Actors and Channels in Core λ-calculi

Simon Fowler
Joint work with Sam Lindley and Philip Wadler

ABCD Meeting, January 2016

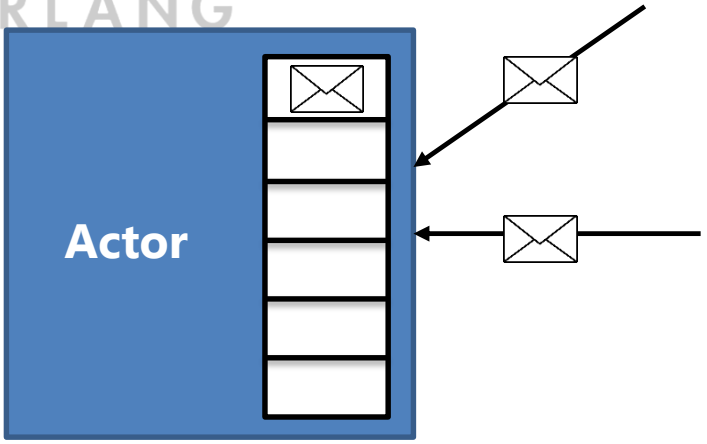
EPSRC Centre for Doctoral Training in Pervasive Parallelism
THE UNIVERSITY of EDINBURGH informatics

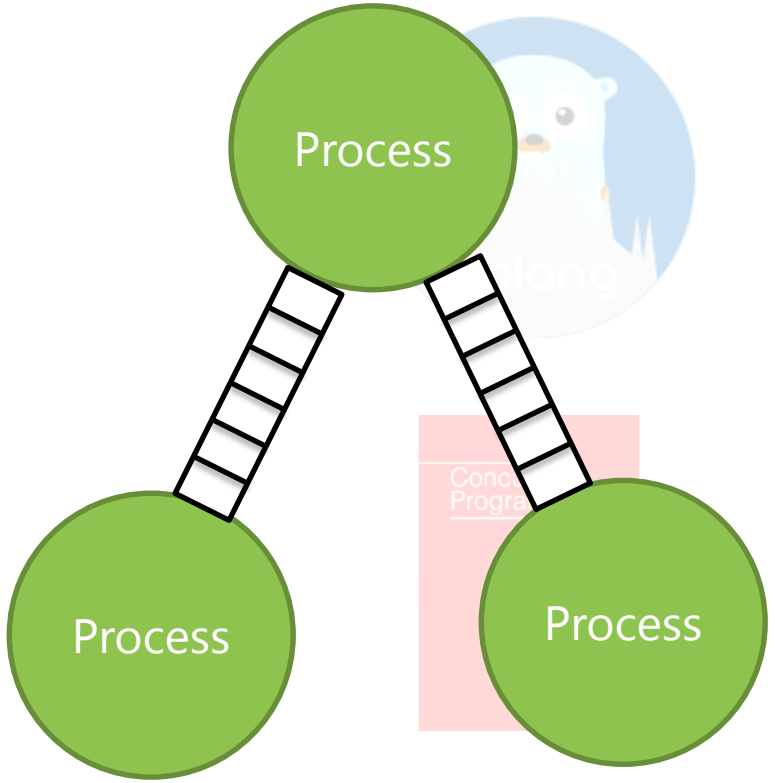
Actors and Channels

elixir

Hopac

Actors and Channels





Actors and Channels

elixir

?

Hopac

Why?

