Effpi

concurrent programming with dependent behavioural types

Alceste Scalas
with Elias Benussi & Nobuko Yoshida

Imperial College London

VeTSS PhD school / FMATS workshop
Microsoft Research Cambridge, 25 September 2018

The problem

Languages and toolkits for **message-passing concurrent programming** provide **intuitive high-level abstractions**

- e.g., actors, channels, processes (Akka, Erlang, Go, . . .)

. . . but **do not allow** to **verify code against behavioural specs**

- risks: **protocol violations**, **deadlocks**, **starvation**, . . .
- issues found **at run-time**, hence **expensive to fix**
- can vehicle **attacks**: e.g., data breaches, DoS

The problem and our solution

Languages and toolkits for **message-passing concurrent programming** provide **intuitive high-level abstractions**

- e.g., actors, channels, processes (Akka, Erlang, Go, . . .)

. . . but **do not allow** to **verify code against behavioural specs**

- risks: **protocol violations**, **deadlocks**, **starvation**, . . .
- issues found **at run-time**, hence **expensive to fix**
- can vehicle **attacks**: e.g., data breaches, DoS

Our solution: Effpi, a toolkit for **strongly-typed concurrent programming** in **Dotty (a.k.a. Scala 3)**

- using **types as behavioural specifications**
- and **type-level model checking** to **verify code properties**

Example: payment service with auditing

A **payment service** should implement the following **specification**:

1. wait to receive a **payment request**
2. then, **either**:
   2.1 **reject** the payment, or
   2.2 report the payment to an **audit** service, and then **accept** it
3. continue from point 1

Example: payment service with auditing

Demo!

What is the Dotty / Scala 3 compiler saying?

```
found:      Out[ActorRef[Result], Accepted]

required:   Out[ActorRef[Result](pay.replyTo), Rejected]
            |
            Out[ActorRef[Audit[_]](aud), Audit[Pay(pay)]] >>:
               Out[ActorRef[Result](pay.replyTo), Accepted]
```

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$\textbf{let}\ pinger\ =\ \lambda self.\lambda pongc.($

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$\textbf{let}\ \mathit{pinger}\ =\ \lambda \mathit{self}.\lambda \mathit{pongc}.($
$\quad \textbf{send}(\mathit{pongc},\ \mathit{self},\ \lambda\_.($

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$\textbf{let}\ \mathit{pinger}\ = \lambda \mathit{self}.\lambda \mathit{pongc}.($
$\quad \textbf{send}(\mathit{pongc}, \mathit{self}, \lambda\_.($
$\quad\quad \textbf{recv}(\mathit{self}, \lambda \mathit{reply}.($

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$$
\begin{array}{l}
\textbf{let}\ \mathit{pinger} = \lambda \mathit{self}.\lambda \mathit{pongc}.(\\
\quad \textbf{send}(\mathit{pongc}, \mathit{self}, \lambda\_.(\\
\quad\quad \textbf{recv}(\mathit{self}, \lambda \mathit{reply}.(\\
\quad\quad\quad \textbf{end}\ ))))))
\end{array}
$$

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$$\begin{array}{l}\textbf{let}\ \mathit{pinger}\ =\ \lambda \mathit{self}\,.\lambda \mathit{pongc}.(\\ \quad \textbf{send}(\mathit{pongc},\ \mathit{self},\ \lambda\_.(\\ \quad\quad \textbf{recv}(\mathit{self},\ \lambda \mathit{reply}.(\\ \quad\quad\quad \textbf{end}\ )))))\end{array}$$

$$\begin{array}{l}\textbf{let}\ \mathit{ponger}\ =\ \lambda \mathit{self}\,.(\\ \quad \textbf{recv}(\mathit{self},\ \lambda \mathit{reqc}.(\\ \quad\quad \textbf{send}(\mathit{reqc},\ \texttt{"Hello!"},\ \lambda\_.(\\ \quad\quad\quad \textbf{end}\ )))))\end{array}$$

