Effpi

concurrent programming with dependent behavioural types

Alceste Scalas
(with Elias Benussi & Nobuko Yoshida)

Imperial College London

University of Novi Sad — 17 September 2018

http://mrg.doc.ic.ac.uk

Mobility Research Group

π-calculus, Session Types research at Imperial College

Home | People | Publications | Grants | Talks | Tutorials | Tools | Awards | Kohei Honda

NEWS

The paper *Multiparty asynchronous session types* by Kohei Honda, Nobuko Yoshida, and Marco Carbone, published in POPL 2008 has been awarded the ACM SIGPLAN Most Influential POPL Paper Award today at POPL 2018.

» more

10 Jan 2018

Estafet has published a page on their usage of the Scribble language developed in our group with RedHat and other industry partners.

» more

25 Sep 2017

Nick spoke at Golang UK 2017 on applying behavioural types to verify concurrent Go programs.

SELECTED PUBLICATIONS

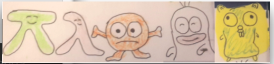
2018

Julien Lange , Nicholas Ng , Bernardo Toninho , Nobuko Yoshida : A Static Verification Framework for Message Passing in Go using Behavioural Types. *To appear in* ICSE 2018 .

Bernardo Toninho , Nobuko Yoshida : Depending On Session Typed Process. *To appear in* FoSSaCS 2018 .

Bernardo Toninho , Nobuko Yoshida : On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings. *To appear in* ESOP 2018 .

Rumyana Neykova , Raymond Hu , Nobuko Yoshida , Fahd Abdeljallal : Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. *To appear in* CC 2018 .



Post-docs:

Simon CASTELLAN

David CASTRO

Francisco FERREIRA

Raymond HU

Rumyana NEYKOVA

Nicholas NG

Alceste SCALAS

PhD Students:

Assel ALTAYEVA

Juliana FRANCO

Eva GRAVERSEN

POPL 2008 MOST INFLUENTIAL PAPER AWARD

www.scribble.org

Home Getting Started Downloads Documentation ▾ Community ▾

Scribble: Describing Multi Party Protocols

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Describe

Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

Verify

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

Project

Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

Implement

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

Monitor

Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.

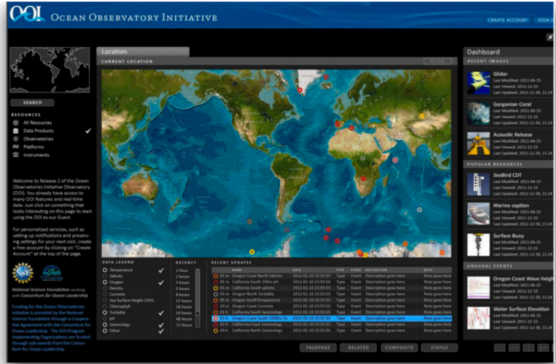
Online tool : http://scribble.doc.ic.ac.uk/

```
module examples;

global protocol HelloWorld(role Me, role World) {
  hello() from Me to World;
  choice at World {
    goodMorning1() from World to Me;
  } or {
    goodMorning1() from World to Me;
  }
}
```

Load a sample | Check | Protocol: examples.HelloWorld | Role: Me | Project | Generate Graph

