Session Types

- Session types specify communication **protocols** and statically guarantee that concurrent programs respect them.

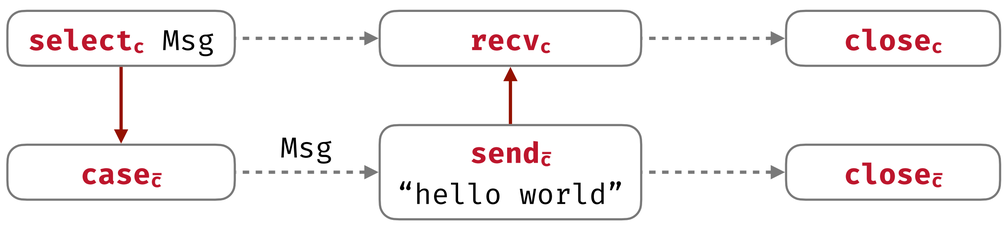
```
S ::= !A.S          send value of type A then continue as S
    | ?A.S          receive value of type A then continue as S
    | ⊕{Lᵢ : Sᵢ}    select label Lᵢ then continue as Sᵢ
    | &{Lᵢ : Sᵢ}    offer branches Lᵢ with continuations Sᵢ
    | End           finished
```

Session-Typed Communication

- Concurrent programs communicate via **dual** channels:

```
c : ⊕{Msg : ?String.End}
c̄ : &{Msg : !String.End}

fork [c : ⊕{Msg : ?String.End}]
     (selectc Msg; val x = recvc; print x; closec )
     (case c̄ { Msg ⟼ send c̄ "hello world"; close c̄ })
➡ hello world
```



Algebraic Effects and Handlers

- Algebraic effects and handlers allow programmers to define, customise, and compose a range of crucial programming features modularly.

```
handle (do Ask + do Ask) { case Ask r ⟼ r 21 }
➡ 21 + handle (do Ask) { case Ask r ⟼ r 21 }
➡ 21 + 21
➡ 42

handle (handle (if (do Choose) then 7 else do Ask)
        { case Choose r ⟼ r true + r false })
{ case Ask r ⟼ r 35 }
➡ handle ( if true  then 7 else do Ask
         + if false then 7 else do Ask)
  { case Ask r ⟼ r 35 }
➡ handle (7 + do Ask) { case Ask r ⟼ r 35 }
➡ 42
```

Effect Types

- For soundness, the type system is usually extended with an **effect system** to track the effects used by programs.

```
f : () {Ask : () ⇒ Int}→ Int
f _ = do Ask + do Ask

g : () {Choose : () ⇒ Bool; Ask : () ⇒ Int}→ Int
g _ = if (do Choose) then 7 else do Ask
```

Session-Typed Effect Handlers

Wenhao Tang
The University of Edinburgh

Extended Abstract
for POPL'24 SRC

Protocols for Effects and Handlers

- Inspired by session types, we use **protocols** specified by **behavioural effect types** to describe the interaction between effects and handlers.
- A naive protocol for one unidirectional effect and deep handler:

```
c : ⊕{Ask : () ⇒ Int}
    invoke Ask (any times) on the channel c
c̄ : &{Ask : () ⇒ Int}
    (deeply) handle Ask on the dual channel c̄
```

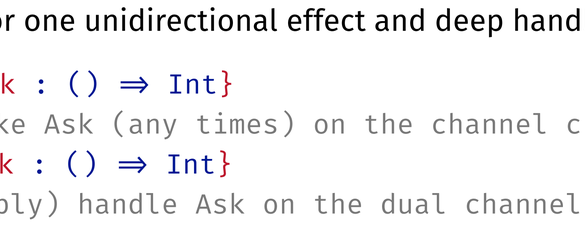
- A protocol for two effects:

```
c : ⊕{Choose : () ⇒ Bool; Ask : () ⇒ Int}
    invoke Choose and Ask
c̄ : &{Choose : () ⇒ Bool; Ask : () ⇒ Int}
    handle Choose and Ask
```