- Concise formalisations of transformations between actors and channels
- Explore design issues with (session) type-parameterised actor calculi
- Influence design of session-typed actors for Links

$\boldsymbol{\lambda_{\mathbf{ch}}}$: A channel-based $\boldsymbol{\lambda}$-calculus

$$
\begin{array}{lll}
\text{Types} & A, B ::= & \mathbf{0} \mid \mathbf{1} \mid A \times B \mid A + B \mid A \to B \mid \mathsf{List}(A) \mid \mathsf{Chan}(A) \\
\text{Terms} & L, M, N ::= & x \mid \lambda x.M \mid M\ N \\
& \mid & (M, N) \mid \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \\
& \mid & \mathsf{rec}\ f(x : A) : B\ .\ M \\
& \mid & \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \{\mathsf{inl}\ x \mapsto N; \mathsf{inr}\ x \mapsto N\} \\
& \mid & [\,]_A \mid [M] \mid M +\!\!+ N \mid \mathsf{case}\ L\ \{[\,]_A \mapsto N; [x] +\!\!+ y \mapsto N'\} \\
& \mid & () \mid M; N \mid \mathsf{fork}\ M \mid \mathsf{give}\ M\ N \mid \mathsf{take}\ M \mid \mathsf{newCh}_A
\end{array}
$$

$\lambda_{\mathbf{ch}}$: Channel Typing Rules

$$\textsc{Give}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \mathsf{Chan}(A)}{\Gamma \vdash \mathsf{give}\ M\ N : \mathbf{1}} \qquad \textsc{Take}\ \frac{\Gamma \vdash M : \mathsf{Chan}(A)}{\Gamma \vdash \mathsf{take}\ M : A}$$

$$\textsc{NewCh}\ \frac{}{\Gamma \vdash \mathsf{newCh}_A : \mathsf{Chan}(A)} \qquad \textsc{Fork}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathsf{fork}\ M : \mathbf{1}}$$

λ_{ch}: Communication Semantics

Configurations $\mathcal{C}, \mathcal{D} ::= \mathcal{C} \parallel \mathcal{C}' \mid (\nu x)\mathcal{C} \mid M \mid x(\overrightarrow{V})$

GIVE

$$\frac{}{E[\mathsf{give}\ W\ x] \parallel x(\overrightarrow{V}) \longrightarrow E[()] \parallel x(\overrightarrow{V}\ W)}$$

TAKE

$$\frac{}{E[\mathsf{take}\ x] \parallel x(W\ \overrightarrow{V}) \longrightarrow E[W] \parallel x(\overrightarrow{V})}$$

NEWCH

$$\frac{x \text{ is a fresh name}}{E[\mathsf{newCh}_A] \longrightarrow (\nu x)(E[x] \parallel x(\epsilon))}$$

FORK

$$\frac{}{G[\mathsf{fork}\ M] \longrightarrow G[()] \parallel M}$$

Actor-based Functional Languages

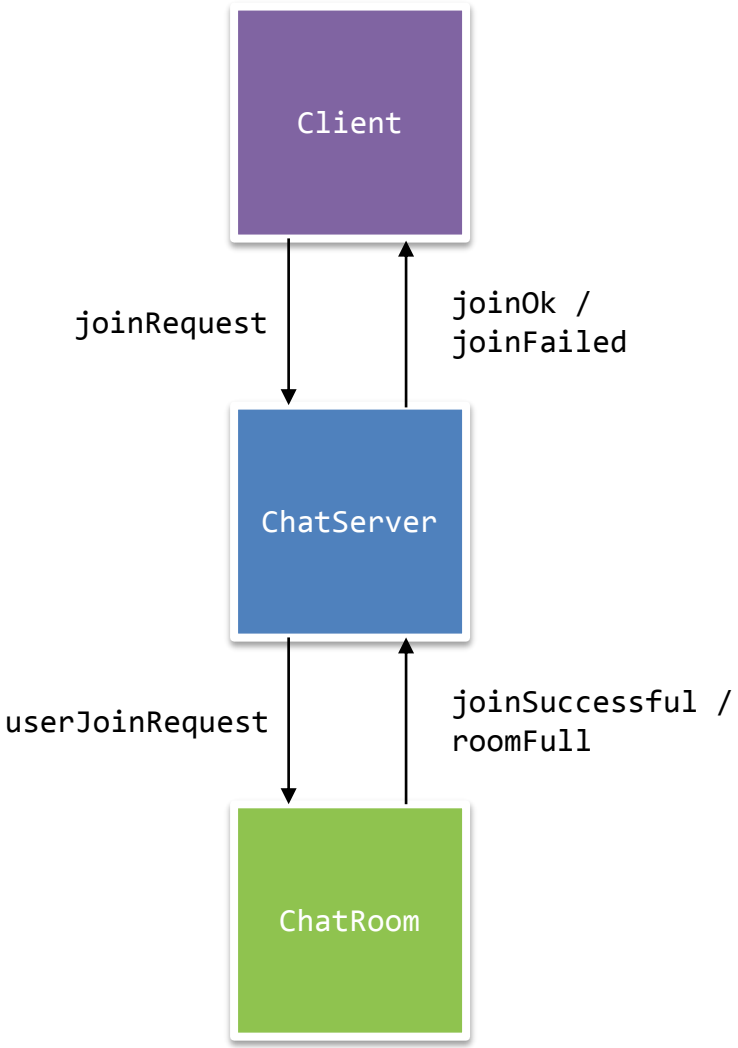
- Based on the actor model (Agha, 1985), **not the same**
  - Actors represented as **lightweight processes**, scheduled by RTS
  - No explicit notion of behaviour / become
  - Computation modelled directly