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$$\textbf{let}\ \mathit{pinger} = \lambda \mathit{self}.\lambda \mathit{pongc}.(\ \textbf{send}(\mathit{pongc}, \mathit{self}, \lambda\_.(\ \textbf{recv}(\mathit{self}, \lambda \mathit{reply}.(\ \textbf{end}\ )))))$$

$$\textbf{let}\ \mathit{ponger} = \lambda \mathit{self}.(\ \textbf{recv}(\mathit{self}, \lambda \mathit{reqc}.(\ \textbf{send}(\mathit{reqc}, \texttt{"Hello!"}, \lambda\_.(\ \textbf{end}\ )))))$$

$$\textbf{let}\ \mathit{pingpong} = \lambda c1.\lambda c2.\big(\ \mathit{pinger}\ c1\ c2 \parallel \mathit{ponger}\ c2\ \big)$$

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$\textbf{let}\ \mathit{pinger} = \lambda \mathit{self}.\lambda \mathit{pongc}.($
$\quad \textbf{send}(\mathit{pongc}, \mathit{self}, \lambda\_.($
$\quad\quad \textbf{recv}(\mathit{self}, \lambda \mathit{reply}.($
$\quad\quad\quad \textbf{end}\ )))))$

$\textbf{let}\ \mathit{ponger} = \lambda \mathit{self}.($
$\quad \textbf{recv}(\mathit{self}, \lambda \mathit{reqc}.($
$\quad\quad \textbf{send}(\mathit{reqc}, \texttt{"Hello!"}, \lambda\_.($
$\quad\quad\quad \textbf{end}\ )))))$

$$\textbf{let}\ \mathit{pingpong} = \lambda c1.\lambda c2.\big(\mathit{pinger}\ c1\ c2 \parallel \mathit{ponger}\ c2\big)$$

$$\textbf{let}\ \mathit{main} = \textbf{let}\ c1 = \textbf{chan}();\ \textbf{let}\ c2 = \textbf{chan}();\ \mathit{pingpong}\ c1\ c2$$

A λ-calculus with communication & concurrency

Example: a *pinger* **process** sends a **communication channel** to a *ponger* process, who uses the channel to reply `"Hello!"`

$$\begin{array}{l} \textbf{let}\ \mathit{pinger}\ =\ \lambda \mathit{self}.\lambda \mathit{pongc}.(\\ \quad \textbf{send}(\mathit{pongc},\ \mathit{self},\ \lambda\_.(\\ \quad\quad \textbf{recv}(\mathit{self},\ \lambda \mathit{reply}.(\\ \quad\quad\quad \textbf{end}\ ))))) \end{array} \qquad \begin{array}{l} \textbf{let}\ \mathit{ponger}\ =\ \lambda \mathit{self}.(\\ \quad \textbf{recv}(\mathit{self},\ \lambda \mathit{reqc}.(\\ \quad\quad \textbf{send}(\mathit{reqc},\ \texttt{"Hello!"},\ \lambda\_.(\\ \quad\quad\quad \textbf{end}\ ))))) \end{array}$$

$$\textbf{let}\ \mathit{pingpong}\ =\ \lambda \mathit{c1}.\lambda \mathit{c2}.\big(\ \mathit{pinger}\ \mathit{c1}\ \mathit{c2}\ \parallel\ \mathit{ponger}\ \mathit{c2}\ \big)$$

$$\textbf{let}\ \mathit{main}\ =\ \textbf{let}\ \mathit{c1} = \textbf{chan}();\ \textbf{let}\ \mathit{c2} = \textbf{chan}();\ \mathit{pingpong}\ \mathit{c1}\ \mathit{c2}$$

Monadic encoding of the **higher-order π-calculus**

- λ-terms model **abstract processes**
- **Continuations** are expressed as λ-terms