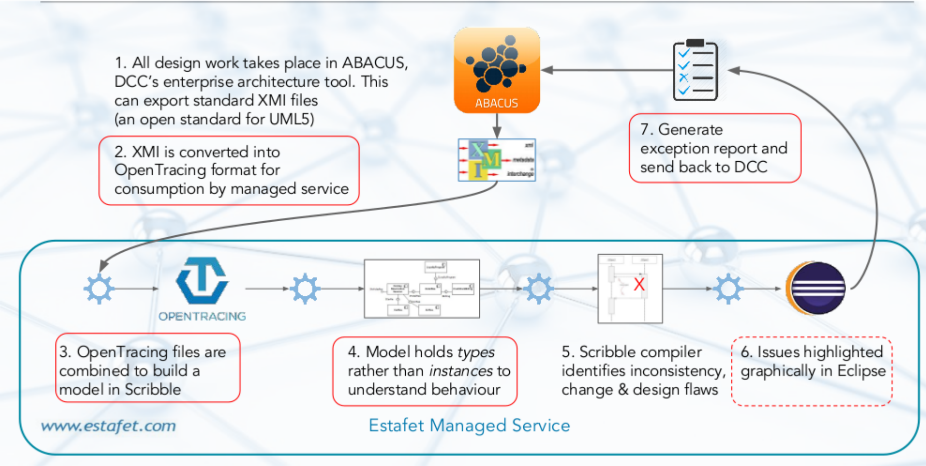
OOI Collaboration



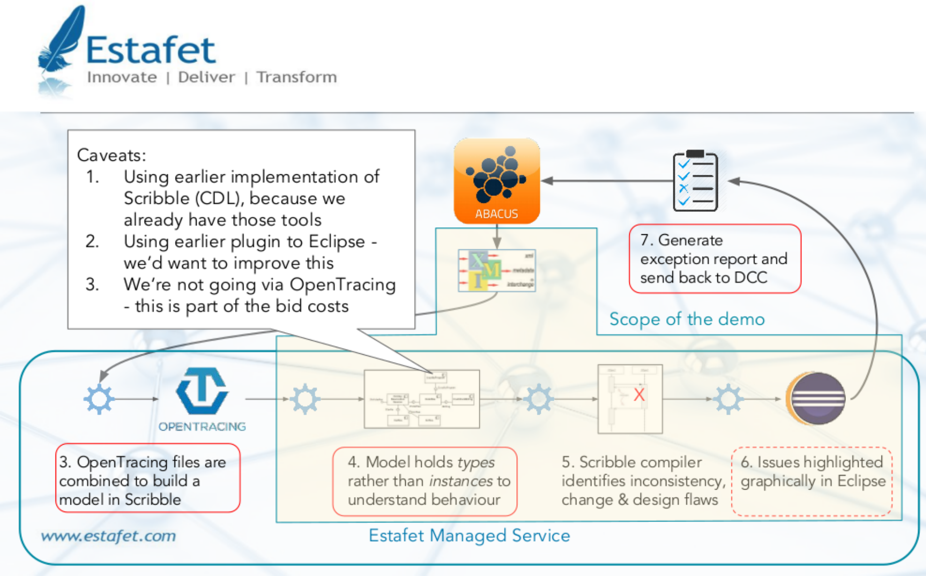
- **TCS'16:** Monitoring Networks through Multiparty Session Types. Laura Bocchi , Tzu-Chun Chen , Romain Demangeon , Kohei Honda , Nobuko Yoshida
- **LMCS'16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida
- **FMSD'15:** Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon , Kohei Honda , Raymond Hu , Rumyana Neykova , Nobuko Yoshida
- **TGC'13:** The Scribble Protocol Language. Nobuko Yoshida , Raymond Hu , Rumyana Neykova , Nicholas Ng

End-to-End Switching Programme by DCC

Estafet
Innovate | Deliver | Transform



End-to-End Switching Programme by DCC



CC'18

A Session Type Provider

Compile-Time API Generation of Distributed Protocols with Refinements in F#

Rumyana Neykova
Imperial College London
United Kingdom

Raymond Hu
Imperial College London
United Kingdom

Nobuko Yoshida
Imperial College London
United Kingdom

Fahd Abdeljallal
Imperial College London
United Kingdom

Abstract

We present a library for the specification and implementation of distributed protocols in native F# (and other .NET languages) based on multiparty session types (MPST). There are two main contributions. Our library is the first practical development of MPST to support what we refer to as *interaction refinements*: a collection of features related to the refinement of *protocols*, such as message-type refinements (value constraints) and message-value dependent control flow. A well-typed endpoint program using our library is ...ranteed to perform only compliant session I/O actions ... the refined protocol, up to premature termination. ... our library is developed as a session *type provider*,

1 Introduction

Type providers [20, 27] are a .NET feature for a form of compile-time meta programming, designed to bridge between programming in statically typed languages such as F# and C#, and working with so-called *information spaces*—structured data sources such as SQL databases or XML data.

