Towards Mailbox Typing for Erlang

Simon Fowler • **Duncan Paul Attard** • Simon Gay • Phil Trinder

Stardust Meeting • 2023

University of Glasgow

Common protocol errors (unexpected request)

```
id_server() →
  receive
    {init, N} → id_server_loop(N)
  end.

id_server_loop(N) →
  receive
    {get, Client} →
      Client ! {id, N},
      id_server_loop(N + 1);
    {init, _} → error
  end.
```

```
client() →

  % Create server process.
  Server = spawn {id_server, []},

  % Initialize server.
  Server ! {init, 5},
  Server ! {init, 5}, ✗
  Server ! {get, self},
  receive
    {id, Id} → print Id
  end.
```

Common protocol errors (omitted reply)

```
id_server() →
  receive
    {init, N} → id_server_loop(N)
  end.

id_server_loop(N) →
  receive
    {get, Client} →
      Client ! {id, N},
      id_server_loop(N + 1);
    {init, _} → error
  end.
```

```
client() →

  % Create server process.
  Server = spawn {id_server, []},

  % Initialize server.
  Server ! {init, 5},

  Server ! {get, self},
  receive
    {id, Id} → print Id
  end.
```

Common protocol errors (self-deadlock)

```
id_server() →
  receive
    {init, N} → id_server_loop(N)
  end.

id_server_loop(N) →
  receive
    {get, Client} →
      Client ! {id, N},
      id_server_loop(N + 1);
    {init, _} → error
  end.
```

```
client() →

  % Create server process.
  Server = spawn {id_server, []},

  % Initialize server.
  Server ! {init, 5},

  Server ! {get, self},
  receive
    {id, Id} → print Id
  end.
```

```
  receive
    {id, Id} → print Id
  end,
  Server ! {get, self}.
```

✗

Common protocol errors (unsupported request)

```
id_server() →
  receive
    {init, N} → id_server_loop(N)
  end.

id_server_loop(N) →
  receive
    {get, Client} →
      Client ! {id, N},
      id_server_loop(N + 1);
    {init, _} → error
  end.
```

```
client() →

  % Create server process.
  Server = spawn {id_server, []},

  % Initialize server.
  Server ! {init, 5},

  Server ! {get, self},  ✗
  receive
    {id, Id} → print Id
  end.
```

Our wish-list to catch protocol errors

Static

Early error detection

Avoids defensive code

Annotated code

Self-contained information

Documents code

Compatible with other tools

Lightweight

Fast execution

Usable during development

Scalable

Applicable to large code bases

Current error-detection tool landscape

Lightweight ↔ Full-blown

Dialyzer (static typing)	Mailbox typing (behavioural typing)	Concuerror (systematic testing)
✓ Code annotations	✓ Code annotations	✗ Relies on test suites
✓ Scalable	✓ Scalable	✗ Less scalable
✗ Not for concurrency	✓ Targets concurrency	✓ Targets concurrency
✓ Detects errors early	✓ Detects errors early	✗ Detects errors late
✗ Less precise	✗ Less precise	✓ More precise

Mailbox types for unordered interactions

Behavioural typing capturing **process interaction** (De'Liguoro & Padovani '18)