How to type a process calculus

For typing, we use a context Γ with **channel types**. E.g.:

$$\Gamma = x:\mathsf{str}, y:\mathsf{c}^{\mathsf{o}}[\mathsf{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash \texttt{"Hello"} +\!\!+ x : \mathsf{str}$$

How to type a process calculus

For typing, we use a context Γ with **channel types**. E.g.:

$$\Gamma = x : \mathsf{str},\ y : \mathsf{c}^{\mathsf{o}}[\mathsf{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash \texttt{"Hello"} +\!\!+\, x : \mathsf{str}$$

How do we **type communication?** E.g., if $\mathrm{t} = \mathbf{send}(y, x, \lambda\_.\mathbf{end})$

Classic approach: $\Gamma \vdash \mathrm{t} : \mathbf{proc}$ ("t is a well-typed process in Γ")

How to type a process calculus

For typing, we use a context Γ with **channel types**. E.g.:

$$\Gamma = x : \mathsf{str}, \; y : \mathsf{c}^{\mathsf{o}}[\mathsf{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash \texttt{"Hello"} +\!\!+ \, x : \mathsf{str}$$

How do we **type communication?** E.g., if $t = \mathbf{send}(y, x, \lambda\_.\mathbf{end})$

Classic approach: $\Gamma \vdash t : \mathbf{proc}$ ("t is a well-typed process in Γ")

Our approach: $\Gamma \vdash t : T$ (**"t behaves as T in Γ"**)

How to type a process calculus

For typing, we use a context Γ with **channel types**. E.g.:

$$\Gamma = x : \mathsf{str}\,,\ y : \mathsf{c}^{\mathsf{o}}[\mathsf{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash \texttt{"Hello"} +\!\!+\, x \;:\; \mathsf{str}$$

How do we **type communication?** E.g., if $t = \mathbf{send}(y, x, \lambda\_.\mathbf{end})$

Classic approach: $\Gamma \vdash t : \mathbf{proc}$ (“t is a well-typed process in Γ”)

Our approach: $\Gamma \vdash t : T$ (**“**t **behaves as** T **in** Γ**”**)

$\Gamma \vdash T \leqslant \mathbf{proc}$ (**“**T **is a refined process type”**)

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str}\,,\; y : \mathsf{c^o[str]} \;\vdash\; \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \qquad : \; \mathsf{T}$$

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str}\,,\; y : \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \;\vdash\; \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \qquad : \; \mathsf{T} \; = \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}],\, \mathsf{str},\, \mathbf{nil}]$$

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str}\,,\; y : \mathsf{c^o[str]} \;\vdash\; \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \; \mathsf{T} \;=\; \mathbf{o}[\mathsf{c^o[str]}, \mathsf{str}, \mathbf{nil}]$$

$$\varnothing \;\vdash\; \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) \; : \; \mathsf{T}'$$

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str}\,,\ y : \mathsf{c^o}[\mathsf{str}] \ \vdash\ \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \ \mathsf{T} \ = \mathbf{o}[\mathsf{c^o}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

$$\varnothing \ \vdash\ \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) \ : \ \mathsf{T}' = \mathsf{str} \to \mathsf{c^o}[\mathsf{str}] \to \mathsf{T}$$

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str},\ y : \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \vdash \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \mathsf{T} = \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

$$\emptyset \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : \mathsf{T}' = \mathsf{str} \to \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \to \mathsf{T}$$

Can we **use types** to **specify** and **verify process behaviours**?

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str},\ y : \mathsf{c^{o}[str]} \vdash \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \mathsf{T} = \mathbf{o}[\mathsf{c^{o}[str]}, \mathsf{str}, \mathbf{nil}]$$

$$\varnothing \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : \mathsf{T}' = \mathsf{str} \to \mathsf{c^{o}[str]} \to \mathsf{T}$$

Can we **use types** to **specify** and **verify process behaviours**?
Yes — almost!

