

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

Luca Padovani and Luca Novara

Dipartimento di Informatica, Torino

BETTY Meeting, Grenoble, 2014

On the structure of programs

$$\frac{\Gamma, x : t \vdash P \quad n < |\Gamma|}{\Gamma, u : ?[t]^n \vdash u?(x).P}$$

On the structure of programs

$$\frac{\Gamma, x : t \vdash P \quad n < |\Gamma|}{\Gamma, u : ?[t]^n \vdash u?(x).P}$$

Ingredients

- λ -calculus
- linear channels
- `open`, `send`, `recv`, `fork`

- Reppy, *Concurrent programming in ML*,
Cambridge University Press, 99

A deadlock in CML [Reppy, 99]

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

A deadlock in CML [Reppy, 99]

![int]

{ send [~]a (recv b) } | { send b (recv a) }

![int] → int → unit

A deadlock in CML [Reppy, 99]

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

`int → unit`

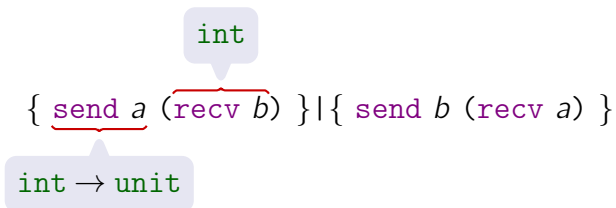
A deadlock in CML [Reppy, 99]

?[int] → int

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

int → unit ?[int]

A deadlock in CML [Reppy, 99]



Outline

- ① Motivation
- ② Technique
- ③ Challenges
- ④ Conclusion

Detecting deadlocks with priorities

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

Detecting deadlocks with priorities **in types**

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

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

Detecting deadlocks with priorities **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

Detecting deadlocks with priorities **in types**

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

int → unit

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

Detecting deadlocks with priorities **in types**

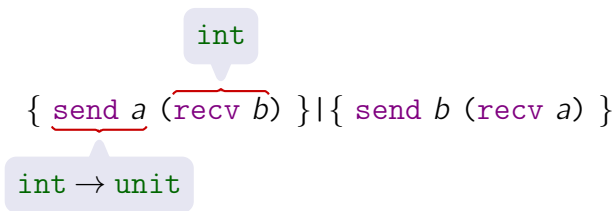
`?[int]m → int`

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

`int → unit` `?[int]m`

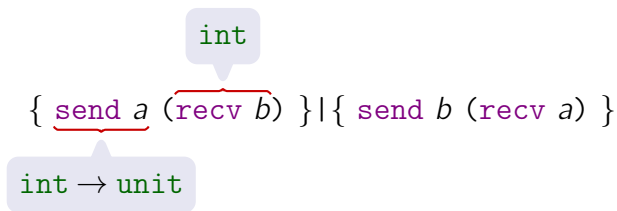
- Amtoft, Nielson, Nielson, *Type and Effect Systems: Behaviours for Concurrency*, 1999

Detecting deadlocks with priorities **in types**



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

Detecting deadlocks with priorities **in types**



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

One more try

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

One more try

`! [int]n & ⊥`

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

`! [int]n → int →n unit & ⊥`

One more try

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

`int \rightarrow^n unit & \perp`

One more try

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

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

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

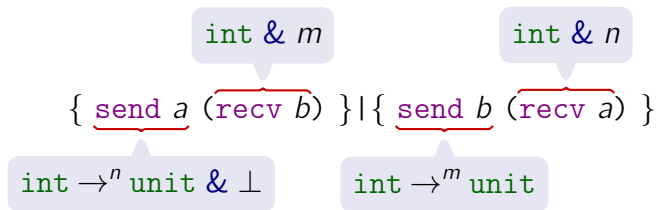
One more try

`int & m`

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


`int \rightarrow^n unit & \perp`

One more try



Priorities vs effects

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




Which type for f ?

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

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

Priorities vs effects

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



Which type for f ?

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

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


$\text{int} \& n$

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

$\text{int} \& m$

Priorities vs effects

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



Which type for f ?

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

$\text{int} \& n$

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

$\text{int} \& m$

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

$\text{int} \& m$

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

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

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

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

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

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

Typing abstractions

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

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

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

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

Typing abstractions

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

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

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

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

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

Typing abstractions

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

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

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

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

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

$a : ![\text{int}]^n \vdash \lambda x. (\text{send } x \ (\text{recv } a))$ $: ![\text{int}]^{n+1} \rightarrow^{n, n+1} \text{unit}$

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

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

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



$a : p[t]^n \vdash (\lambda x. x) a$



Typing applications

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

$\vdash (\lambda x. x) 3$ ☺

$a : p[t]^n \vdash (\lambda x. x) a$ ☺

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

Typing applications

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

$\vdash (\lambda x. x) 3$ ☺

$a : p[t]^n \vdash (\lambda x. x) a$ ☺

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

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

Typing applications

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

$\vdash (\lambda x. x) 3$ 😊

$a : p[t]^n \vdash (\lambda x. x) a$ 😊

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

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

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

Typing forks

$$\text{fork} : \forall i. \forall j. (\text{unit} \rightarrow^{i,j} \text{unit}) \rightarrow \text{unit}$$