Mailboxes: **first-class** entities with a **type**

Type = Capability + pattern

!*P* Messages that **must** be sent

?*P* Messages that mailbox **can** contain

Many writer, one reader

! reference is **sharable**

? reference is **not sharable**

Pattern = commutative regular expression

Invariant on the mailbox contents

Captures **out-of-order** message deposits

Captures **selective** message reception

Receive one `init` and zero or more `get` messages ⇒ `?"init.get*"`

Send one `id` message ⇒ `!"id"`

Receive zero or more `get` and one `init` message ⇒ `?"init.get*"`

Aim: sends and receives must **balance out**

Challenge 1: Instantiating mailbox types to a PL

Process calculus

Shows a snapshot in a **system state**

Names declared **statically** upfront

Names remain **constant**

Programming language

Specifies what is to be **executed**

Names introduced via **reduction**

Names can be **aliased**

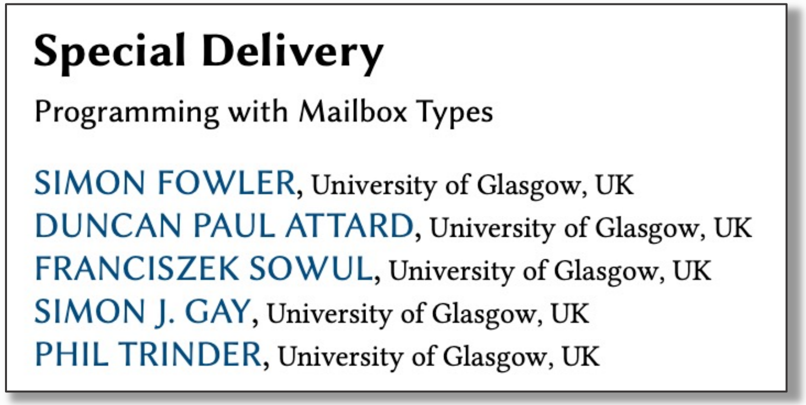
Difficulties

Sequenced expressions, nested evaluation contexts

Using names **many times** to **send**, but **once** to **receive**

Solution 1: Programming with mailbox types

ICFP'23



Special Delivery

Programming with Mailbox Types

SIMON FOWLER, University of Glasgow, UK
DUNCAN PAUL ATTARD, University of Glasgow, UK
FRANCISZEK SOWUL, University of Glasgow, UK
SIMON J. GAY, University of Glasgow, UK
PHIL TRINDER, University of Glasgow, UK

Mailbox types for a core PL calculus: **Pat**

Declarative type system

Corresponding **algorithmic** type system

OCaml type checker for **Pat**

Sufficiently **expressive**

Fast execution

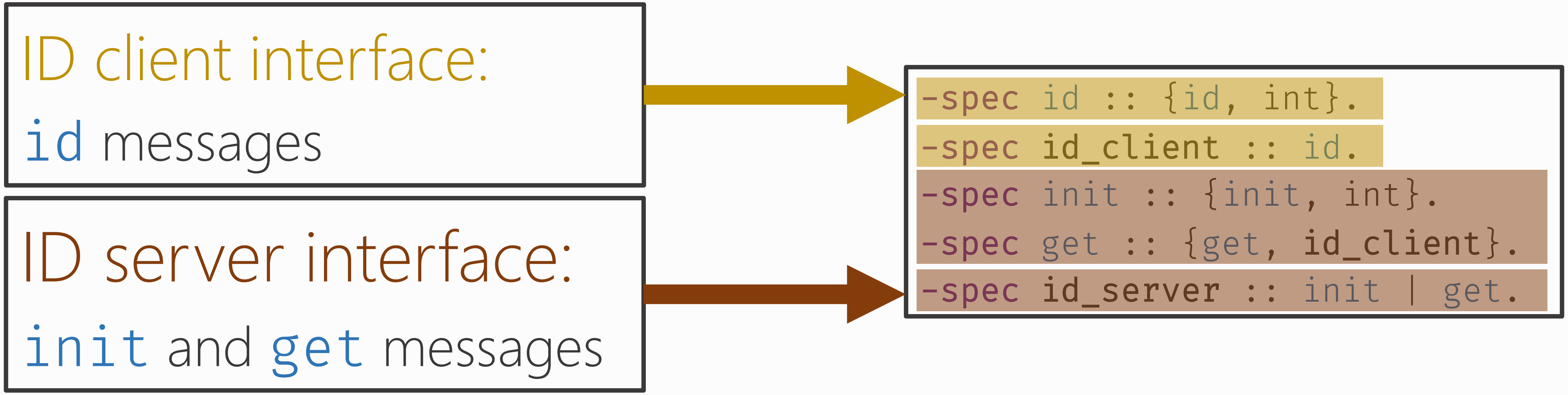
Challenge 2: Applying mailbox typing to Erlang

First-class mailboxes	Erlang mailboxes
Explicitly created and freed	**Tied** to the lifecycle of processes
Process can own **many mailboxes**	Processes own **one mailbox**
Mailbox **needed** for receiving	Mailbox **implicit** when receiving
Mailbox has a **precise type**	Mailbox is **untyped**

Interface = isolates mailbox type + state

A **set of messages** that a mailbox can **receive**

Annotates process functions: `-new` or `-use`



ID client interface:

`id` messages

ID server interface:

`init` and `get` messages

```
-spec id :: {id, int}.
-spec id_client :: id.
-spec init :: {init, int}.
-spec get :: {get, id_client}.
-spec id_server :: init | get.
```

```
-new id_server.
-spec id_server() ⟶ unit.
id_server() ⟶
  assert("init.get*"),
  receive
    {init, N} ⟶ id_server_loop(N)
  end.

-use id_server.
-spec id_server_loop(int) ⟶ unit.
id_server_loop(N) ⟶
  assert("get*"),
  receive
    {get, Client} ⟶
      Client ! {id, N},
      id_server_loop(N + 1)
  end.
```

Logical `id_server` mailbox

Thread `id_server` mailbox through

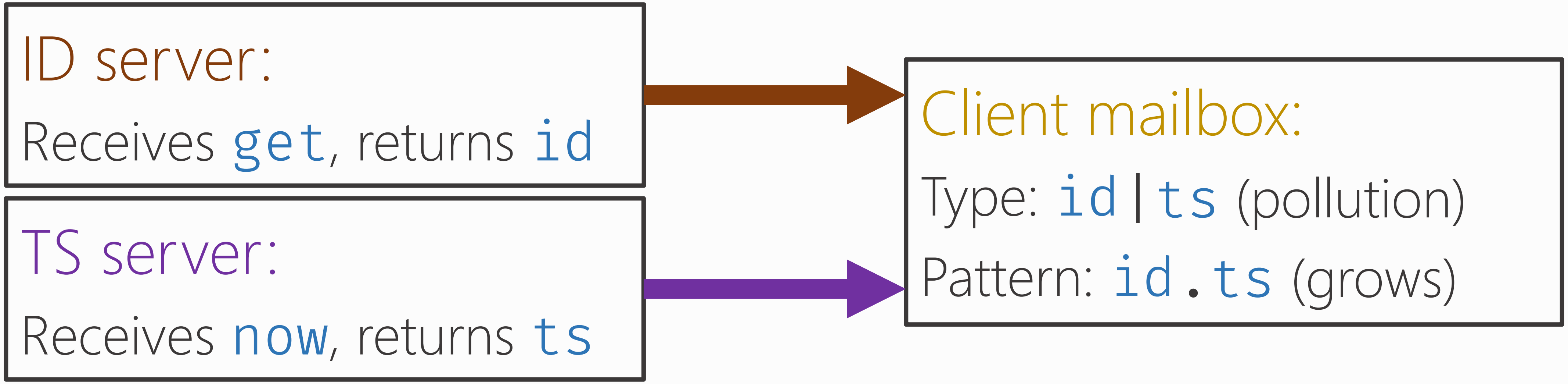
```
-new id_client.
-spec client() ⟶ unit.
client() ⟶
  Server = spawn {id_server, []},

  Server ! {init, 5},
  Server ! {get, self},
  assert("id"),
  receive
    {id, Id} ⟶ print Id
  end.
```

Logical `id_client` mailbox

Remove defensive error-handling code `{init, _} ⟶ error`

Limitations with typing one monolithic mailbox



ID server:

Receives `get`, returns `id`

TS server:

Receives `now`, returns `ts`

Client mailbox:

Type: `id`|`ts` (pollution)

Pattern: `id.ts` (grows)

Does **not delineate** conceptually unrelated messages

Tracking mailbox state can quickly become **intractable**

```
-spec id_rpc(id_server) → int.
id_rpc(To) →
  To ! {get, self},
  assert(“id”),
  assert(“id.ts”),
  receive
    {id, Id} → Id
  end.

-spec ts_asy(ts_server) → unit.
ts_asy(To) → To ! {now, self}.

-spec ts_get() → int.
ts_get() →
  assert(“ts”),
  receive
    {ts, Ts} → Ts
  end.
```

Pattern grows

Reasoning not local

```
-spec id_server :: init | get.
-spec ts_server :: now.
-spec client :: id | ts.
```

Type pollution

Patterns depends on order of function invocations

```
-new client.
-spec client() → unit.
client() →
  IdServer = spawn {id_server, …
  TsServer = spawn {ts_server, …

  ts_asy(TsServer), %Async.
  print(IdServer),
  id_rpc(IdServer), %Sync.
  print(TsServer),
  print ts_get().
```