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x:\mathsf{str},\ y:\mathsf{c}^{\mathsf{o}}[\mathsf{str}] \vdash \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \mathsf{T} = \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

$$\varnothing \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : \mathsf{T}' = \mathsf{str} \to \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \to \mathsf{T}$$

Can we **use types** to **specify** and **verify process behaviours**?
Yes — almost!

If a term t has type T' above, we know that:

1. t is an **abstract process**. . .
2. that takes a string and a channel. . .
3. sends **some** string on **some** channel, then terminates

Behavioural types (inspired by π-calculus theory)

Some examples:

$$x : \mathsf{str}\,,\; y : \mathsf{c^o}[\mathsf{str}] \;\vdash\; \mathbf{send}(y, x, \lambda\_.\mathbf{end}) \quad : \mathsf{T} = \mathbf{o}[\mathsf{c^o}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

$$\varnothing \;\vdash\; \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) \;:\; \mathsf{T}' = \mathsf{str} \to \mathsf{c^o}[\mathsf{str}] \to \mathsf{T}$$

Can we **use types** to **specify** and **verify process behaviours**?
Yes — almost!

If a term t has type T' above, we know that:

1. t is an **abstract process**. . .
2. that takes a string and a channel. . .
3. sends **some** string on **some** channel, then terminates

Here's a term **with the same type** T', but **different behaviour**:

$$\lambda x.\lambda y.\big(\mathbf{let}\; z = \mathbf{chan}();\; \mathbf{send}(z, \texttt{"Hello!"}, \lambda\_.\mathbf{end})\big)$$

Behavioural types

This type is not very precise: e.g., it **does not track channel use**

$$T' = \mathsf{str} \to \mathsf{c}^{\circ}[\mathsf{str}] \to \mathbf{o}[\mathsf{c}^{\circ}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

Behavioural types and dependent function types

This type is not very precise: e.g., it **does not track channel use**

$$\mathsf{T}' = \mathsf{str} \to \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \to \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

Introduce **dependent function types** (adapted from Dotty / Scala 3):

$\Pi(x{:}\mathsf{T}_1)\mathsf{T}_2$ where the return type T_2 can refer to x

Behavioural types and dependent function types

This type is not very precise: e.g., it **does not track channel use**

$$\mathsf{T}' = \mathsf{str} \to \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \to \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

Introduce **dependent function types** (adapted from Dotty / Scala 3):

$\Pi(x{:}\mathsf{T}_1)\mathsf{T}_2$ where the return type T_2 can refer to x

E.g., if term t has type $\mathsf{T}'' = \Pi(x{:}\mathsf{str})\,\Pi(y{:}\mathsf{c}^{\mathsf{o}}[\mathsf{str}])\,\mathbf{o}[y, x, \mathbf{nil}]$

1. t is an **abstract process**. . .
2. that takes a string x and a channel y. . .
3. sends x on channel y, then terminates

Behavioural types and dependent function types

This type is not very precise: e.g., it **does not track channel use**

$$T' = \mathsf{str} \rightarrow \mathsf{c^o}[\mathsf{str}] \rightarrow \mathbf{o}[\mathsf{c^o}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

Introduce **dependent function types** (adapted from Dotty / Scala 3):

$\Pi(x{:}T_1)T_2$ where the return type T_2 can refer to x

E.g., if term t has type $T'' = \Pi(x{:}\mathsf{str})\,\Pi(y{:}\mathsf{c^o}[\mathsf{str}])\;\mathbf{o}[y, x, \mathbf{nil}]$

1. t is an **abstract process**. . .
2. that takes a string x and a channel y. . .
3. sends x on channel y, then terminates

We can have multiple **levels of refinement**:
$\varnothing \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : T''$

Behavioural types and dependent function types

This type is not very precise: e.g., it **does not track channel use**

$$T' = \text{str} \to c^{o}[\text{str}] \to \mathbf{o}[c^{o}[\text{str}], \text{str}, \mathbf{nil}]$$

