hybrid protocol conformance verification for binary sessions

Hernán Melgratti, Universidad de Buenos Aires
Luca Padovani, Università di Torino

Pre- and Post-Deployment Verification Techniques 2017, Torino

protocol conformance with types and contracts

A hybrid approach

- two different mechanisms with different expressive power
- early (compile time) vs late (execution time) verification

Self-imposed constraints

- (almost) no additional tools or skills required
- do everything with the language and its compiler

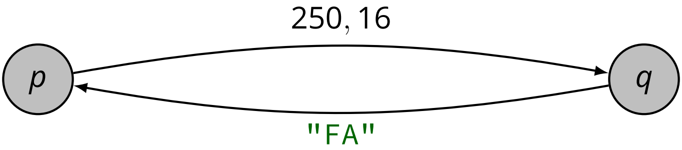
The language

- `OCaml`, no prior knowledge assumed in this talk
- approach portable to other languages, restrictions may apply

outline

outline

problem statement



Given a program with two threads ***p*** and ***q*** where

- *p* and *q* exchange messages over **one channel**
- *p* sends to *q* two integer numbers *n* and *b*
- *q* sends to *p* the string representation of *n* in base *b*

verify that

- *p* and *q* conform to (some) **protocol** and
- possibly find out who's to **blame** if this isn't the case

standard channels are too restrictive

API of **standard** channels

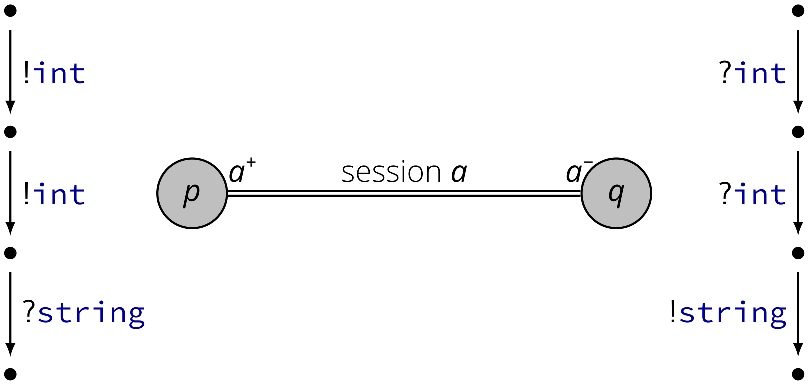
```
val send    : α → α t → unit
val receive : α t → α
```

Code of `Client` module

```
send 250 c;
send 16 c;
let s = receive c in
print_string s              (* type error! *)
```

- standard channels are **uniformly** typed

specifying protocols using session types



- **session type** = protocol specification as a type (≈ FSA)
- **peer** endpoints ⇒ **dual** session types

syntax of session types

Session type	T	$::=$	`end`	no more interactions
			$!t.T$	output
			$?t.T$	input
			A	session type variable
			$\cdots$	branching/recursion/…
Type	t	$::=$	`int` $\cdots$	basic types
			α	type variable
			T	session type
			$\cdots$	

the client

API of binary sessions

```
val send    : α → !α.A → A
val receive : ?α.A → α * A
val close   : end → unit
```

Code of `Client` module

```
let main c =
  let     c = send 250 c in          c:!int.!int.?string
  let     c = send 16 c in           c:!int.?string
  let s, c = receive c in            c:?string
  print_endline s; close c           c:end
```

Note

- c must be used **linearly** throughout the code

the server

Code of `Server` module

```
let main c =
  let n, c = receive c in            c:?int.?int.!string
  let b, c = receive c in            c:?int.!string
  let c = send (convert b n) c       c:!string
  in close c                         c:end
```

API for registration and connection

$$\texttt{val register} : (\bar{A} \rightarrow \texttt{unit}) \rightarrow A\ \texttt{server\_t}$$
$$\texttt{val connect} : A\ \texttt{server\_t} \rightarrow A$$

Connecting `Client` and `Server`

```
let server = register Server.main
let _ = Client.main (connect server)
```

demo

properties of well-typed programs

Communication safety

- threads that respect endpoint linearity communicate safely
- linearity violations are detected at runtime (at the latest)

Protocol fidelity

- the order of communications is consistent with the protocol

Progress, to some extent

- 2 threads sharing 1 session don't block on communications
- this property scales to forest-like network topologies

outline

well-typed programs may go wrong

Issues with n

- $n < 0$ — "index out of bounds" exception

Issues with b

- $b < 0$ or $b > 16$ — "index out of bounds" exception
- $b = 0$ — "division by zero" exception
- $b = 1$ — server loops

Issues with the string

- `"0FA"` — leading `0` is unnecessary

By the way, who's to blame for these issues?