- Primitives
  - `spawn`
  - `send`
  - `receive`
  - *`(wait)`*

```
-module(stack).
-compile(export_all).
loop(State) ->
  receive
    {pop, Pid} ->
      [Hd|Tl] = State,
      Pid ! Hd,
      loop(Tl);
    {push, Item} ->
      loop([Item] ++ State)
  end.

spawn_stack() ->
  spawn(stack, loop, [[]]).
```

The Type Pollution Problem



ChatServer has type

Pid(joinRequest + joinSuccessful + roomFull)

...wherever we use it!

$\lambda_{\mathbf{act}}$: A core actor-based λ-calculus

$$
\begin{array}{lll}
\text{Types} & A, B, C ::= & \mathbf{0} \mid \mathbf{1} \mid A \times B \mid A + B \mid A \to^{C} B \mid \mathsf{List}(A) \mid \mathsf{Pid}(A, B) \\
\text{Terms} & L, M, N ::= & x \mid \lambda x.M \mid M\ N \\
& & \mid\ (M, N) \mid \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \\
& & \mid\ \mathsf{rec}\ f(x : A) : B\ .\ M \\
& & \mid\ \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{case}\ M\ \{\mathsf{inl}\ x \mapsto N; \mathsf{inr}\ x \mapsto N\} \\
& & \mid\ [\,]_A \mid [M] \mid M +\!\!+ N \mid \mathsf{case}\ L\ \{[\,]_A \mapsto N; [x] +\!\!+ y \mapsto N'\} \\
& & \mid\ \mathsf{spawn}\ M \mid \mathsf{send}\ M\ N \mid \mathsf{receive} \mid \mathsf{wait}\ M
\end{array}
$$

$\lambda_{\mathbf{act}}$: Selected Typing Rules

$$\boxed{\Gamma; C \vdash M : A}$$

$$\textsc{Send}\ \frac{\Gamma; C \vdash M : A \quad \Gamma; C \vdash N : \mathsf{Pid}(A, B)}{\Gamma; C \vdash \mathsf{send}\ M\ N : \mathbf{1}}$$

$$\textsc{Recv}\ \frac{}{\Gamma; C \vdash \mathsf{receive} : C}$$

$$\textsc{Wait}\ \frac{\Gamma; C \vdash M : \mathsf{Pid}(A, B)}{\Gamma; C \vdash \mathsf{wait}\ M : B}$$

$$\textsc{Spawn}\ \frac{\Gamma; A \vdash M : \mathsf{Pid}(A, B) \to^{A} B}{\Gamma; C \vdash \mathsf{spawn}\ M : \mathsf{Pid}(A, B)}$$

