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]

$$\{\ \texttt{send}\ a\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Ingredients

- call-by-value λ-calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

A deadlock in CML [Reppy, 1999]

$$\{\ \underbrace{\texttt{send}}_{\texttt{![int]} \to \texttt{int} \to \texttt{unit}}\ \overbrace{a}^{\texttt{![int]}}\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Ingredients

- call-by-value λ-calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

A deadlock in CML [Reppy, 1999]

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Ingredients

- call-by-value λ-calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

A deadlock in CML [Reppy, 1999]

$$\{\ \underbrace{\texttt{send}}_{\texttt{int} \to \texttt{unit}}\ a\ (\overbrace{\texttt{recv}}^{?[\texttt{int}] \to \texttt{int}}\ \underbrace{b}_{?[\texttt{int}]})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Ingredients

- call-by-value λ-calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

A deadlock in CML [Reppy, 1999]

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\overbrace{\texttt{recv}\ b}^{\texttt{int}})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Ingredients

- call-by-value λ-calculus
- `open`, `send`, `recv`, `fork`
- **linear** channels

Outline

Outline

Channel levels

$$\{\ \texttt{send}\ a^n\ (\texttt{recv}\ b^m)\ \} \mid \{\ \texttt{send}\ b^m\ (\texttt{recv}\ a^n)\ \}$$

Channel levels **in types**

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

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

Channel levels **in types**

$$\{\ \underbrace{\texttt{send}}_{\texttt{!}[\texttt{int}]^n \to \texttt{int} \to \texttt{unit}}\ \overbrace{a}^{\texttt{!}[\texttt{int}]^n}\ (\texttt{recv}\ b)\ \}|\{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

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

Channel levels **in types**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

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

Channel levels **in types**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\overbrace{\texttt{recv}}^{?[\texttt{int}]^m \to \texttt{int}}\ \underbrace{b}_{?[\texttt{int}]^m})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

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

Channel levels **in types**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\overbrace{\texttt{recv}\ b}^{\texttt{int}})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

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

Channel levels **in types**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to \texttt{unit}}\ (\overbrace{\texttt{recv}\ b}^{\texttt{int}})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

- 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**

$$\{\ \underbrace{\texttt{send}}_{\texttt{![int]}^n \to \texttt{int} \to^n \texttt{unit}\ \&\ \bot}\ \overbrace{a}^{\texttt{![int]}^n\ \&\ \bot}\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Channel levels **in types and effects**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \rightarrow^{n} \texttt{unit}\ \&\ \bot}\ (\texttt{recv}\ b)\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Channel levels **in types and effects**

$$\{\ \mathtt{send}\ a\ (\overbrace{\mathtt{recv}}^{?[\mathtt{int}]^m \to^m \mathtt{int}\ \&\ \bot}\ \underbrace{b}_{?[\mathtt{int}]^m\ \&\ \bot})\ \} \mid \{\ \mathtt{send}\ b\ (\mathtt{recv}\ a)\ \}$$

Channel levels **in types and effects**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int} \to^{n} \texttt{unit}\ \&\ \bot}\ (\overbrace{\texttt{recv}\ b}^{\texttt{int}\ \&\ m})\ \} \mid \{\ \texttt{send}\ b\ (\texttt{recv}\ a)\ \}$$

Channel levels **in types and effects**

$$\{\ \underbrace{\texttt{send}\ a}_{\texttt{int}\ \to^{n}\ \texttt{unit}\ \&\ \bot}\ (\overbrace{\texttt{recv}\ b}^{\texttt{int}\ \&\ m})\ \} \mid \{\ \underbrace{\texttt{send}\ b}_{\texttt{int}\ \to^{m}\ \texttt{unit}}\ (\overbrace{\texttt{recv}\ a}^{\texttt{int}\ \&\ n})\ \}$$

More on arrow types

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

Which type for f?

$f : \mathtt{int} \to^{m} \mathtt{unit}$

$f : \mathtt{int} \to^{n} \mathtt{unit}$

More on arrow types

$$<$$

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

Which type for f?

$$f : \mathtt{int} \to^m \mathtt{unit} \qquad\qquad f : \mathtt{int} \to^n \mathtt{unit}$$

$$\{\ (\underbrace{f\ 3}_{\mathtt{int}\ \&\ m});\ \overbrace{\mathtt{recv}\ b}^{\mathtt{int}\ \&\ n}\ \} \mid \{\ \mathtt{recv}\ a\ \}$$

More on arrow types

$<$

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

Which type for f?

$$f : \texttt{int} \to^m \texttt{unit}$$

$$\{\ \underbrace{(f\ 3)}_{\texttt{int}\ \&\ m};\ \overbrace{\texttt{recv}\ b}^{\texttt{int}\ \&\ n}\ \} | \{\ \texttt{recv}\ a\ \}$$

$$f : \texttt{int} \to^n \texttt{unit}$$

$$\{\ \underbrace{f}_{\texttt{int} \to^n \texttt{unit}\ \&\ \bot}\ (\overbrace{\texttt{recv}\ a}^{\texttt{int}\ \&\ m})\ \} | \{\ \texttt{recv}\ b\ \}$$

Typing abstractions

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

$$\vdash \lambda x\,.\,x \qquad\qquad : \mathtt{int} \rightarrow^{\top,\bot} \mathtt{int}$$