- client and server must agree on a **contract**

from session types to **contracts**

- *p* sends to *q* an integer number
- *p* sends to *q* an integer number
- *q* sends to *p* a string

Session type

```
!int.!int.?string.end
```

from session types to **contracts**

- p sends to q an integer number $n \geq 0$
- p sends to q an integer number b such that $2 \leq b \leq 16$
- q sends to p a string s such that $|s| = \lfloor \log_b(n) \rfloor + 1$

Session type

```
!int.!int.?string.end
```

Contract

?

an embedded DSL of contracts

Example of contract definition

```
let client_c =
  send_c
    (flat_c (fun n → n ≥ 0))
    (send_c
       (flat_c (fun b → 2 ≤ b ∧ b ≤ 16))
       any_endpoint_c)
```

Contract constructors

```
flat_c          : (α → bool) → [α]
send_c          : [α] → [A] → [!α.A]
any_endpoint_c  : [A]

client_c        : [!int.!int.A]
```

starting a session with contract agreement

API for registration and connection

```
val register : (Ā → unit) → [A] → string → A server_t
val connect  : A server_t → string → A
```

Connecting `Client` and `Server`

```
let server = register Server.main client_c "Server"
let _ = Client.main (connect server "Client")
```

Notes

- the code of client and server doesn't change (but types do)
- contract/session type consistency is checked at compile time
- `"Client"` and `"Server"` are labels to assign **blames**
- the contract is checked at **runtime**, as the session progresses

demo

how does monitoring work?

$$[e]^{\mathrm{c},p,q}$$

- expressions may be wrapped by a **monitor**
- c is the contract that ***e*** is supposed to satisfy
- ***p*** is responsible for values flowing **out** of ***e***
- ***q*** is responsible for values flowing **into** ***e***

semantics of flat contracts

$$[V]^{\texttt{flat\_c}\ f, p, q}$$

semantics of flat contracts

$[v]^{\texttt{flat\_c}\ f,p,q}$

$\downarrow$

$\texttt{if}\ f\ v\ \texttt{then}\ v\ \texttt{else}\ \texttt{blame}\ p$

$f\ v \to^* \texttt{true}$: v

$f\ v \to^* \texttt{false}$: $\texttt{blame}\ p$

semantics of output contracts

$$\texttt{send}\ v\ [a]^{\texttt{send\_c}\ \mathrm{c}\ \mathrm{d},p,q} \parallel \texttt{receive}\ a$$

semantics of output contracts

$$\texttt{send}\ v\ [a]^{\texttt{send\_c}\ \mathrm{c}\ \mathrm{d},p,q} \parallel \texttt{receive}\ a$$

v **checked** against c

$$\rightarrow \quad [\texttt{send}\ [v]^{\mathrm{c},q,p}\ a]^{\mathrm{d},p,q} \parallel \texttt{receive}\ a$$

semantics of output contracts

$$\texttt{send}\ v\ [a]^{\texttt{send\_c}\ \mathrm{c}\ \mathrm{d},p,q} \quad \| \quad \texttt{receive}\ a$$

v **checked** against c

$$\rightarrow \quad [\texttt{send}\ [v]^{\mathrm{c},q,p}\ a]^{\mathrm{d},p,q} \quad \| \quad \texttt{receive}\ a$$

v **satisfies** c

$$\rightarrow^{*} \quad [\texttt{send}\ v\ a]^{\mathrm{d},p,q} \quad \| \quad \texttt{receive}\ a$$

semantics of output contracts

$$\texttt{send}\ v\ [a]^{\texttt{send\_c}\ \mathrm{c}\ \mathrm{d},p,q} \parallel \texttt{receive}\ a$$

v **checked** against c

$$\rightarrow \quad [\texttt{send}\ [v]^{\mathrm{c},q,p}\ a]^{\mathrm{d},p,q} \parallel \texttt{receive}\ a$$

v **satisfies** c

$$\rightarrow^{*} \quad [\texttt{send}\ v\ a]^{\mathrm{d},p,q} \parallel \texttt{receive}\ a$$

a's contract **updated**

v **sent**

$$\rightarrow \quad [a]^{\mathrm{d},p,q} \parallel (v, a)$$

outline

1. Prologue
2. Protocol conformance using session types
3. Specifying properties of messages using contracts
4. Dependent and higher-order contracts
5. Epilogue

dependent contracts

- p sends to q an integer number $n \geq 0$
- p sends to q an integer number b such that $2 \leq b \leq 16$
- q sends to p a string s such that $|s| = \lfloor \log_b(n) \rfloor + 1$

Note

- contract of s **depends** on messages exchanged **earlier on**
- idea: **compute** the contract for s once we know n and b

specifying dependent contracts

```
let client_c =
  send_d
    (flat_c (fun n → n ≥ 0))
    (fun n →
      send_d
        (flat_c (fun b → 2 ≤ b ∧ b ≤ 16))
        (fun b →
          (receive_c
             (flat_c (fun s → length s == log b n + 1))
             any_endpoint_c)))
```

More contract constructors

```
send_d    : [α] → (α → [A]) → [!α.A]
receive_c : [α] → [A] → [?α.A]
```

demo

properties of blame assignment (1/2)

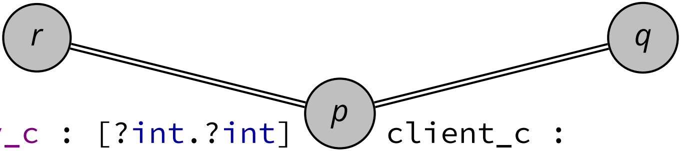
Question

Is it always the case that, if a message triggers a contract violation, the **sender** of the message is always the module to **blame**?

