

# **Types and Effects for Deadlock-Free Higher-Order Concurrent Programs**

Luca Padovani and Luca Novara

Dipartimento di Informatica, Torino

NII Shonan Meeting – 2014

$c?(x).P$

$c?(x).P$

the matching  $c!v$  is not in here

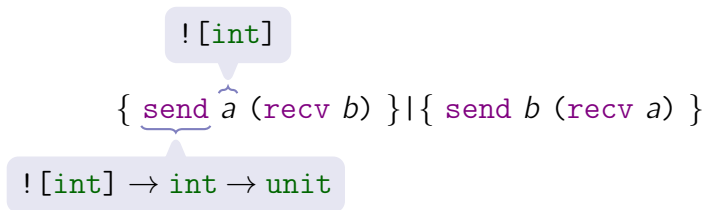
# A deadlock in CML [Reppy, 1999]

$$\{ \text{send } a \text{ (recv } b) \} | \{ \text{send } b \text{ (recv } a) \}$$

## Ingredients

- call-by-value  $\lambda$ -calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

# A deadlock in CML [Reppy, 1999]



## Ingredients

- call-by-value  $\lambda$ -calculus
- open, send, recv, fork
- **linear** channels

# A deadlock in CML [Reppy, 1999]

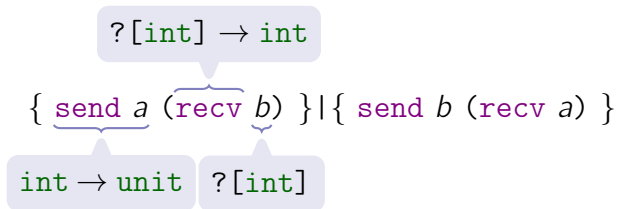
$\{ \underbrace{\text{send } a}_{\text{int} \rightarrow \text{unit}} (\text{recv } b) \} | \{ \text{send } b (\text{recv } a) \}$

`int → unit`

## Ingredients

- call-by-value  $\lambda$ -calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

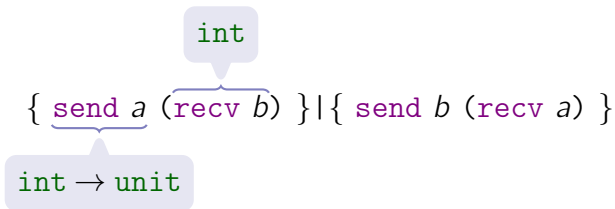
# A deadlock in CML [Reppy, 1999]



## Ingredients

- call-by-value  $\lambda$ -calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

# A deadlock in CML [Reppy, 1999]



## Ingredients

- call-by-value  $\lambda$ -calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels



# Outline

- ① Types
- ② Examples
- ③ Conclusions

# Outline

① Types

② Examples

③ Conclusions

# Channel levels

$\{ \text{send } a^n \text{ (recv } b^m) \} | \{ \text{send } b^m \text{ (recv } a^n) \}$



## Channel levels **in types**

$$\{ \text{send } a \text{ (recv } b) \} | \{ \text{send } b \text{ (recv } a) \}$$

- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

# Channel levels **in types**

`! [int]n`

`{ send ^a (recv b) } | { send b (recv a) }`

`! [int]n → int → unit`

- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

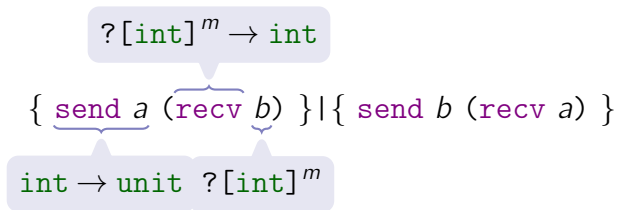
# Channel levels **in types**

{ send a (recv b) } | { send b (recv a) }

int → unit

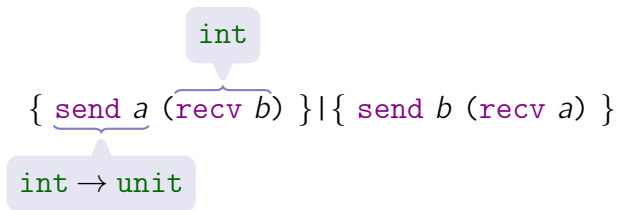
- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