Handling Effects on the Dual Channel

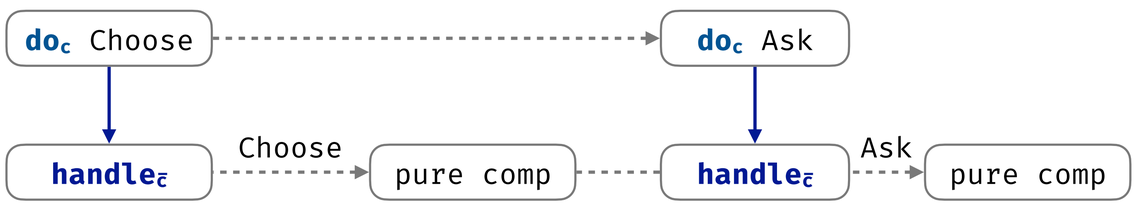
- Inspired by the channels, effects must and can only be handled in the **dual** channel. Different channels never interfere with each other.

```
fork [c : ⊕{Ask : () ⇒ Int}]
  (doc Ask + doc Ask)
  (handle c̄ oc̄ { case Ask r ⟼ r 21 })
```



- Channels generalise *handler names* since multiple handlers can share one channel:

```
fork [c : ⊕{Choose : () → Bool; Ask : () → Int}]
  (if (doc Choose) then 7 else doc Ask)
  (handle c̄ (handle c̄ oc̄ { case Choose r ⟼ r true + r false })
   { case Ask r ⟼ r 35 })
```



Bidirectional Interaction of Effects and Handlers

- Behavioural effect types can easily encode **bidirectional effects**.
- 2-layer bidirectional interaction:

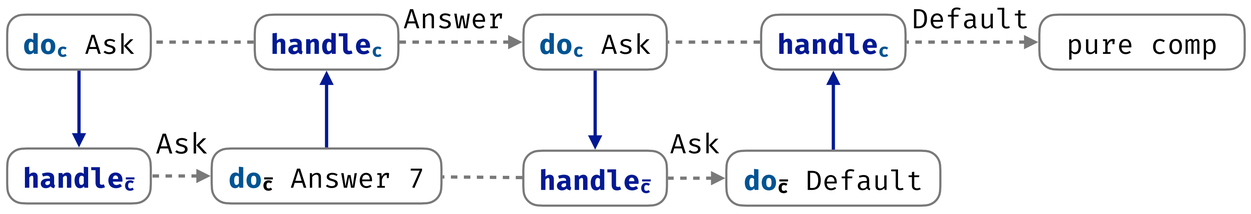
```
c : E = ⊕{Ask : () → Int : &{Raise : String → ⊥}}
    invoke Ask, then must handle Raise outside
c̄ : E = &{Ask : () → Int : ⊕{Raise : String → ⊥}}
    handle Ask, then can invoke Raise when resuming

fork [c : E]
  (handlec (40 + doc Ask) {case Raise s r ⟼ print ("Error: " ++ s)})
  (handle c̄ oc̄ { case Ask r ⟼ r {do c̄ Raise "Don't ask me."} })
```

- 4-layer bidirectional interaction:

```
c : E = ⊕{Ask: &{Answer: ⊕{Ask: &{Default}}}}

fork [c : E]
  (handlec (doc Ask) { case Answer x r ⟼ r {
     handle (x + doc Ask)
     { case Default r ⟼ r 35 }
   } })
  (handle c̄ oc̄ { case Ask r ⟼ r {
     handle (do c̄ Answer 7)
     { case Ask r ⟼ r {do c̄ Default}}
   } })
```



Future Work

- Pretty much work in progress.
- Some interesting or necessary things we are working on:
  - first-class or second-class ***linear channels*** with ***shallow handlers***
  - truly concurrency semantics
  - "standard" metatheory, polymorphism, subtyping, recursive types, etc