Mailbox types: induce **structured communication**

Organising the Erlang mailbox logically

Gives a **projected view** of an otherwise monolithic mailbox

Isolates message types

Minimises type pollution

Types are more precise

Isolates mailbox state

Patterns are localised

Reasoning becomes compositional

```
-new id_client.
-spec id_rpc(id_server) → int.
id_rpc(To) →
  To ! {get, self},
  assert("id"),
  receive
    {id, Id} → Id
  end.

-spec ts_asy(ts_server) → unit.
ts_asy(To) → To ! {now, self}.

-spec ts_get() → int.
ts_get() →
  assert("ts"),
  receive
    {ts, Ts} → Ts
  end.
```

Logical id_client mailbox

Local reasoning

```
-spec id_server :: init | get.
-spec ts_server :: now.
-spec id_client :: id.
-spec ts_client :: ts.
```

Two interfaces

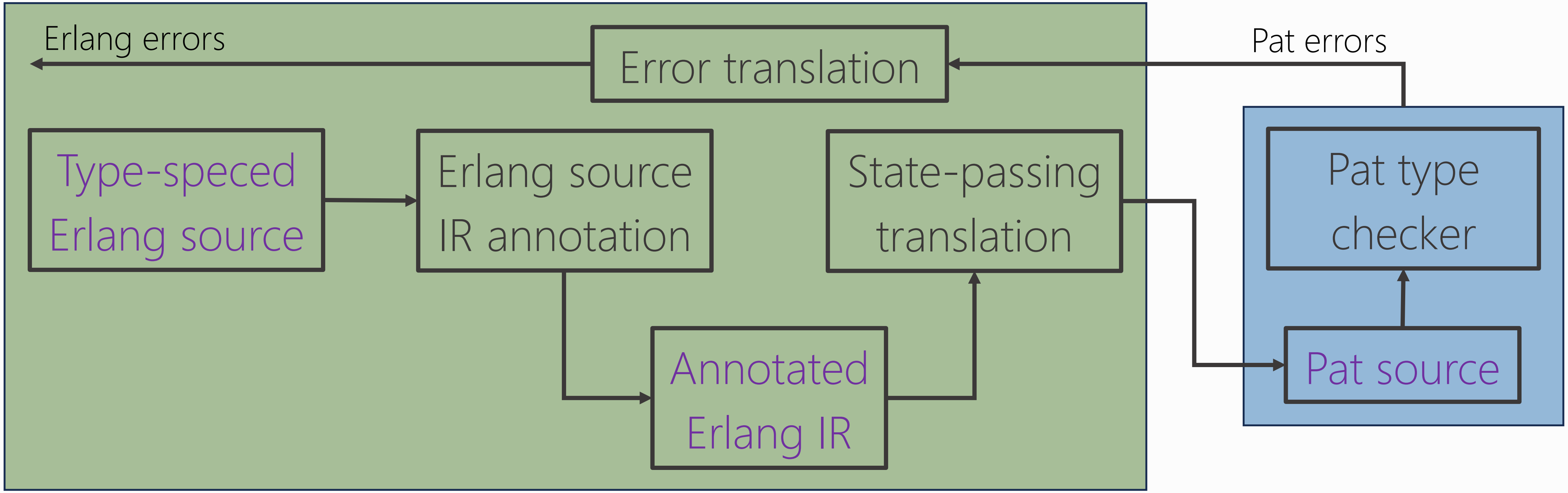
```
-new ts_client.
-spec client() → unit.
client() →
  IdServer = spawn {id_server, …
  TsServer = spawn {ts_server, …

  ts_asy(TsServer), % Async.
  print id_rpc(IdServer), % Sync.
  print ts_get().
```

Logical ts_client mailbox

Isolated mailboxes

Towards mailbox typing for Erlang



Status summary

In progress

Design with Erlang typespecs

Annotated Erlang IR

Refinement of Pat

Formalised **IR** ⟶ **Pat** translation

Next

Implement **IR** ⟶ **Pat** translation

Implement error translation

Formalise **Erlang** ⟶ **IR** translation

Implement **Erlang** ⟶ **IR** translation

Why mailbox typing?

Actors

Type the **mailbox contents**, not the process interactions

Fits **asynchrony**: out-of-order mailbox reading and writing

Fits **asymmetric** interaction: many writers, one reader paradigm

Erlang

Overlay a structure on top of a monolithic mailbox

Document communication between processes