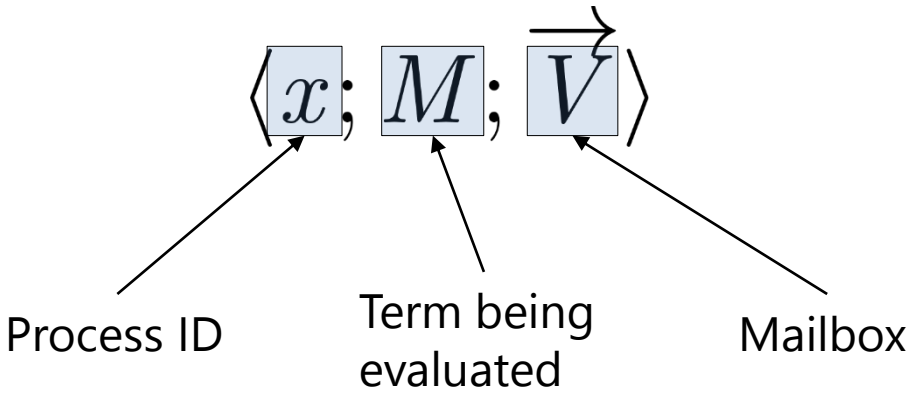
(An alternative formulation of wait)

spawnWait construct: combination of spawn and wait

$$\textsc{SpawnWait}\ \frac{\Gamma; A \vdash M : \mathsf{Pid}(A) \to^{A} B}{\Gamma; C \vdash \mathsf{spawnWait}\ M : B}$$

$\lambda_{\mathbf{act}}$: Configurations

Actors are a three-tuple:



$$\text{Configurations} \quad \mathcal{C}, \mathcal{D} ::= \mathcal{C} \parallel \mathcal{C}' \mid (\nu x)\mathcal{C} \mid \langle x; M; \overrightarrow{V} \rangle$$

$\lambda_{\mathbf{act}}$: Communication Semantics

SPAWN

$$\frac{y \text{ is a fresh name}}{\langle x; E[\mathsf{spawn}\ M]; \overrightarrow{V}\rangle \longrightarrow (\nu y)(\langle x; E[y]; \overrightarrow{V}\rangle \parallel \langle y; M\ y; \epsilon\rangle)}$$

SEND

$$\frac{}{\langle x; E[\mathsf{send}\ V'\ y]; \overrightarrow{V}\rangle \parallel \langle y; E'[M]; \overrightarrow{W}\rangle \longrightarrow \langle x; E[()]; \overrightarrow{V}\rangle \parallel \langle y; E'[M]; \overrightarrow{W}\ V'\rangle}$$

RECEIVE

$$\frac{}{\langle x; E[\mathsf{receive}]; W\ \overrightarrow{V}\rangle \longrightarrow \langle x; E[W]; \overrightarrow{V}\rangle}$$

WAIT

$$\frac{}{\langle x; E[\mathsf{wait}\ y]; \overrightarrow{W}\rangle \parallel \langle y; V; \overrightarrow{W'}\rangle \longrightarrow \langle x; E[V]; \overrightarrow{W}\rangle}$$

