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!"

let pinger = λself.λpong.(

**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!"

```latex
let pinger = \text{\textit{self}} . \lambda \text{\textit{pong}} .
\text{send}(\text{\textit{pong}}, \text{\textit{self}}, \lambda_\cdot ()
```
A $\lambda$-calculus with communication & concurrency

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

```
let pinger = \self. \pongc. (send(pongc, self, \_. (recv(self, \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!"

\[
\text{let } pinger = \lambda self. \lambda pongc. ( \\
\text{send}(pong, self, \lambda_. ( \\
\text{recv}(self, \lambda reply. ( \\
\text{end } ))) ) )
\]
A $\lambda$-calculus with communication & concurrency

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

```
let pinger = \self. \pongc.( send(pongc, self, \_.( recv(self, \reply.( end )))))

let ponger = \self.( recv(self, \reqc.( send(reqc, "Hello!", \_.( end ))))))
```
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!"

```
let pinger = λself.λpongc.(send(pongc, self, λ_.(recv(self, λreply.(end )))))
let ponger = λself.(recv(self, λreqc.(send(reqc, "Hello!", λ_.(end )))))

let pingpong = λc1.λc2.(pinger c1 c2 | ponger 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!"

\[
\begin{align*}
\text{let } pinger &= \lambda \text{self}. \lambda \text{pongc}. ( \\
& \quad \text{send}(\text{pongc}, \text{self}, \lambda_. ( \\
& \quad \quad \text{recv}(\text{self}, \lambda \text{reply}. ( \\
& \quad \quad \quad \text{end } ))) ) ) \\
\text{let } ponger &= \lambda \text{self}. ( \\
& \quad \text{recv}(\text{self}, \lambda \text{reqc}. ( \\
& \quad \quad \text{send}(\text{reqc}, \text{"Hello!"}, \lambda_. ( \\
& \quad \quad \quad \text{end } )) ) ) ) \\
\text{let } pingpong &= \lambda \text{c1}. \lambda \text{c2}. ( \ pinger \ \text{c1} \ \text{c2} \mid \ ponger \ \text{c2} ) \\
\text{let } main &= \text{let } \text{c1} = \text{chan}(); \ \text{let } \text{c2} = \text{chan}(); \ \text{pingpong} \ \text{c1} \ \text{c2}
\end{align*}
\]
A $\lambda$-calculus with communication & concurrency

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

```latex
let pinger = \self. \pongc. (send(pongc, self, \_. (recv(self, \reply. (end ))))))

let ponger = \self. (recv(self, \reqc. (send(reqc, "Hello!", \_. (end ))))))

let pingpong = \c1. \c2. (pinger c1 c2 \parallel ponger c2)

let main = let c1 = chan(); let c2 = chan(); pingpong c1 c2
```

Monadic encoding of the higher-order $\pi$-calculus

- $\lambda$-terms model abstract processes
- Continuations are expressed as $\lambda$-terms
How to type a process calculus

For typing, we use a context $\Gamma$ with channel types. E.g.:

$$\Gamma = x:str, y:c^0[str]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash "Hello " \leftrightarrow x : str$$
How to type a process calculus

For typing, we use a context $\Gamma$ with channel types. E.g.:

$$\Gamma = x : \text{str}, y : \text{c}^0[\text{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash "\text{Hello } " ++ x : \text{str}$$

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

Classic approach: $\Gamma \vdash t : \text{proc}$ ("$t$ is a well-typed process in $\Gamma$")
How to type a process calculus

For typing, we use a context $\Gamma$ with \textit{channel types}. E.g.:

$$\Gamma = x : \text{str}, \ y : \text{c}[\text{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash "\text{Hello }" \ ++ \ x : \text{str}$$

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

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

- \textbf{Our approach:} $\Gamma \vdash t : T$ ("$t$ behaves as $T$ in $\Gamma$")
How to type a process calculus

For typing, we use a context $\Gamma$ with channel types. E.g.:

$$\Gamma = x : \text{str}, y : c^\circ[\text{str}]$$

Typing judgements are (partly) standard:

$$\Gamma \vdash "Hello " \leftrightarrow x : \text{str}$$

How do we type communication? E.g., if $t = \text{send}(y, x, \lambda_. \text{end})$

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

Our approach: $$\Gamma \vdash t : T \quad ("t \text{ behaves as } T \text{ in } \Gamma")$$
$$\Gamma \vdash T \leq \text{proc} \quad ("T \text{ is a refined process type"})$$
Behavioural types (inspired by π-calculus theory)

Some examples:

\[ x : \text{str}, \ y : c^o[\text{str}] \vdash \text{send}(y, x, \lambda_.\text{end}) : T \]
Behavioural types (inspired by $\pi$-calculus theory)

