -> r(true) };           # one-shot handler
       # fun r(b) { var i = if (b) then 42 else 84; printInt(i) }
42
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }


links> handle (ndprinter())
       { case <Choose => r> -> r(true) };           # one-shot handler
       # fun r(b) { var i = if (b) then 42 else 84; printInt(i) }
42

links> handle (ndprinter())
       { case <Choose => r> -> r(true); r(false) }; # multi-shot handler
```

# Implementing Nondeterminism

```
sig ndprinter   : () { Choose: () => Bool | _ }~> ()
# the function type is decorated with an effect type { Choose: () => Bool | _ }
# which means this function may use the Choose operation
# which takes no parameter and returns a boolean value
# _ is an anonymous effect variable which can be instantiated to other operations
fun ndprinter() { var i = if (do Choose) then 42 else 84; printInt(i) }


links> handle (ndprinter())
       { case <Choose => r> -> r(true) };           # one-shot handler
       # fun r(b) { var i = if (b) then 42 else 84; printInt(i) }
42

links> handle (ndprinter())
       { case <Choose => r> -> r(true); r(false) }; # multi-shot handler
42 84
```

# Breaking LINKS

# Well-typed programs in Links CAN go wrong !

We can break Links by duplicating a linear channel with multi-shot effect handlers!

## Well-typed programs in LINKS CAN go wrong !

We can break LINKS by duplicating a linear channel with multi-shot effect handlers!

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

## Well-typed programs in LINKS CAN go wrong !

We can break LINKS by duplicating a linear channel with multi-shot effect handlers!

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

# Well-typed programs in LINKS CAN go wrong !

We can break LINKS by duplicating a linear channel with multi-shot effect handlers!

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

continuation of Choose:
       fun r(b) { var x  = if (b) then 42 else 84;
                  var c' = send(x, c); # c is captured in the continuation
                  close(c') }          # it is closed when excuting r(true)
```

# Why doesn't LINKS reject us using r twice?

```
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')
}
```

```
continuation of Choose:
fun r(b) { var x  = if (b) then 42 else 84; var c' = send(x, c); close(c') }
```

## Why doesn't LINKS reject us using r twice?

```
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')
}

continuation of Choose:
fun r(b) { var x  = if (b) then 42 else 84; var c' = send(x, c); close(c') }
```

One point of view:

Conventional linear type systems only track ***value linearity***, i.e., linearity of primitive values, pairs, functions, etc. They already exist in the source code in the form of values. However, the continuation function `r` of `Choose` is dynamically created during evaluation.

# Why doesn't LINKS reject us using r twice?

```
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')
}

continuation of Choose:
fun r(b) { var x  = if (b) then 42 else 84; var c' = send(x, c); close(c') }
```

Another point of view:

Conventional linear type systems assume that the ***control flow*** goes normally from the beginning to the end. It only enters the continuation of `do Choose` once.

However, effect handlers allow the control flow to jump back to `do Choose`.

# Why doesn't LINKS reject us using r twice?

```
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')
}

continuation of Choose:
fun r(b) { var x  = if (b) then 42 else 84; var c' = send(x, c); close(c') }
```

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

- A *control-flow-linear* operation: the control flow must enter its cont exactly once.
- A *control-flow-unlimited* operation: the control flow may enter its cont any times.

# Fixing Links

# Tracking Control-Flow Linearity in Links

```
sig ndsender :
  (!Int.End)
    { Choose: () => Bool
    | _ }~>
  ()
fun ndsender(c) {
  # by default, the control-flow linearity is unlimited
  var x  = if (do Choose)
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

Ill-typed because we cannot use the linear variable `c` in a control-flow-unlimited environment after the control-flow-unlimited operation `Choose`.

# Tracking Control-Flow Linearity in LINKS

```
sig ndsender :
  (!Int.End)
    { Choose: () =@ Bool      # annotate Choose as control-flow linear 🍭
    | _ }~>
  ()
fun ndsender(c) {
  # by default, the control-flow linearity is unlimited
  var x  = if (do Choose)
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

# Tracking Control-Flow Linearity in LINKS

```
sig ndsender :
  (!Int.End)
    { Choose: () =@ Bool     # annotate Choose as control-flow linear 🍭
    | _ }~>
  ()
fun ndsender(c) {
  # by default, the control-flow linearity is unlimited
  var x  = if (lindo Choose) # invoke a control-flow-linear operation
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

# Tracking Control-Flow Linearity in LINKS

```
sig ndsender :
  (!Int.End)
    { Choose: () =@ Bool      # annotate Choose as control-flow linear 🍭
    | _ }~>
  ()
fun ndsender(c) {
  xlin;                       # switch the control-flow linearity to linear
  var x  = if (lindo Choose) # invoke a control-flow-linear operation
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

# Tracking Control-Flow Linearity in Links

```
sig ndsender :
  (!Int.End)
    { Choose: () =@ Bool      # annotate Choose as control-flow linear 🍭
    | _::Lin }~>              # require other potential operations to be linear
  ()
fun ndsender(c) {
  xlin;                       # switch the control-flow linearity to linear
  var x  = if (lindo Choose) # invoke a control-flow-linear operation
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

# Tracking Control-Flow Linearity in LINKS