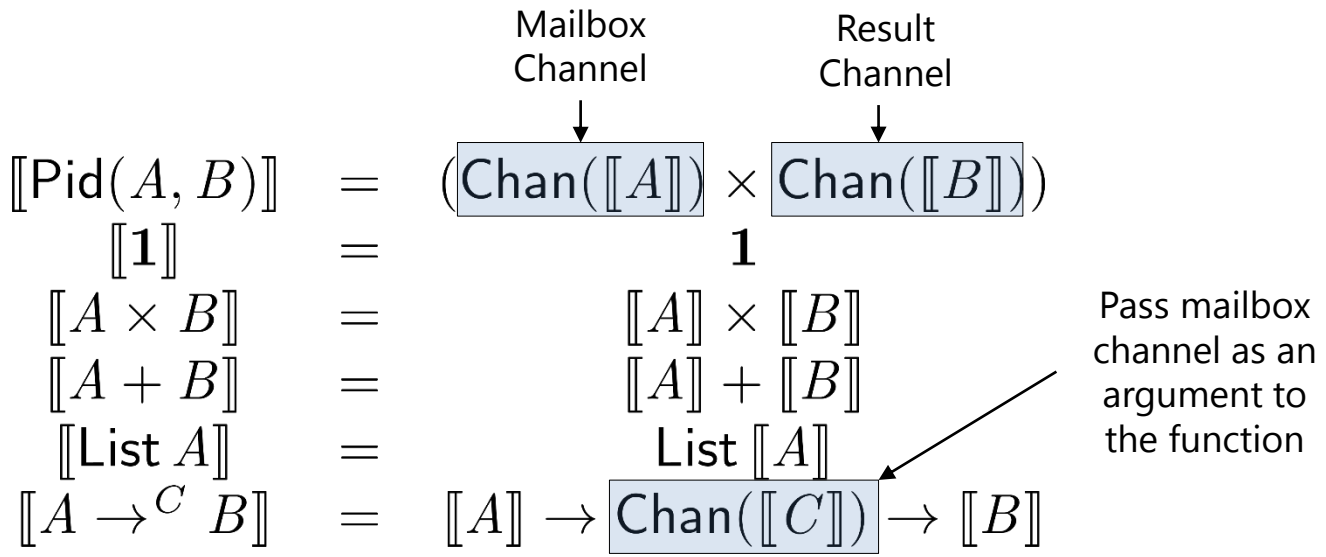
Translations: Desired Properties

- Translation function $[\![-]\!]$: Actors in terms of channels
- Type Preservation:
  - Terms:
    If $\Gamma; C \vdash_{\lambda_{\mathrm{act}}} M : A$, then $[\![\Gamma]\!], c : \mathsf{Chan}([\![C]\!]) \vdash_{\lambda_{\mathrm{ch}}} [\![M]\!]\, c : [\![A]\!]$
  - Configurations
    If $\Gamma \vdash_{\lambda_{\mathrm{act}}} \mathcal{C}$, then $[\![\Gamma]\!] \vdash_{\lambda_{\mathrm{ch}}} [\![\mathcal{C}]\!]$.

- Semantics Preservation
  - Terms
    If $M \longrightarrow_{\lambda_{\mathrm{act}}} M'$, then let $c = \mathsf{newCh}_{[\![C]\!]}$ in $[\![M]\!]\, c \longrightarrow^*_{\lambda_{\mathrm{ch}}} [\![M']\!]\, c$ for some type C.
  - Configurations
    If $\mathcal{C} \longrightarrow_{\lambda_{\mathrm{act}}} \mathcal{C}'$, then $[\![\mathcal{C}]\!] \longrightarrow^*_{\lambda_{\mathrm{ch}}} [\![\mathcal{C}']\!]$

Actors implemented using channels

Idea (based on Cooper et al., 2006): create a channel and use it as a mailbox, "threaded through" the translation



λ_{act} in λ_{ch}: Translation on Terms

Top-level term: create mailbox channel, pass as parameter to translation

$$[\![M]\!] \quad = \quad \text{let } chMb = \text{newCh}_A \text{ in } [\![M]\!]\, chMb$$

Extra parameter c' to pass mailbox channel to the function

$$[\![\lambda x.M]\!]\, c \quad = \quad \lambda x.\lambda c'.[\![M]\!]\, c'$$

$$[\![M\ N]\!]\, c \quad = \quad ([\![M]\!]\, c)\,([\![N]\!]\, c)\, c$$

c applied to translated function M

$\lambda_{\mathbf{act}}$ in $\lambda_{\mathbf{ch}}$: Translation on Terms

Create a mailbox and result channel; fork a new process; evaluate result; and give to result channel

$$[\![\text{spawn } M]\!]\ c = \text{let } chMb = \text{newCh}_A \text{ in let } chRes = \text{newCh}_B \text{ in fork}(\text{let } res = [\![M]\!]\ chMb \text{ in give } res\ chRes);\ (chMb, chRes)$$

Translate PID; `give` to mailbox channel

$$[\![\text{send } M\ N]\!]\ c = \text{let } (chMb, \_) = [\![N]\!]\ c \text{ in give } ([\![M]\!]\ c)\ chMb$$