## Channel levels **in types**



- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

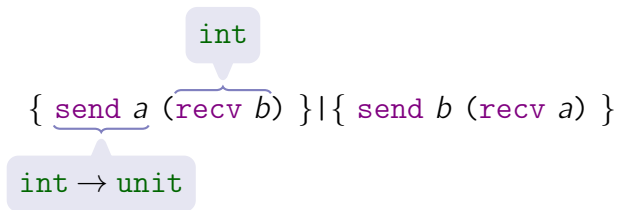
## Channel levels **in types**



- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999



# Channel levels **in types**



- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

## Channel levels **in types and effects**

{ send a (recv b) } | { send b (recv a) }

# Channel levels in types and effects

$! [\text{int}]^n \& \perp$

$\{ \underbrace{\text{send } \hat{a}}_{\text{}} (\text{recv } b) \} | \{ \text{send } b (\text{recv } a) \}$

$! [\text{int}]^n \rightarrow \text{int} \rightarrow^n \text{unit} \& \perp$

# Channel levels **in types and effects**

$\{ \underbrace{\text{send } a}_{\text{effect}} (\text{recv } b) \} | \{ \text{send } b (\text{recv } a) \}$

`int  $\rightarrow^n$  unit &  $\perp$`

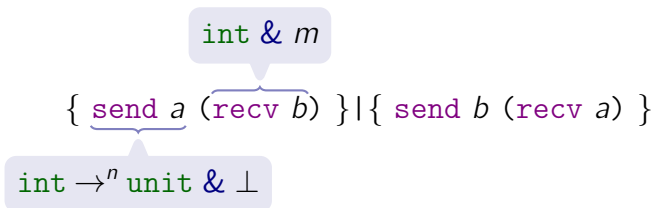
# Channel levels **in types and effects**

$?[\text{int}]^m \rightarrow^m \text{int} \ \& \ \perp$

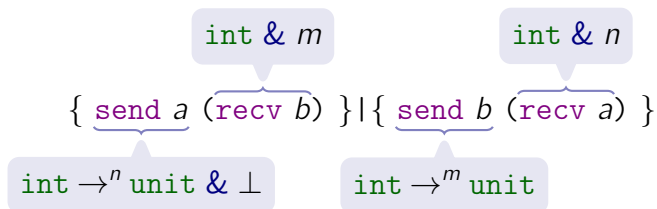
$\{ \text{send } a \ (\overbrace{\text{recv } b}) \} \mid \{ \text{send } b \ (\text{recv } a) \}$

$?[\text{int}]^m \ \& \ \perp$

# Channel levels **in types and effects**




# Channel levels in types and effects



## More on arrow types

$f \equiv \lambda x. (\text{send } a^m \ x; \text{send } b^n \ x)$



Which type for  $f$ ?


$f : \text{int} \rightarrow^m \text{unit}$

$f : \text{int} \rightarrow^n \text{unit}$



## More on arrow types

$f \equiv \lambda x. (\text{send } a^m \ x; \text{send } b^n \ x)$



Which type for  $f$ ?

$f : \text{int} \rightarrow^m \text{unit}$

$f : \text{int} \rightarrow^n \text{unit}$


$\text{int} \ \& \ n$

$\{ (f \ 3); \text{recv } b \} \mid \{ \text{recv } a \}$

$\text{int} \ \& \ m$

## More on arrow types

$f \equiv \lambda x. (\text{send } a^m x; \text{send } b^n x)$



Which type for  $f$ ?