Some examples:

\[
x \colon \text{str}, \ y \colon \text{c}^\circ[\text{str}] \vdash \text{send}(y, x, \lambda_.\text{end}) \quad : \ T = \text{o}[\text{c}^\circ[\text{str}], \text{str}, \text{nil}]$
\]
Behavioural types (inspired by $\pi$-calculus theory)

Some examples:

$$x : \text{str}, \ y : \text{c}^0[\text{str}] \vdash \text{send}(y, x, \lambda_. \text{end}) : T = \text{o}[\text{c}^0[\text{str}], \text{str}, \text{nil}]$$

$$\emptyset \vdash \lambda x. \lambda y. \text{send}(y, x, \lambda_. \text{end}) : T'$$
Behavioural types (inspired by π-calculus theory)

Some examples:

\[ x : \text{str}, y : \text{co[\text{str}]} \vdash \text{send}(y, x, \lambda\_\cdot\text{end}) : T = \text{o[co[\text{str}], \text{str}, \text{nil}]} \]

\[ \emptyset \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda\_\cdot\text{end}) : T' = \text{str} \to \text{co[\text{str}] \to T} \]
**Behavioural types** (inspired by $\pi$-calculus theory)

Some examples:

$x : \text{str}, y : c^o[\text{str}] \vdash \text{send}(y, x, \lambda_. \text{end}) : T = o[c^o[\text{str}], \text{str}, \text{nil}]$

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

Can we use types to specify and verify process behaviours?
Behavioural types (inspired by π-calculus theory)

Some examples:

\[
\begin{align*}
    x : \text{str}, y : \text{c}^\circ[\text{str}] & \vdash \text{send}(y, x, \lambda \_ . \text{end}) & : T = \text{o}[\text{c}^\circ[\text{str}], \text{str}, \text{nil}] \\
    \emptyset & \vdash \lambda x . \lambda y . \text{send}(y, x, \lambda \_ . \text{end}) & : T' = \text{str} \rightarrow \text{c}^\circ[\text{str}] \rightarrow T
\end{align*}
\]

Can we use types to specify and verify process behaviours?
Yes — almost!
Behavioural types (inspired by $\pi$-calculus theory)

Some examples:

$x : \text{str}, y : c^o[\text{str}] \vdash \text{send}(y, x, \lambda_. \text{end}) : T = o[c^o[\text{str}], \text{str}, \text{nil}]$

$\emptyset \vdash \lambda x. \lambda y. \text{send}(y, x, \lambda_. \text{end}) : T' = \text{str} \rightarrow c^o[\text{str}] \rightarrow T$

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:

\[ x : \text{str}, \ y : \text{c}^0[\text{str}] \vdash \text{send}(y, x, \lambda \_\text{.end}) \quad : \quad T = \text{o}[\text{c}^0[\text{str}], \text{str}, \text{nil}] \]

\[ \emptyset \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda \_\text{.end}) \quad : \quad T' = \text{str} \rightarrow \text{c}^0[\text{str}] \rightarrow T \]

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:

\[ \lambda x.\lambda y.(\text{let } z = \text{chan}(); \text{send}(z, "Hello!", \lambda \_\text{.end})) \]
Behavioural types

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

\[ T' = \text{str} \rightarrow \text{c}^\circ[\text{str}] \rightarrow \text{o}[\text{c}^\circ[\text{str}], \text{str}, \text{nil}] \]
Behavioural types and dependent function types

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

\[ T' = \text{str} \rightarrow \text{c}[\text{str}] \rightarrow \text{o}[\text{c}[\text{str}], \text{str}, \text{nil}] \]

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

\[ \Pi(x:T_1)T_2 \]

where the return type \( T_2 \) can refer to \( x \)
Behavioural types and dependent function types

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

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

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 \( T'' = \Pi(x:\text{str}) \Pi(y:\text{c}^{\circ}[\text{str}]) \text{o}[y, x, \text{nil}] \)

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

\[ T' = \text{str} \rightarrow \text{c}^\circ[\text{str}] \rightarrow \text{o}[\text{c}^\circ[\text{str}], \text{str}, \text{nil}] \]

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 \( T'' = \Pi(x:\text{str}) \Pi(y:\text{c}^\circ[\text{str}]) \text{o}[y, x, \text{nil}] \)

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:

\[ \varnothing \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda_.\text{end}) : T'' \]
Behavioural types and dependent function types

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

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

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 \( T'' = \Pi(x:\text{str}) \Pi(y:\text{c}^\circ[\text{str}]) \text{o}[y, x, \text{nil}] \)