- effect masking [Amtoft, Nielson, Nielson, 99]

Example: parallel Fibonacci

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

Properties of well-typed programs

Theorem

- ① *typing is **preserved** by reductions*
- ② *computations are **confluent***
- ③ *well-typed, **closed** programs are **deadlock free***
- ④ *well-typed, **convergent** programs typed with **discrete** priorities eventually **use** all of their channels*

`send a (rec x x)`

*(some sensible programs require **dense** priorities)*

Polymorphic effects and recursion

```
let rec fibo n c =  
  if n ≤ 1 then n  
  else let (a , b ) = (open(), open()) in  
    fork λ_.(fibo (n - 1) a );  
    fork λ_.(fibo (n - 2) b );  
    send c (recv a  + recv b  )
```

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

Polymorphic effects and recursion

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

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

Polymorphic effects and recursion

```
let rec fibo n ci =  
  if n ≤ 1 then n  
  else let (ai-2, bi-1) = (open(), 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]

The priority of type variables

$$\lambda x. \lambda y. (x, y) : \quad \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta$$

The priority of type variables

$$\lambda x. \lambda y. (x, y) : \forall i. \forall j. \forall \alpha^i. \forall \beta^j. \alpha \rightarrow \beta \rightarrow^{i, \perp} \alpha \times \beta$$

Constraints on priority variables

Parallel `sends`

```
 $\lambda x. \lambda y. \text{fork } \lambda_. (\text{send } x \ 1); \text{ fork } \lambda_. (\text{send } y \ 2)$ 
```

```
 $\forall i. \forall j. ![\text{int}]^i \rightarrow ![\text{int}]^j \rightarrow^i \text{unit}$ 
```

Constraints on priority variables

Parallel sends

$\lambda x. \lambda y. \text{fork } \lambda_. (\text{send } x \ 1); \text{ fork } \lambda_. (\text{send } y \ 2)$

$\forall i. \forall j. ![\text{int}]^i \rightarrow ![\text{int}]^j \rightarrow^i \text{unit}$

Sequential sends

$\lambda x. \lambda y. \text{send } x \ 1; \text{ send } y \ 2$

$\forall i. \forall j. (i < j) \Rightarrow ![\text{int}]^i \rightarrow ![\text{int}]^j \rightarrow^i \text{unit}$

Recursive types

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

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

type of x

type of y

$\forall \alpha . \quad \alpha \text{ In} \rightarrow \alpha \text{ Out} \rightarrow \text{unit}$

Recursive types

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

```
type α Ini = ?[α × α Ini ]i  
type α Outj = ![α × α Inj ]j
```

$\forall i. \forall j. \forall \alpha . \quad \alpha \text{ In}^i \rightarrow \alpha \text{ Out}^j \rightarrow^j \text{unit}$

Recursive types

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

```
type α Ini = ?[α × α Ini ]i
```

```
type α Outj = ![α × α Inj ]j
```

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

Recursive types

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

```
type  $\alpha \text{ In}^i = ?[\alpha \times \alpha \text{ In}^{i+1}]^i$  non-regular type  
type  $\alpha \text{ Out}^j = ![\alpha \times \alpha \text{ In}^j]^j$ 
```

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

Recursive types

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

```
type  $\alpha \text{ In}^i = ?[\alpha \times \alpha \text{ In}^{i+1}]^i$   
type  $\alpha \text{ Out}^j = ![\alpha \times \alpha \text{ In}^{j+1}]^j$  non-regular type
```

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

Recursive types

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

```
type  $\alpha \text{ In}^i = ?[\alpha \times \alpha \text{ In}^{i+1}]^i$   
type  $\alpha \text{ Out}^j = ![\alpha \times \alpha \text{ In}^{j+1}]^j$ 
```

unlimited messages only!

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

Recursive types

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

```
type  $\alpha \text{ In}^i = ?[\alpha \times \alpha \text{ In}^{i+1}]^i$   
type  $\alpha \text{ Out}^j = ![\alpha \times \alpha \text{ In}^{j+1}]^j$ 
```

tail applications only!

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

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
```

Concluding remarks

Question

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

Answer

Simple mechanism, but full integration requires advanced features

- priority polymorphism
- priority constraints
- polymorphic recursion (doable)
- higher-rank polymorphism (see [Reppy, 99])
- non-regular types (regular representation possible)

Variations and ongoing work

Lazy evaluation

- Monadic I/O
 - Luca Novara

Type reconstruction for the linear π -calculus

- Preliminary results
(monomorphic types, polymorphism on priorities)
 - Tzu-Chun Chen
 - Andrea Tosatto