Translate PID; `take` from result channel

$$[\![\text{receive}]\!]\ c = \text{take } c$$

$$[\![\text{wait } M]\!]\ c = \text{let } (\_, chRes) = [\![M]\!]\ c \text{ in take } chRes$$

Translations: Desired Properties

- Translation function $[\![-]\!]$: Channels in terms of actors
- Type Preservation:
  - Terms:
    $\text{If } \Gamma \vdash_{\lambda_{\text{ch}}} M : A, \text{ then } (\!|\Gamma|\!); C \vdash_{\lambda_{\text{act}}} (\!|M|\!) : (\!|A|\!) \text{ for some type } C.$
  - Configurations
    $\text{If } \Gamma \vdash_{\lambda_{\text{ch}}} \mathcal{C}, \text{ then } (\!|\Gamma|\!) \vdash_{\lambda_{\text{act}}} (\!|\mathcal{C}|\!).$

- Semantics Preservation
  - Terms
    $\text{If } M \longrightarrow_{\lambda_{\text{ch}}} M', \text{ then } (\!|M|\!) \longrightarrow^*_{\lambda_{\text{act}}} (\!|M|\!).$
  - Configurations
    $\text{If } \mathcal{C} \longrightarrow_{\lambda_{\text{ch}}} \mathcal{C}', \text{ then } (\!|\mathcal{C}|\!) \longrightarrow^*_{\lambda_{\text{act}}} (\!|\mathcal{C}'|\!).$

Channels implemented using Actors

- Idea: represent **channels** as **processes**; emulate `give` and `take` using internal state

Channels: pair of `take` and `give` functions

$$
\begin{aligned}
(\!| \mathsf{Chan}(A) |\!) &= ((\mathbf{1} \to^{(\!| A |\!)} (\!| A |\!)) \times ((\!| A |\!) \to^{(\!| A |\!)} \mathbf{1})) \\
(\!| \mathbf{1} |\!) &= \mathbf{1} \\
(\!| A \times B |\!) &= (\!| A |\!) \times (\!| A |\!) \\
(\!| A + B |\!) &= (\!| A |\!) + (\!| B |\!) \\
(\!| A \to B |\!) &= (\!| A |\!) \to^{C} (\!| B |\!) \\
(\!| \mathsf{List}\ A |\!) &= \mathsf{List}\ (\!| A |\!)
\end{aligned}
$$

C can be **any type**, as **translated** functions don't have the ability to receive from mailboxes

Translations on Terms (the easy ones)

$$
\begin{aligned}
(\!|\, \mathsf{fork}\ M \,|\!) &= \mathsf{let}\ \_ = \mathsf{spawn}\ (\!|\, M \,|\!)\ \mathsf{in}\ () \\
(\!|\, \mathsf{give}\ M\ N \,|\!) &= \mathsf{let}\ (\_, \texttt{giveFn}) = (\!|\, N \,|\!)\ \mathsf{in} \\
&\quad\ \texttt{giveFn}\ (\!|\, M \,|\!) \\
(\!|\, \mathsf{take}\ M \,|\!) &= \mathsf{let}\ (\texttt{takeFn}, \_) = (\!|\, M \,|\!)\ \mathsf{in} \\
&\quad\ \texttt{takeFn}\ () \\
(\!|\, \mathsf{newCh}_D \,|\!) &= \ldots
\end{aligned}
$$