A type provider works as a compiler plugin that performs on-demand generation of *types*: it takes a schema for an external information space, and generates types that allow the data to be manipulated via a strongly-typed interface, with benefits such as static error detection and IDE auto-completion. For example, an instantiation of the in-built type provider for WSDL Web services [6] may look like

Graydon Hoare @graydon_pub

(This stuff is _fantastic_)

11:31 PM - 11 Mar 2018

32 Retweets 83 Likes

shots fired @zeeshanlakhani · Mar 12

Replying to @graydon_pub @dsyme

Awesome!

Brendan Zabarauskas @brendanzab ·

Replying to @graydon_pub

This stuff fills me with hope!

Ryan Riley @panesofglass · Mar 12

Replying to @graydon_pub

This is amazing! I guess I need to switch

weekly

F# eXchange

Behavioural Type-Based Static Verification Framework for

Go concurrency verification research at DoC grabs headline

currency
erates a
tured in the
hich
nteresting
easily
e of the
L (Principles

the morning paper

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Home About InfoQ QR Editions Subscribe

A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

tags: Concurrency, Programming Languages

A static verification framework for message passing in Go using behavioural types Lange et al., *ICSE 18*

With thanks to Alexis Richardson who first forwarded this paper to me.

We're jumping ahead to ICSE 18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we've been exploring this week though, so I thought I'd cover it now. We've seen verification techniques applied in the context of Rust and JavaScript, looked at the integration of linear types in Haskell, and today it is the turn of Go!

SUBSCRIBE

never miss an issue! The Morning Paper delivered straight to your inbox.

SEARCH

type and press enter

ARCHIVES

Select Month

MOST READ IN THE LAST FEW DAYS

Selected Publications 2017/2018

- **[LICS'18]** Romain Demangeon, NY: Casual Computational Complexity of Distributed Processes.
- **[CC'18]** Rumyana Neykova , Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
- **[FoSSaCS'18]** Bernardo Toninho, NY: Depending On Session Typed Process.
- **[ESOP'18]** Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings.
- **[ESOP'18]** Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu , Lukasz Ziarek: A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems.
- **[ICSE'18]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY : A Static Verification Framework for Message Passing in Go using Behavioural Types
- **[ECOOP'17]** Alceste Scala, Raymond Hu, Ornela Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming..
- **[COORDINATION'17]** Keigo Imai, NY, Shoji Yuen: Session-ocaml: a session-based library with polarities and lenses.
- **[FoSSaCS'17]** Julien Lange, NY: On the Undecidability of Asynchronous Session Subtyping.
- **[FASE'17]** Raymond Hu, NY: Explicit Connection Actions in Multiparty Session Types.
- **[CC'17]** Rumyana Neykova, NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- **[POPL'17]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY: Fencing off Go: Liveness and Safety for Channel-based Programming.

Selected Publications 2017/2018

- **[LICS’18]** Romain Demangeon, NY: Casual Computational Complexity of Distributed Processes.
- **[CC’18]** Rumyana Neykova , Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
- **[FoSSaCS’18]** Bernardo Toninho, NY: Depending On Session Typed Process.
- **[ESOP’18]** Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings.
- **[ESOP’18]** Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu , Lukasz Ziarek: A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems.
- **[ICSE’18]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY : A Static Verification Framework for Message Passing in Go using Behavioural Types.
- **[ECOOP’17]** Alceste Scala, Raymond Hu, Ornela Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming.
- **[COORDINATION’17]** Keigo Imai, NY, Shoji Yuen: Session-ocaml: a session-based library with polarities and lenses.
- **[FoSSaCS’17]** Julien Lange, NY: On the Undecidability of Asynchronous Session Subtyping.
- **[FASE’17]** Raymond Hu, NY: Explicit Connection Actions in Multiparty Session Types.
- **[CC’17]** Rumyana Neykova, NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- **[POPL’17]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY: Fencing off Go: Liveness and Safety for Channel-based Programming.