```
sig ndsender :
  (!Int.End)
    { Choose: () =@ Bool      # annotate Choose as control-flow linear 🍭
    | _::Lin }~>              # require other potential operations to be linear
  ()
fun ndsender(c) {
  xlin;                       # switch the control-flow linearity to linear
  var x  = if (lindo Choose) # invoke a control-flow-linear operation
           then 42 else 84;
  var c' = send(x, c);
  close(c')
}
```

Well-typed since we are using a linear variable `c` and a control-flow-linear operation `Choose` in a control-flow-linear environment.

# Back to the Full Example

```
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.
```

# Back to the Full Example

```
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: ... =@ does not match => ...
```

# Back to the Full Example

```
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) };
       # use =@ for handler clauses of control-flow-linear operations
Type Error: Variable r has linear type but is used 2 times.
```

# Back to the Full Example

```
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) };
       # use =@ for handler clauses of control-flow-linear operations
Type Error: Variable r has linear type but is used 2 times.
```

Well-typed programs cannot go wrong!

# Beyond LINKS

# Restriction of Linear Types and Control-Flow Linearity in Links

We lose *principal types*. As a result, we need to have different versions of (almost) the same function with different types, which breaks modularity and reusability.

# Restriction of Linear Types and Control-Flow Linearity in Links

We lose *principal types*. As a result, we need to have different versions of (almost) the same function with different types, which breaks modularity and reusability.

Consider the verbose identity function

```
fun verboseId(x) {do Print("id"); x}
```

# Restriction of Linear Types and Control-Flow Linearity in Links

We lose *principal types*. As a result, we need to have different versions of (almost) the same function with different types, which breaks modularity and reusability.

Consider the verbose identity function

```
fun verboseId(x) {do Print("id"); x}
```

Without linear types, we only need one version of it with the type

```
sig verboseId : (a) { Print : (String) => () | _ }-> a
fun verboseId(x) {do Print("id"); x}
```

# Restriction of Linear Types and Control-Flow Linearity in Links

We lose *principal types*. As a result, we need to have different versions of (almost) the same function with different types, which breaks modularity and reusability.

Consider the verbose identity function

```
fun verboseId(x) {do Print("id"); x}
```

With linear types, we have two versions

```
sig verboseId : (a::Any) { Print : (String) => () | _ }-> a::Any
fun verboseId(x) {do Print("id"); x}
sig verboseId : (a::Any) { Print : (String) => () | _ }-@ a::Any
linfun verboseId(x) {do Print("id"); x}
```

# Restriction of Linear Types and Control-Flow Linearity in Links

We lose *principal types*. As a result, we need to have different versions of (almost) the same function with different types, which breaks modularity and reusability.

Consider the verbose identity function

```
fun verboseId(x) {do Print("id"); x}
```

Further with control-flow linearity, we have six versions

```
sig verboseId : (a) { Print : (String) => () | _ }-> a
fun verboseId(x) {do Print("id"); x}
sig verboseId : (a) { Print : (String) =@ () | _ }-> a
fun verboseId(x) {lindo Print("id"); x}
sig verboseId : (a::Any) { Print : (String) =@ () | _::Lin }-> a::Any
fun verboseId(x) {xlin; lindo Print("id"); x}
linfun ... linfun ... linfun ...
```

# Principal Types with Constraints

We can restore principal types in LINKS using constraints / qualified types

```
sig verboseId : a { Print : (String) =>^ϕ () | ρ }->^ϕ' a with (a ≤ ϕ, a ≤ ρ)
fun verboseId(x) {do Print("id"); x}
```

# Principal Types with Constraints

We can restore principal types in LINKS using constraints / qualified types

```
sig verboseId : a { Print : (String) =>ϕ () | ρ }->ϕ' a with (a ≤ ϕ, a ≤ ρ)
fun verboseId(x) {do Print("id"); x}
```

`->`$^{\phi'}$ can be instantiated to either `->` or `-@`

`=>`$^{\phi}$ can be instantiated to either `=>` or `=@` satisfying the condition that when `a` is a linear type, it must be `=@`

$\rho$ can either have kind `Lin` or `Any` satisfying the condition that when `a` is a linear type, it must have kind `Lin`

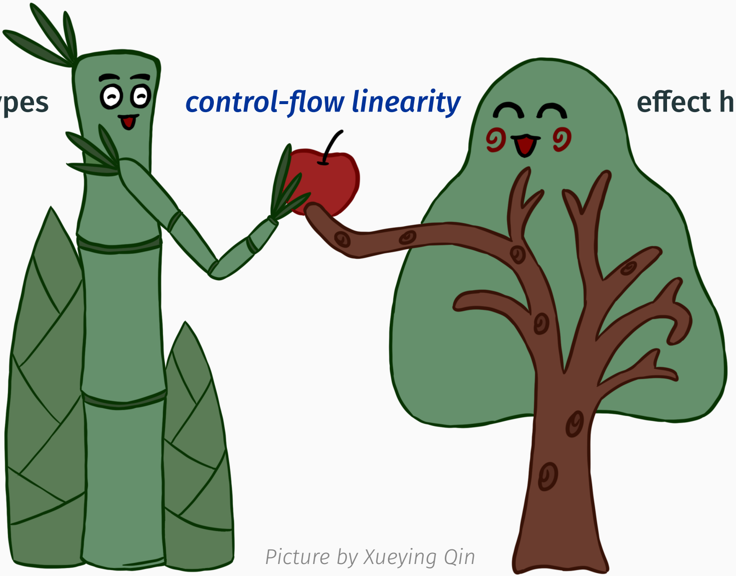
# More in the Paper

$F^{\circ}_{\text{eff}}$
- *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)

$Q^{\circ}_{\text{eff}}$
- *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*