Introduce **dependent function types** (adapted from Dotty / Scala 3):

$\Pi(x{:}T_1)T_2$ where the return type T_2 can refer to x

E.g., if term t has type $T'' = \Pi(x{:}\text{str})\,\Pi(y{:}c^{o}[\text{str}])\,\mathbf{o}[y, x, \mathbf{nil}]$

1. t is an **abstract process**...
2. that takes a string x and a channel y...
3. sends x on channel y, then terminates

We can have multiple **levels of refinement**:
$\varnothing \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : T'' \leqslant T'$

Behavioural types and dependent function types

This type is not very precise: e.g., it **does not track channel use**

$$\mathsf{T}' = \mathsf{str} \to \mathsf{c}^{\mathsf{o}}[\mathsf{str}] \to \mathbf{o}[\mathsf{c}^{\mathsf{o}}[\mathsf{str}], \mathsf{str}, \mathbf{nil}]$$

Introduce **dependent function types** (adapted from Dotty / Scala 3):

$\Pi(x{:}\mathsf{T}_1)\mathsf{T}_2$ where the return type T_2 can refer to x

E.g., if term t has type $\mathsf{T}'' = \Pi(x{:}\mathsf{str})\, \Pi(y{:}\mathsf{c}^{\mathsf{o}}[\mathsf{str}])\, \mathbf{o}[y, x, \mathbf{nil}]$

1. t is an **abstract process**. . .
2. that takes a string x and a channel y. . .
3. sends x on channel y, then terminates

We can have multiple **levels of refinement**:

$$\varnothing \vdash \lambda x.\lambda y.\mathbf{send}(y, x, \lambda\_.\mathbf{end}) : \mathsf{T}'' \leqslant \mathsf{T}' \leqslant \mathsf{c}^{\mathsf{o}}[\mathsf{none}] \to \mathsf{str} \to \mathbf{proc}$$

Types as behavioural specifications: examples

Types can provide **accurate behavioural specifications**. E.g.:

$$\mathsf{T}_1 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{o}[\,y\,,\,x,\,\mathbf{i}[\,x,\,\Pi(z{:}\ldots)\,\mathbf{nil}\,]\,]$$

"Take x and y; use y send x; use x to receive some z; and terminate"

Types as behavioural specifications: examples

Types can provide **accurate behavioural specifications**. E.g.:

$$\mathsf{T}_1 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{o}[\,y\,,\,x,\,\mathbf{i}[\,x,\,\Pi(z{:}\ldots)\,\mathbf{nil}\,]\,]$$

"Take x and y; use y send x; use x to receive some z; and terminate"

$$\mathsf{T}_2 \;=\; \Pi(x{:}\ldots)\,\mathbf{i}[\,x\,,\,\Pi(y{:}\ldots)\,\mathbf{o}[y,\mathsf{str},\mathbf{nil}]\,]$$

"Take x; use x to input some y; use y to send a string; and terminate"

Types as behavioural specifications: examples

Types can provide **accurate behavioural specifications**. E.g.:

$$\mathsf{T}_1 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{o}[\,y\,,\,x,\,\mathbf{i}[\,x,\,\Pi(z{:}\ldots)\,\mathbf{nil}\,]\,]$$

"Take x and y; use y send x; use x to receive some z; and terminate"

$$\mathsf{T}_2 \;=\; \Pi(x{:}\ldots)\,\mathbf{i}[\,x\,,\,\Pi(y{:}\ldots)\,\mathbf{o}[y,\mathsf{str},\mathbf{nil}]\,]$$

"Take x; use x to input some y; use y to send a string; and terminate"

- T_1 and T_2 are respectively the types of the *pinger* and *ponger* processes

Types as behavioural specifications: examples

Types can provide **accurate behavioural specifications**. E.g.:

$$\mathsf{T}_1 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{o}[\,y\,,\,x,\,\mathbf{i}[\,x,\,\Pi(z{:}\ldots)\,\mathbf{nil}\,]\,]$$