NO

With **higher-order** sessions the module to blame may be different from the sender.

contracts for **higher-order** sessions



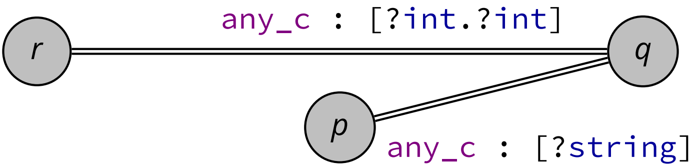
any_c : [?int.?int]

client_c : [!(?int.?int).?string]

```
let client_c =
  send_c
    (receive_c
      (flat_c (fun n → n ≥ 0))
      (receive_c
        (flat_c
          (fun b → 2 ≤ b ∧ b ≤ 16))
        any_endpoint_c))
    any_endpoint_c
```

- *p* **delegates** the session with *r* to *q*
- *p* **lies** to *q*
- *r* sends a message that violates `client_c`
- *q* blames *p*, not *r*

contracts for **higher-order** sessions




```
let client_c =
  send_c
    (receive_c
      (flat_c (fun n → n ≥ 0))
      (receive_c
        (flat_c
          (fun b → 2 ≤ b ∧ b ≤ 16))
        any_endpoint_c))
    any_endpoint_c
```

- *p* **delegates** the session with *r* to *q*
- *p* **lies** to *q*
- *r* sends a message that violates `client_c`
- *q* blames *p*, not *r*

properties of blame assignment (2/2)

Question

Can a module be blamed **by mistake**?

NO

If *p* conforms with what *p* **thinks** is the contract of the endpoints it uses, *p* won't be blamed even if other modules conspire against *p*.

Several names for this property

- blame correctness, blame safety, blame soundness, . . .

outline

related work on sessions

- Hüttel *et al.*, “Foundations of Session Types and Behavioural Contracts”, ACM Computing Surveys, 2016. **OA**
- Bartoletti *et al.*, “Combining behavioural types with security analysis”, Journal of Logical and Algebraic Methods in Programming, 2015.
- Ancona *et al.*, “Behavioral Types in Programming Languages”, Foundations and Trends in Programming Languages, 2016.
- Gay *et al.* (eds), “Behavioural Types: from Theory to Tools”, River Publishers, 2017. **OA**

`www.behavioural-types.eu`

- Luca Padovani, “A Simple Library Implementation of Binary Sessions”, Journal of Functional Programming, 2017.

related work on contracts

Contracts for higher-order functions and mutable objects

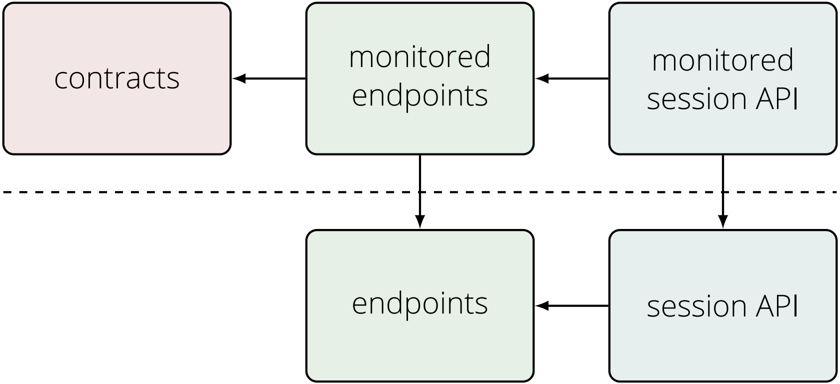
- Findler & Felleisen, “Contracts for Higher-Order Functions”, Proceedings of ICFP, 2002.
- Strickland *et al.*, “Chaperones and impersonators: run-time support for reasonable interposition”, Proceedings of OOPSLA, 2012.

Contracts for higher-order sessions

- Melgratti & Padovani, “Chaperone Contracts for Higher-Order Sessions”, Proceedings of ICFP, 2017. **OA**

implementation

- FuSe available from my home page
- monitoring not integrated yet, but **available** on ACM DL
- **modular** design, portable to other session libraries



wrap-up slide

Hybrid technique based on **types** and **contracts**

- communication safety
- protocol fidelity
- obligations/guarantees on the content of messages

Main highlights

- **low impact** on the programmer's workflow
- **gradual** application of contracts, benign "**blame war**"
- **useful** information to **locate** the source of problems

THANKS!