Typing abstractions

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

$$\begin{aligned} &\vdash \lambda x\,.\,x && : \texttt{int} \to^{\top,\bot} \texttt{int} \\ a : \texttt{!}\,\texttt{[int]}^{n} &\vdash \lambda x\,.\,(x\,,\ a) && : \texttt{int} \to^{n,\bot} \texttt{int} \times \texttt{!}\,\texttt{[int]}^{n} \end{aligned}$$

Typing abstractions

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

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

Typing abstractions

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

$$\begin{array}{rll}
& \vdash \lambda x\,.\,x & : \texttt{int} \rightarrow^{\top,\bot} \texttt{int} \\
a : \,!\,[\texttt{int}]^{n} & \vdash \lambda x\,.\,(x,\ a) & : \texttt{int} \rightarrow^{n,\bot} \texttt{int} \times\, !\,[\texttt{int}]^{n} \\
& \vdash \lambda x\,.\,(\texttt{send}\ x\ 3) & : \,!\,[\texttt{int}]^{n} \rightarrow^{\top,n} \texttt{unit} \\
a : \,?\,[\texttt{int}]^{n} & \vdash \lambda x\,.\,(\texttt{recv}\ a{+}x) & : \texttt{int} \rightarrow^{n,n} \texttt{int}
\end{array}$$

Typing applications

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

Typing applications

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

$\vdash (\lambda x.x)\ 3$ ☺

Typing applications

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

$$\vdash (\lambda x.x)\ 3 \qquad ☺$$

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

Typing applications

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

$$\vdash (\lambda x.x)\ 3 \qquad ☺$$

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

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

Typing applications

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

$\vdash (\lambda x.x)\ 3$ ☺

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

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

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

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*

$$\texttt{send}\ a\ (\texttt{rec}\ x\ x)$$

(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

Example: parallel Fibonacci

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

$$\texttt{fibo} : \quad \texttt{int} \rightarrow \texttt{!}[\texttt{int}] \rightarrow \quad \texttt{unit}$$

Example: parallel Fibonacci

$$
\begin{array}{l}
\texttt{let rec fibo } n\ c^{i}\ \texttt{=} \\
\quad \texttt{if } n \leq 1 \texttt{ then send } c^{i}\ \texttt{1} \\
\quad \texttt{else let } a^{i-2}\ \texttt{= open() in} \\
\qquad\quad \texttt{let } b^{i-1}\ \texttt{= open() in} \\
\qquad\quad \texttt{fork } \lambda\texttt{().(fibo (}n\texttt{ - 1) } a^{i-2}\texttt{);} \\
\qquad\quad \texttt{fork } \lambda\texttt{().(fibo (}n\texttt{ - 2) } b^{i-1}\texttt{);} \\
\qquad\quad \texttt{send } c^{i}\ \texttt{(recv } a^{i-2}\texttt{ + recv } b^{i-1}\texttt{)}
\end{array}
$$

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

Example: parallel Fibonacci

```
let rec fibo n c^i =
  if n ≤ 1 then send c^i 1
  else let a^{i−2} = open() in
       let b^{i−1} = open() in
       fork λ().(fibo (n - 1) a^{i−2});
       fork λ().(fibo (n - 2) b^{i−1});
       send c^i (recv a^{i−2} + recv b^{i−1})
```

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

- 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)
```

$$\texttt{forward} : \quad \forall \alpha. \quad ?[\alpha] \to ![\alpha] \to \texttt{unit}$$

Example: linear forwarder

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

$$\texttt{forward} : \forall i, j. \forall \alpha. \qquad \texttt{?[}\alpha\texttt{]}^i \rightarrow \texttt{![}\alpha\texttt{]}^j \rightarrow^{i,j} \texttt{unit}$$

Example: linear forwarder

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

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

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
```

$\texttt{type In}\ \ \alpha = \texttt{?}[\alpha \times \texttt{In}\ \ \alpha]$ type of x

$\texttt{type Out}\ \alpha = \texttt{!}[\alpha \times \texttt{In}\ \ \alpha]$ type of y

$$\texttt{copy} : \forall\alpha.\ \texttt{In}\ \alpha \to \texttt{Out}\ \alpha \to \texttt{unit}$$

Example: persistent forwarder

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

$$\texttt{type}\ \ \texttt{In}\ i\ \alpha\ \texttt{=}\ \texttt{?[}\alpha \times \texttt{In}\ \ i \qquad \alpha\texttt{]}^{i}$$

$$\texttt{type}\ \texttt{Out}\ j\ \alpha\ \texttt{=}\ \texttt{![}\alpha \times \texttt{In}\ \ j \qquad \alpha\texttt{]}^{j}$$

$$\texttt{copy} : \forall i, j. \forall \alpha. \qquad \texttt{In}\ i\ \alpha \to \texttt{Out}\ j\ \alpha \to^{i,} \quad \texttt{unit}$$

Example: persistent forwarder

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

receive from x...

...then send on y

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

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

Example: persistent forwarder

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

c received from x

```
type  In i α = ?[α × In (i + 1) α]^i
type Out j α = ![α × In  j      α]^j
```

non-regular type

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

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}
```

d sent on *y*

$$\texttt{type}\ \ \texttt{In}\ i\ \alpha = \texttt{?}[\alpha \times \texttt{In}\ (i+1)\ \alpha]^{i}$$
$$\texttt{type}\ \texttt{Out}\ j\ \alpha = \texttt{!}[\alpha \times \texttt{In}\ (j+1)\ \alpha]^{j}$$

non-regular type

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

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 α = ?[α × In (i + 1) α]^i
type Out j α = ![α × In (j + 1) α]^j
```

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

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

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