"Take x and y; use y send x; use x to receive some z; and terminate"

$$\mathsf{T}_2 \;=\; \Pi(x{:}\ldots)\,\mathbf{i}[\,x\,,\,\Pi(y{:}\ldots)\,\mathbf{o}[y,\mathsf{str},\mathbf{nil}]\,]$$

"Take x; use x to input some y; use y to send a string; and terminate"

- T_1 and T_2 are respectively the types of the *pinger* and *ponger* processes

$$\mathsf{T}_3 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{p}[\,\mathsf{T}_1\,x\,y\;,\;\mathsf{T}_2\,y\,]$$

"Take x and y; use them to apply T_1 and T_2; run such behaviours in parallel"

Types as behavioural specifications: examples

Types can provide **accurate behavioural specifications**. E.g.:

$$\mathsf{T}_1 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{o}[\,y\,,\,x,\,\mathbf{i}[\,x,\,\Pi(z{:}\ldots)\,\mathbf{nil}\,]\,]$$

"Take x and y; use y send x; use x to receive some z; and terminate"

$$\mathsf{T}_2 \;=\; \Pi(x{:}\ldots)\,\mathbf{i}[\,x\,,\,\Pi(y{:}\ldots)\,\mathbf{o}[y,\mathsf{str},\mathbf{nil}]\,]$$

"Take x; use x to input some y; use y to send a string; and terminate"

- T_1 and T_2 are respectively the types of the *pinger* and *ponger* processes

$$\mathsf{T}_3 \;=\; \Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{p}[\,\mathsf{T}_1\,x\,y\;,\;\mathsf{T}_2\,y\,]$$

"Take x and y; use them to apply T_1 and T_2; run such behaviours in parallel"

- T_3 is the type of the *pingpong* process

Types as behavioural specifications (cont'd)

Type checking guarantees **type safety**. . .

- E.g.: no strings can be sent on channels carrying integers

Types as behavioural specifications (cont'd)

Type checking guarantees **type safety**. . .

- E.g.: no strings can be sent on channels carrying integers

. . . and conformance with **rich behavioural specifications** — that can be **complicated**, especially when **composed**

- E.g., the *pingpong* type: $\Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{p}[\,\mathsf{T}_1\,x\,y\,,\,\mathsf{T}_2\,y\,]$

Types can model **races** on shared channels, and **deadlocks**!

Types as behavioural specifications (cont'd)

Type checking guarantees **type safety**. . .

- E.g.: no strings can be sent on channels carrying integers

. . . and conformance with **rich behavioural specifications** — that can be **complicated**, especially when **composed**

- E.g., the *pingpong* type: $\Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{p}[\,\mathsf{T}_1\,x\,y\;,\;\mathsf{T}_2\,y\,]$

Types can model **races** on shared channels, and **deadlocks**!

Verification via **"type-level symbolic execution"**

- Give a **labelled semantics** to a type T
- **Model check** the **safety/liveness** properties of T
- Show how, if $\vdash \mathsf{t} : \mathsf{T}$ holds, then t "inherits" T's properties

Types as behavioural specifications (cont'd)

Type checking guarantees **type safety**. . .

- E.g.: no strings can be sent on channels carrying integers

. . . and conformance with **rich behavioural specifications** — that can be **complicated**, especially when **composed**

- E.g., the *pingpong* type: $\Pi(x{:}\ldots)\,\Pi(y{:}\ldots)\,\mathbf{p}[\,\mathsf{T}_1\,x\,y\,,\,\mathsf{T}_2\,y\,]$

Types can model **races** on shared channels, and **deadlocks**!