$f : \text{int} \rightarrow^m \text{unit}$

$\text{int} \& n$

$\{ (f \ 3); \text{recv } b \} \mid \{ \text{recv } a \}$

$\text{int} \& m$

$f : \text{int} \rightarrow^n \text{unit}$

$\text{int} \& m$

$\{ f (\text{recv } a) \} \mid \{ \text{recv } b \}$

$\text{int} \rightarrow^n \text{unit} \& \perp$

# Typing abstractions

$$\frac{\Gamma, x : t \vdash e : s \& \rho}{\Gamma \vdash \lambda x. e : t \rightarrow^{|\Gamma|, \rho} s \& \perp}$$

$$\vdash \lambda x. x \quad : \text{int} \rightarrow^{\top, \perp} \text{int}$$

# Typing abstractions

$$\frac{\Gamma, x : t \vdash e : s \& \rho}{\Gamma \vdash \lambda x. e : t \rightarrow^{|\Gamma|, \rho} s \& \perp}$$

$$\begin{array}{ll} \vdash \lambda x. x & : \text{int} \rightarrow^{\top, \perp} \text{int} \\ a : ![ \text{int} ]^n \vdash \lambda x. (x, a) & : \text{int} \rightarrow^{n, \perp} \text{int} \times ![ \text{int} ]^n \end{array}$$

# Typing abstractions

$$\frac{\Gamma, x : t \vdash e : s \& \rho}{\Gamma \vdash \lambda x. e : t \rightarrow^{|\Gamma|, \rho} s \& \perp}$$

$$\begin{array}{ll} \vdash \lambda x. x & : \text{int} \rightarrow^{\top, \perp} \text{int} \\ a : ![ \text{int} ]^n \vdash \lambda x. (x, a) & : \text{int} \rightarrow^{n, \perp} \text{int} \times ![ \text{int} ]^n \\ \vdash \lambda x. (\text{send } x \ 3) & : ![ \text{int} ]^n \rightarrow^{\top, n} \text{unit} \end{array}$$

# Typing abstractions

$$\frac{\Gamma, x : t \vdash e : s \& \rho}{\Gamma \vdash \lambda x. e : t \rightarrow^{|\Gamma|, \rho} s \& \perp}$$

$$\vdash \lambda x. x \quad : \text{int} \rightarrow^{\top, \perp} \text{int}$$

$$a : ![\text{int}]^n \vdash \lambda x. (x, a) \quad : \text{int} \rightarrow^{n, \perp} \text{int} \times ![\text{int}]^n$$

$$\vdash \lambda x. (\text{send } x \ 3) \quad : ![\text{int}]^n \rightarrow^{\top, n} \text{unit}$$

$$a : ?[\text{int}]^n \vdash \lambda x. (\text{recv } a+x) \quad : \text{int} \rightarrow^{n, n} \text{int}$$

# Typing applications

$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho, \sigma} s \ \& \ \tau_1 \quad \Gamma_2 \vdash e_2 : t \ \& \ \tau_2 \quad \tau_1 < |\Gamma_2| \quad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \ \& \ \sigma \vee \tau_1 \vee \tau_2}$$

# Typing applications

$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho, \sigma} s \ \& \ \tau_1 \quad \Gamma_2 \vdash e_2 : t \ \& \ \tau_2 \quad \tau_1 < |\Gamma_2| \quad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \ \& \ \sigma \vee \tau_1 \vee \tau_2}$$

$\vdash (\lambda x. x) 3$





# Typing applications

$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho, \sigma} s \& \tau_1 \quad \Gamma_2 \vdash e_2 : t \& \tau_2 \quad \tau_1 < |\Gamma_2| \quad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \& \sigma \vee \tau_1 \vee \tau_2}$$

$\vdash (\lambda x. x) 3$  ☺

$a : ?[t]^n \vdash (\lambda x. x) (\text{recv } a)$  ☺

# Typing applications

$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho, \sigma} s \ \& \ \tau_1 \quad \Gamma_2 \vdash e_2 : t \ \& \ \tau_2 \quad \tau_1 < |\Gamma_2| \quad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \ \& \ \sigma \vee \tau_1 \vee \tau_2}$$

$\vdash (\lambda x. x) \ 3$  😊

$a : ?[t]^n \vdash (\lambda x. x) \ (\text{recv } a)$  😊

$a : ?[t]^n \vdash (\lambda x. (x, a)) \ (\text{recv } a)$  😞

# Typing applications

$$\frac{\Gamma_1 \vdash e_1 : t \rightarrow^{\rho, \sigma} s \ \& \ \tau_1 \quad \Gamma_2 \vdash e_2 : t \ \& \ \tau_2 \quad \tau_1 < |\Gamma_2| \quad \tau_2 < \rho}{\Gamma_1 + \Gamma_2 \vdash e_1 e_2 : s \ \& \ \sigma \vee \tau_1 \vee \tau_2}$$

