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

NEWS

SELECTED PUBLICATIONS

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

Kohei Honda, Nobuko Yoshida and Marco Carbone

Multiparty asynchronous session types
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.
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;
  }
}
OII 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

1. All design work takes place in ABACUS, DCC’s enterprise architecture tool. This can export standard XMI files (an open standard for UML5).

2. XMI is converted into OpenTracing format for consumption by managed service.

3. OpenTracing files are combined to build a model in Scribble.

4. Model holds types rather than instances to understand behaviour.

5. Scribble compiler identifies inconsistency, change & design flaws.

6. Issues highlighted graphically in Eclipse.

7. Generate exception report and send back to DCC.

www.estafet.com

Estafet Managed Service
End-to-End Switching Programme by DCC

Caveats:
1. Using earlier implementation of Scribble (CDL), because we already have those tools
2. Using earlier plugin to Eclipse - we'd want to improve this
3. We're not going via OpenTracing - this is part of the bid costs

Scope of the demo

3. OpenTracing files are combined to build a model in Scribble
4. Model holds types rather than instances to understand behaviour
5. Scribble compiler identifies inconsistency, change & design flaws
6. Issues highlighted graphically in Eclipse
7. Generate exception report and send back to DCC
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 guaranteed to perform only compliant session I/O actions on 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 [5] may look like

shots fired @zeeshanlakhani · Mar 12
Replying to @graydon_pub @dsyme
Awesome!

Brendan Zabarauskas @brendanzab · Mar 12
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
Behavioural Type-Based Static Verification Framework for GO

Julian Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

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!
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#.
- **[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, Ornella 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

- [CC’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
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: \texttt{Out[ActorRef[Result], Accepted]}

required: \texttt{Out[ActorRef[Result](pay.replyTo), Rejected]}
|\texttt{Out[ActorRef[Audit[_])(aud), Audit[Pay(pay)]]} \triangleright>: \texttt{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 \( \pi \)-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 $\lambda$-calculus with communication & concurrency

Example: a *ping*er process sends a communication channel to a *pon*ger process, who uses the channel to reply "Hello!"
A \( \lambda \)-calculus with communication & concurrency

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

\[
\text{let } pinger = \lambda \text{self}. \lambda \text{pongc}. (
\]

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!"

\[
\text{let } pinger = \lambda \textit{self}. \lambda \textit{pong}c. (send(pong\textit{c}, \textit{self}, \lambda \_.)}
\]
A \( \lambda \)-calculus with communication & concurrency

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

\begin{verbatim}
let pinger = \lambda self . \lambda pongc .
  send(pongc, self, \lambda .
    recv(self, \lambda reply .

let ponger = \lambda self .
  recv(self, \lambda reqc .
    send(reqc, self, "Hello!", \lambda .

let pingpong = \lambda c1 . \lambda c2 .
  /\langle . alt1
  pinger c1 c2
  /\rangle . alt0

let main = let c1 = chan();
  let c2 = chan();
  pingpong c1 c2
/
\end{verbatim}
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 = \lambda self. \lambda pongc. (send(pongcr, self, \lambda_. (recv(self, \lambda reply. (end))))))
```

"λ-terms model abstract processes. Continuations are expressed as \( \lambda \)-terms (monadic style)."
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 = λself.λpongc.(send(pongc, self, λ_.(recv(self, λreply.(end ))))))

let ponger = λself.(recv(self, λreqc.(send(reqc, "Hello!", λ_.(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.(let ponger = λself.(
    send(pongc, self, λ_.(recv(self, λreply.(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!"

\[
\text{let } \text{pinger} = \lambda \text{self}. \lambda \text{pongc}. ( \\
\quad \text{send}(\text{pongc}, \text{self}, \lambda_. ( \\
\quad \quad \text{recv}(\text{self}, \lambda \text{reply}. ( \\
\quad \quad \quad \text{end } )))) ) \\
\text{let } \text{ponger} = \lambda \text{self}. ( \\
\quad \text{recv}(\text{self}, \lambda \text{reqc}. ( \\
\quad \quad \text{send}(\text{reqc}, "Hello!", \lambda_. ( \\
\quad \quad \quad \text{end } )))) ) \\
\text{let } \text{pingpong} = \lambda \text{c1}. \lambda \text{c2}. ( \text{pinger c1 c2 | ponger c2 } ) \\
\text{let } \text{main} = \text{let c1 = chan(); let c2 = chan(); pingpong c1 c2}
\]
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 = $\lambda$self.$\lambda$pongc.(send(pongc, self, $\lambda$_.(recv(self, $\lambda$reply.(end ))))))

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

let pingpong = $\lambda$c1.$\lambda$c2.(pinger c1 c2 | ponger c2)

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

- $\lambda$-terms model **abstract processes**
- **Continuations** are expressed as $\lambda$-terms (monadic style)
How to type a process calculus

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

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

Therefore, we have classic typing judgements:

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

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

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

Therefore, we have classic typing judgements:

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

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** $\Gamma$ and **channel types**. E.g.:

$$\Gamma = x : \text{str}, y : \text{c}_0[\text{str}]$$

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

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

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

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

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

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

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

Therefore, we have classic typing judgements:

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

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 $\Gamma$")

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

$\Gamma \vdash T \leq \text{proc}$ ("$T$ is a refined process type")
Behavioural types

Some examples:

\[ x : \text{str}, \ y : \co[\text{str}] \vdash \text{send}(y, x, \lambda.\text{end}) \quad : \ T \]
Behavioural types

Some examples:

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

Some examples:

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

\( \emptyset \vdash \lambda x . \lambda y . \text{send}(y, x, \lambda_. \text{end}) \quad : \quad T' \)
Behavioural types

Some examples:

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

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

Some examples:

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

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

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

Some examples:

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

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

Can we use types to specify and verify process behaviours?

Yes — almost!
**Behavioural types**

Some examples:

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

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

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:

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

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

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:

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

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

\[ T' = \text{str} \to \text{co}[\text{str}] \to \text{o}[\text{co}[\text{str}], \text{str}, \text{nil}] \]
Behavioural types and dependent function types

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

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

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

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

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
Behavioural types and dependent function types

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

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

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:
\[ \emptyset \vdash \lambda x.\lambda y.\text{send}(y, x, \lambda_.\text{end}) : T'' \]
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**:

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

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

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

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

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

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

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

"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”

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

“Take \(x\); use \(x\) to input some \(y\); use \(y\) to send a \text{string}; and terminate”
Types as behavioural specifications: examples

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

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

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

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

“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 *pinger* and *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 string; and terminate”

- \(T_1\) and \(T_2\) are respectively the types of the pinger and 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.:

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

"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 \texttt{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"

- \(T_3\) is the type of the \textit{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: \( \Pi(x:\ldots) \Pi(y:\ldots) p[ T_1 x y, T_2 y ] \)
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: $\Pi(x:\ldots) \Pi(y:\ldots) p[ T_1 x y , T_2 y ]$

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

...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 \( \vdash t : T \) 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

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

- E.g., the *pingpong* type: \(\Pi(x:\ldots) \Pi(y:\ldots) p[T_1 x y, T_2 y]\)

...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 \(\vdash t : T\) 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 \( c^i[T] \), 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
From theory to Dotty / Scala3

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

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

\[
\downarrow
\]

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

We directly translate our types in Dotty:

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

\[
\downarrow
\]

(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?

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

```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())
            }
        }
    }
}
```
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

💡 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
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 \( \pi \)-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