Verification via **"type-level symbolic execution"**

- Give a **labelled semantics** to a type T
- **Model check** the **safety/liveness** properties of T
- Show how, if $\vdash \mathrm{t} : \mathsf{T}$ holds, then t "inherits" T's properties

Model checking is **decidable** for T, but **not** for t (Goltz'90; Esparza'97)

From theory to Dotty / Scala3

We **directly translate our types in Dotty / Scala 3**:

$$\Pi(x{:}\mathsf{str})\ \Pi(y{:}\mathsf{c}^{\mathsf{o}}[\mathsf{str}])\ \mathbf{o}[y,\ x,\ \mathbf{nil}]$$

$$\Downarrow$$

```
(x: String, y: OChan[String]) => Out[y.type, x.type, Nil]
```

From theory to Dotty / Scala3

We **directly translate our types in Dotty / Scala 3**:

$$\Pi(x{:}\mathsf{str})\ \Pi(y{:}\mathsf{c}^{\circ}[\mathsf{str}])\ \mathbf{o}[y,\ x,\ \mathbf{nil}]$$

$$\Downarrow$$

```
(x: String, y: OChan[String]) => Out[y.type, x.type, Nil]
```

We implement our calculus as a **deeply-embedded DSL**. E.g.:

- calling `send(...)` yields an **object of type** `Out[...]`
- the object **describes** *(does not perform!)* **the desired output**
- the object is **interpreted** by a **runtime system**. . .
- . . . that performs the actual output

From theory to Dotty / Scala3

Demo!

A simplified actor-based DSL

We have discussed a **process-based calculus and DSL**. . .
. . . but the opening example was **actor-based!**

A simplified actor-based DSL

We have discussed a **process-based calculus and DSL**. . .
. . . but the opening example was **actor-based!**

- An **actor is a process** with an **implicit input channel**
- The channel acts as a **FIFO mailbox** (as in the Akka framework)
- The actor DSL is **syntactic sugar on the process DSL**

Payoffs:

- we have **almost no actor-specific code**
- we **preserve the connection** to the underlying **theory**

How can we run our DSLs?

```
def payment(aud: ActorRef[Audit[_]]): Actor[Pay, _] =
  forever {
    read { pay: Pay =>
      if (pay.amount > 42000) {
        send(pay.replyTo, Rejected())
      } else {
        send(aud, Audit(pay)) >>
        send(pay.replyTo, Accepted())
      }
    }
  }
```

Naive approach: run each actor/process in a **dedicated thread**

How can we run our DSLs?

```scala
def payment(aud: ActorRef[Audit[_]]): Actor[Pay, _] =
  forever {
    read { pay: Pay =>
      if (pay.amount > 42000) {
        send(pay.replyTo, Rejected())
      } else {
        send(aud, Audit(pay)) >>
        send(pay.replyTo, Accepted())
      }
    }
  }
```

Naive approach: run each actor/process in a **dedicated thread**

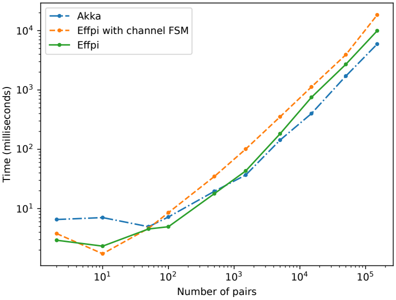
As in our λ-calculus, **continuations are λ-terms** (closures)

For **better scalability**, we can:

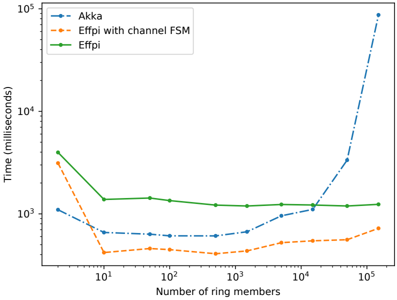
- **schedule** closures to run on a **limited number of threads**
- **unschedule** closures that are **waiting for input**

Scalability and performance

Ping-pong (lower is better)