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**:

\[ \emptyset \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda_.\text{end}) : T'' \leq T' \]
Behavioural types and dependent function types

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

\[ T' = \text{str} \rightarrow \text{co}[\text{str}] \rightarrow \text{o}[\text{co}[\text{str}], \text{str}, \text{nil}] \]

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 \( T'' = \Pi(x:\text{str}) \Pi(y:\text{co}[\text{str}]) \text{o}[y, x, \text{nil}] \)

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:

\[ \emptyset \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda_.\text{end}) : T'' \leq T' \leq \text{co}[\text{none}] \rightarrow \text{str} \rightarrow \text{proc} \]
Types as behavioural specifications: examples

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

\[ T_1 = \Pi(x: \ldots) \Pi(y: \ldots) o[ y, x, i[ x, \Pi(z: \ldots) nil ] ] \]

“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.:

\[ T_1 = \Pi(x:\ldots) \Pi(y:\ldots) o[y, x, i[x, \Pi(z:\ldots) nil]] \]

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

\[ T_2 = \Pi(x:\ldots) i[x, \Pi(y:\ldots) o[y, str, nil]] \]

“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.:

\[
T_1 = \Pi(x:\ldots) \Pi(y:\ldots) o[y, x, i[x, \Pi(z:\ldots) \text{nil}]]
\]

“Take \(x\) and \(y\); use \(y\) send \(x\); use \(x\) to receive some \(z\); and terminate”

\[
T_2 = \Pi(x:\ldots) i[x, \Pi(y:\ldots) o[y, \text{str}, \text{nil}]]
\]

“Take \(x\); use \(x\) to input some \(y\); use \(y\) to send a \text{string}; and terminate”

- \(T_1\) and \(T_2\) are respectively the types of the \textit{pinger} and \textit{ponger} processes
Types as behavioural specifications: examples

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

\[ T_1 = \Pi(x:\ldots) \Pi(y:\ldots) o[y, x, i[x, \Pi(z:\ldots) \text{nil}]] \]

“Take \( x \) and \( y \); use \( y \) send \( x \); use \( x \) to receive some \( z \); and terminate”

\[ T_2 = \Pi(x:\ldots) i[x, \Pi(y:\ldots) o[y, \text{str}, \text{nil}]] \]

“Take \( x \); use \( x \) to input some \( y \); use \( y \) to send a \text{string}; and terminate”

- \( T_1 \) and \( T_2 \) are respectively the types of the \textit{pinger} and \textit{ponger} processes

\[ T_3 = \Pi(x:\ldots) \Pi(y:\ldots) p[T_1 x y, T_2 y] \]

“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.:

\[ T_1 = \Pi(x: \ldots) \Pi(y: \ldots) o[y, x, i[x, \Pi(z: \ldots) \text{nil}]] \]

“Take \(x\) and \(y\); use \(y\) send \(x\); use \(x\) to receive some \(z\); and terminate”

\[ T_2 = \Pi(x: \ldots) i[x, \Pi(y: \ldots) o[y, \text{str}, \text{nil}]] \]

“Take \(x\); use \(x\) to input some \(y\); use \(y\) to send a \text{str}ing; and terminate”

\[ T_3 = \Pi(x: \ldots) \Pi(y: \ldots) p[T_1 x y, T_2 y] \]

“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 \text{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: \( \Pi(x:\ldots) \Pi(y:\ldots) p[T_1 x y, T_2 y] \)

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: \( \Pi(x:\ldots) \Pi(y:\ldots) p[T_1 x y, T_2 y] \)

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 \( \vdash t : T \) 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: \( \Pi(x:\ldots) \Pi(y:\ldots) p[T_1 x y, T_2 y] \)

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 \( \vdash t : T \) 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:

\[ \Pi(x: \text{str}) \Pi(y: \text{c}^[[\text{str}])] \circ [y, x, \text{nil}] \]

\[ \downarrow \]

\((x: \text{String}, y: \text{OChan}[\text{String}]) \Rightarrow \text{Out}[y.\text{type}, x.\text{type}, \text{Nil}]\)
From theory to Dotty / Scala3

We directly translate our types in Dotty / Scala 3:

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

$\Downarrow$

$$(x:\text{String}, y:\text{OChan[String]}) \Rightarrow \text{Out}[y.\text{type}, x.\text{type}, \text{Nil}]$$

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?

Naive approach: run each actor/process in a dedicated thread
How can we run our DSLs?

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

💡 As in our $\lambda$-calculus, continuations are $\lambda$-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

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 $\pi$-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 $\pi$-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


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 $T = \Pi(x:c^{io}[int])T'$

- 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