$\vdash (\lambda x. x) \ 3$  😊

$a : ?[t]^n \vdash (\lambda x. x) \ (\text{recv } a)$  😊

$a : ?[t]^n \vdash (\lambda x. (x, a)) \ (\text{recv } a)$  😞

$a : ?[t \rightarrow t]^0, b : ?[t]^1 \vdash (\text{recv } a) \ (\text{recv } b)$  😊

# Properties

## Theorem (soundness)

- ① well-typed, *closed* programs are **deadlock free**
- ② well-typed, *convergent* programs typed with *discrete* levels eventually **use** all of their channels

`send a (rec x x)`

(some sensible programs require *dense* levels)

## Theorem (expressiveness)

Most interaction protocols describable by a (multiparty) session type are realizable by (a set of) well-typed processes

# Outline

① Types

② Examples

③ Conclusions

## Example: parallel Fibonacci

```
let rec fibo n c =  
  if  $n \leq 1$  then send c 1  
  else let a = open() in  
        let b = open() in  
        fork  $\lambda().(\text{fibo } (n - 1) a )$ ;  
        fork  $\lambda().(\text{fibo } (n - 2) b )$ ;  
        send c (recv a + recv b )
```

`fibo : int → ![int] → unit`

## Example: parallel Fibonacci

```
let rec fibo n ci =  
  if n ≤ 1 then send ci 1  
  else let ai-2 = open() in  
        let bi-1 = open() in  
        fork λ().(fibo (n - 1) ai-2);  
        fork λ().(fibo (n - 2) bi-1);  
        send ci (recv ai-2 + recv bi-1)
```

$\text{fibo} : \forall i.\text{int} \rightarrow ![\text{int}]^i \rightarrow^{\text{T},i} \text{unit}$

## Example: parallel Fibonacci

```
let rec fibo n ci =  
  if n ≤ 1 then send ci 1  
  else let ai-2 = open() in  
        let bi-1 = open() in  
        fork λ().(fibo (n - 1) ai-2);  
        fork λ().(fibo (n - 2) bi-1);  
        send ci (recv ai-2 + recv bi-1)
```

$\text{fibo} : \forall i. \text{int} \rightarrow ![\text{int}]^i \rightarrow^{\top, i} \text{unit}$

- type inference for polymorphic recursion is **undecidable**
- ... but is **decidable** when limited to effects  
[Amtoft, Nielson, Nielson, 99]



## Example: linear forwarder

```
let forward x y = send y (recv x)
```

```
forward :  $\forall \alpha. \quad ?[\alpha] \rightarrow ![\alpha] \rightarrow \text{unit}$ 
```

## Example: linear forwarder

```
let forward  $x^i$   $y^j$  = send  $y^j$  (recv  $x^i$ )
```

```
forward :  $\forall i, j. \forall \alpha. \quad ?[\alpha]^i \rightarrow ![\alpha]^j \rightarrow^{i,j} \text{unit}$ 
```

## Example: linear forwarder

```
let forward  $x^i y^j = \text{send } y^j (\text{recv } x^i)$ 
```

```
forward :  $\forall i, j. \forall \alpha. (i < j) \Rightarrow ?[\alpha]^i \rightarrow ![\alpha]^j \rightarrow^{i,j} \text{unit}$ 
```

## Example: persistent forwarder

```
let rec copy x y =  
  let (v, c) = recv x in  
  let d = open () in  
  send y (v, d);  
  copy c d
```

```
type In  $\alpha = ?[\alpha \times \text{In } \alpha]$   
type Out  $\alpha = ![\alpha \times \text{In } \alpha]$ 
```

type of x

type of y

```
copy :  $\forall \alpha. \text{In } \alpha \rightarrow \text{Out } \alpha \rightarrow \text{unit}$ 
```

## Example: persistent forwarder

```
let rec copy xi yj =  
  let (v, c) = recv xi in  
  let d = open () in  
  send yj (v, d);  
  copy c d
```

```
type In i α = ?[α × In i α]i  
type Out j α = ![α × In j α]j
```