Streaming ring (lower is better)



The general performance is **not too far from Akka**

- main source of **overhead**: DSL interpretation

4 × Intel Core i7-4790 @ 3.60GHz; 16 GB RAM; Ubuntu 16.04; Java 1.8.0_181; Dotty 0.9.0-RC1; Scala 2.12.6; Akka 2.5.16

Conclusion

Effpi is an experimental framework for **strongly-typed concurrent programming** in **Dotty / Scala 3**

- with **process-based** and **actor-based APIs**
- with a **runtime** supporting **highly concurrent** applications

Theoretical foundations:

- a **concurrent functional calculus**
- equipped with a **novel type system**, blending:
  - **behavioural types** (inspired by π-calculus theory)
  - **dependent function types** (inspired by Dotty / Scala 3)
- verify the **behaviour of processes** by **model checking types**

Conclusion

Effpi is an experimental framework for **strongly-typed concurrent programming** in **Dotty / Scala 3**

- with **process-based** and **actor-based APIs**
- with a **runtime** supporting **highly concurrent** applications

Theoretical foundations:

- a **concurrent functional calculus**
- equipped with a **novel type system**, blending:
  - **behavioural types** (inspired by π-calculus theory)
  - **dependent function types** (inspired by Dotty / Scala 3)
- verify the **behaviour of processes** by **model checking types**

Work in progress:

- **Dotty compiler plugin** to verify **type-level properties** via **model checking**, using `mCRL2`

Appendix

Some references

- D. Sangiorgi and D. Walker, *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- A. Igarashi and N. Kobayashi, "A generic type system for the π-calculus," *TCS*, vol. 311, no. 1, 2004.
- N. Yoshida and M. Hennessy, "Assigning types to processes," *Inf. Comput.*, vol. 174, no. 2, 2002.
- N. Yoshida, "Channel dependent types for higher-order mobile processes," in *POPL*, 2004.
- M. Hennessy, J. Rathke, and N. Yoshida, "safeDpi: a language for controlling mobile code," *Acta Inf.*, vol. 42, no. 4-5, pp. 227–290, 2005.
- D. Ancona *et al.*, "Behavioral Types in Programming Languages," *Foundations and Trends in Programming Languages*, vol. 3(2-3), 2017.
- N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki, "The essence of dependent object types," in *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, 2016.
- L. Cardelli, S. Martini, J. Mitchell, and A. Scedrov, "An extension of System F with subtyping," *Information and Computation*, vol. 109, no. 1, 1994.

Verified mobile code

Modern distributed programming toolkits allow to send/receive **program thunks**, e.g. to:

- execute **user-supplied functions** (e.g., Amazon AWS Lambda)
- perform **remote updates of running code** (e.g., Erlang)

How can we **verify** that **the received thunks behave correctly?**

Verified mobile code

Modern distributed programming toolkits allow to send/receive **program thunks**, e.g. to:

- execute **user-supplied functions** (e.g., Amazon AWS Lambda)
- perform **remote updates of running code** (e.g., Erlang)

How can we **verify** that **the received thunks behave correctly?**

In our theory, if a **program thunk** is received from a channel of type $c^{i}[T]$, we can **deduce its behaviour** by inspecting T

Verified mobile code

Modern distributed programming toolkits allow to send/receive **program thunks**, e.g. to:

- execute **user-supplied functions** (e.g., Amazon AWS Lambda)
- perform **remote updates of running code** (e.g., Erlang)

How can we **verify** that **the received thunks behave correctly?**

In our theory, if a **program thunk** is received from a channel of type $c^{i}[T]$, we can **deduce its behaviour** by inspecting T

E.g., if $T = \Pi(x{:}c^{io}[\mathsf{int}])T'$

- we know that the thunk **needs a channel** x carrying strings
- from T', we can deduce **if and how** the thunk uses x
- from T', we can ensure that the thunk is not a **forkbomb**