Effpi

concurrent programming with dependent behavioural types

Alceste Scalas
(with Elias Benussi & Nobuko Yoshida)

Imperial College London

University of Novi Sad — 17 September 2018

Example: payment service with auditing

A scenario in **message-passing concurrency**

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. restart 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]
```

Behind the scenes

What you have seen is based on:

- a **concurrent functional calculus**
- equipped with a **novel type system**:
  - **behavioural types** (inspired by π-calculus theory)
  - **dependent function types** (inspired by Dotty / Scala 3)
- **implemented in Dotty / Scala 3** (via deep embedding)
  - also offering a simplified **actor-based API**
  - with a **runtime** supporting **highly concurrent** applications

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!"`

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

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!"`

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

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!"`

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

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!"`

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

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

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

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!"`

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

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

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

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

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!"`

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

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

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

- λ-terms model **abstract processes**
- **Continuations** are expressed as λ-terms (monadic style)

How to type a process calculus

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

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

Therefore, we have classic **typing judgements**:

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

How to type a process calculus

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

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

Therefore, we have classic **typing judgements**:

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

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")

How to type a process calculus

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

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

Therefore, we have classic **typing judgements**:

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

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: $\Gamma \vdash t : \mathsf{T}$ (**"t behaves as T in Γ"**)

How to type a process calculus

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

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

Therefore, we have classic **typing judgements**:

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

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")

Our approach: $\Gamma \;\vdash\; \mathrm{t} \;:\; \mathsf{T}$ (**"** t **behaves as** T **in** Γ**"**)

$\Gamma \;\vdash\; \mathsf{T} \leqslant \mathbf{proc}$ (**"** T **is a refined process type"**)

Behavioural types

Some examples:

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

Behavioural types

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

Some examples:

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

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

Behavioural types

Some examples:

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

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

Behavioural types

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**?

Behavioural types

Some examples:

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

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

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

Behavioural types

Some examples:

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

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

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

Some examples:

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

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

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**

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

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

$\Pi(x{:}T_1)T_2$ 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**

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

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**

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

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

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

Behavioural types and dependent function types

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

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

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

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**:

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

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.:

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

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

$$T_2 \;=\; \Pi(x{:}\ldots)\,\mathbf{i}[\,x\,,\,\Pi(y{:}\ldots)\,\mathbf{o}[y,\mathsf{str},\mathbf{nil}]\,]$$

"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

But our types also allow for **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 as behavioural specifications (cont'd)

Type checking guarantees **type safety**

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

But our types also allow for **rich behavioural specifications** that can be **complicated**, especially when **composed**. . .

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

. . . and they 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

But our types also allow for **rich behavioural specifications** that can be **complicated**, especially when **composed**. . .

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

. . . and they can model **races** on shared channels, and **deadlocks**

- Give a **labelled semantics** to a type T
- Verify **safety/liveness** properties of T via **model checking**
- Show that 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

But our types also allow for **rich behavioural specifications** that can be **complicated**, especially when **composed**. . .

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

. . . and they can model **races** on shared channels, and **deadlocks**

- Give a **labelled semantics** to a type T
- Verify **safety/liveness** properties of T via **model checking**
- Show that 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)

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

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

- 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**

From theory to Dotty / Scala3

We **directly translate our types in Dotty**:

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

$$\Downarrow$$

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

From theory to Dotty / Scala3

We **directly translate our types in Dotty**:

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

$$\Downarrow$$

```
(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 **very little 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())
      }
    }
  }
```

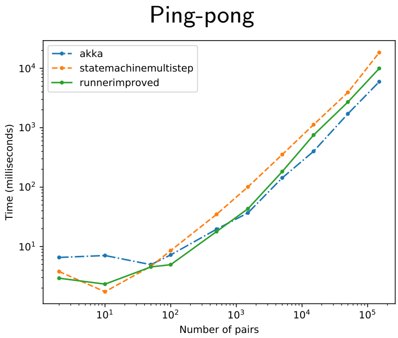
Naive approach: run each actor/process in a **dedicated thread**

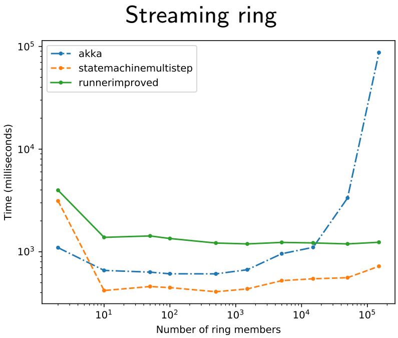
As in our λ-calculus, **continuations are λ-terms** (closures)

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





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

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**:
  - **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**:
  - **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.