$\text{copy} : \forall i, j. \forall \alpha. \text{In } i \ \alpha \rightarrow \text{Out } j \ \alpha \rightarrow^i \text{unit}$

## Example: persistent forwarder

```
let rec copy xi yj =  
  let (v, c) = recv xi in receive from x...  
  let d = open () in  
  send yj (v, d); ... then send on y  
  copy c d
```

```
type In i α = ?[α × In i α]i  
type Out j α = ![α × In j α]j
```

$\text{copy} : \forall i, j. \forall \alpha. (i < j) \Rightarrow \text{In } i \ \alpha \rightarrow \text{Out } j \ \alpha \rightarrow^i \text{ unit}$

## Example: persistent forwarder

```
let rec copy  $x^i$   $y^j$  =  
  let ( $v$ ,  $c^{i+1}$ ) = recv  $x^i$  in  $c$  received from  $x$   
  let  $d$  = open () in  
  send  $y^j$  ( $v$ ,  $d$  );  
  copy  $c^{i+1}$   $d$ 
```

```
type In  $i$   $\alpha$  = ?[ $\alpha$   $\times$  In ( $i+1$ )  $\alpha$ ] $i$  non-regular type  
type Out  $j$   $\alpha$  = ![ $\alpha$   $\times$  In  $j$   $\alpha$ ] $j$ 
```

$\text{copy} : \forall i, j. \forall \alpha. (i < j) \Rightarrow \text{In } i \ \alpha \rightarrow \text{Out } j \ \alpha \rightarrow^i \text{unit}$

## Example: persistent forwarder

```
let rec copy  $x^i$   $y^j$  =  
  let ( $v$ ,  $c^{i+1}$ ) = recv  $x^i$  in  
  let  $d^{j+1}$  = open () in  
  send  $y^j$  ( $v$ ,  $d^{j+1}$ );  $d$  sent on  $y$   
  copy  $c^{i+1}$   $d^{j+1}$ 
```

```
type In  $i$   $\alpha$  = ?[ $\alpha \times$  In ( $i + 1$ )  $\alpha$ ] $i$   
type Out  $j$   $\alpha$  = ![ $\alpha \times$  In ( $j + 1$ )  $\alpha$ ] $j$  non-regular type
```

$\text{copy} : \forall i, j. \forall \alpha. (i < j) \Rightarrow \text{In } i \ \alpha \rightarrow \text{Out } j \ \alpha \rightarrow^i \text{ unit}$



## Example: persistent forwarder

```
let rec copy  $x^i$   $y^j$  =  
  let ( $v$ ,  $c^{i+1}$ ) = recv  $x^i$  in  
  let  $d^{j+1}$  = open () in  
  send  $y^j$  ( $v$ ,  $d^{j+1}$ );  
  copy  $c^{i+1}$   $d^{j+1}$ 
```

```
type In  $i$   $\alpha$  = ?[ $\alpha$   $\times$  In ( $i+1$ )  $\alpha$ ] $i$   
type Out  $j$   $\alpha$  = ![ $\alpha$   $\times$  In ( $j+1$ )  $\alpha$ ] $j$ 
```

$\text{copy} : \forall i, j. \forall \alpha. (i < j) \Rightarrow \text{In } i \ \alpha \rightarrow \text{Out } j \ \alpha \rightarrow^{i, \top} \text{unit}$

tail applications only!

## Example: filter

```
let rec filter p x y =  
  let (v, c) = recv x in  
  if p v then  
    let d = open () in  
    fork λ().(send y (v, d));  
    filter p c d  
  else  
    filter p c y
```

- communication on  $y$  depends on **data** from  $x$
- well typed only with **dense** levels

# Outline

① Types

② Examples

③ Conclusions

# Wrap up

- session type systems  $\Rightarrow$  each session is deadlock free
- deadlock freedom  $\Rightarrow$  inter-channel dependencies

## Question

How hard is it to adapt a type system for deadlock freedom to a real-world programming language?

## Answer

Doable, but full integration requires somewhat advanced features

- effect polymorphism + polymorphic recursion
- effect constraints
- non-regular types