Translation of **newCh**…

$(\!|\, \mathsf{newCh}_A \,|\!) =$

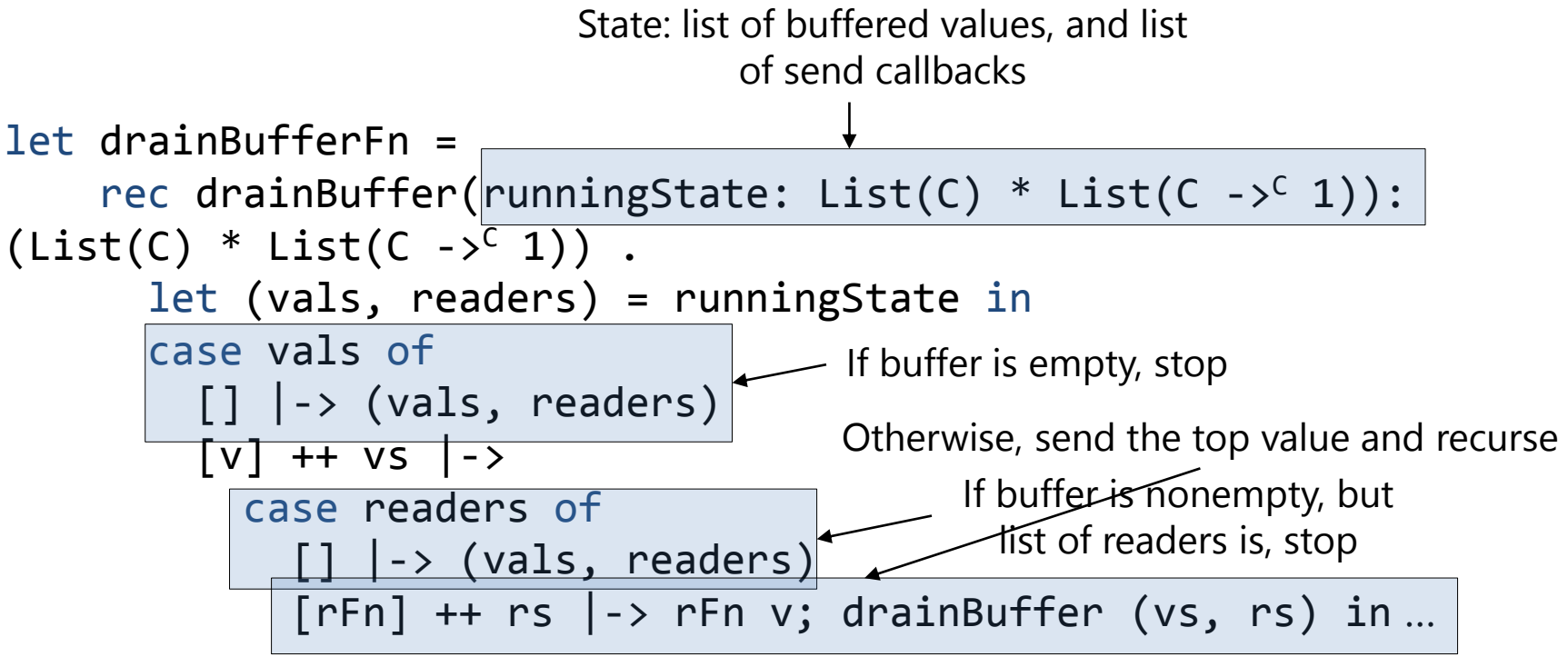
```
let drainBufferFn =
    rec drainBuffer(runningState: List(C) * List(C ->C 1)): (List(C), * List(C ->C 1)) .
      let (vals, readers) = runningState in
      case vals of
        [] |-> (vals, readers)
        [v] ++ vs |->
          case readers of
            [] |-> (vals, readers)
            [rFn] ++ rs |-> rFn v; drainBuffer (vs, rs) in

let stepFn = rec step(state: (List(C) * List(C ->C 1)): 1.
  let (vals, readers) = drainBufferFn state in
  let msg = receive in
  case msg of
      inl v |-> step (vals ++ [v], readers)
      inr sendFn |-> step (vals, readers ++ [sendFn]) in

let chanPid = spawn(stepFn([], [])) in
let giveFn = λx. send (inl x) chanPid in
let takeFn = λx.
  let newPid =
    spawn (λnewPid -> send chanPid (inr (λval -> send val newPid)); receive) in
  wait newPid in
(takeFn, giveFn)
```

Seriously though:

1. Define a function to ensure that either the buffer or list of blocked readers is empty at each step.



Seriously though:

2. Define a loop function `stepFn` which calls `drainBufferFn`, then receives a message and modifies the state

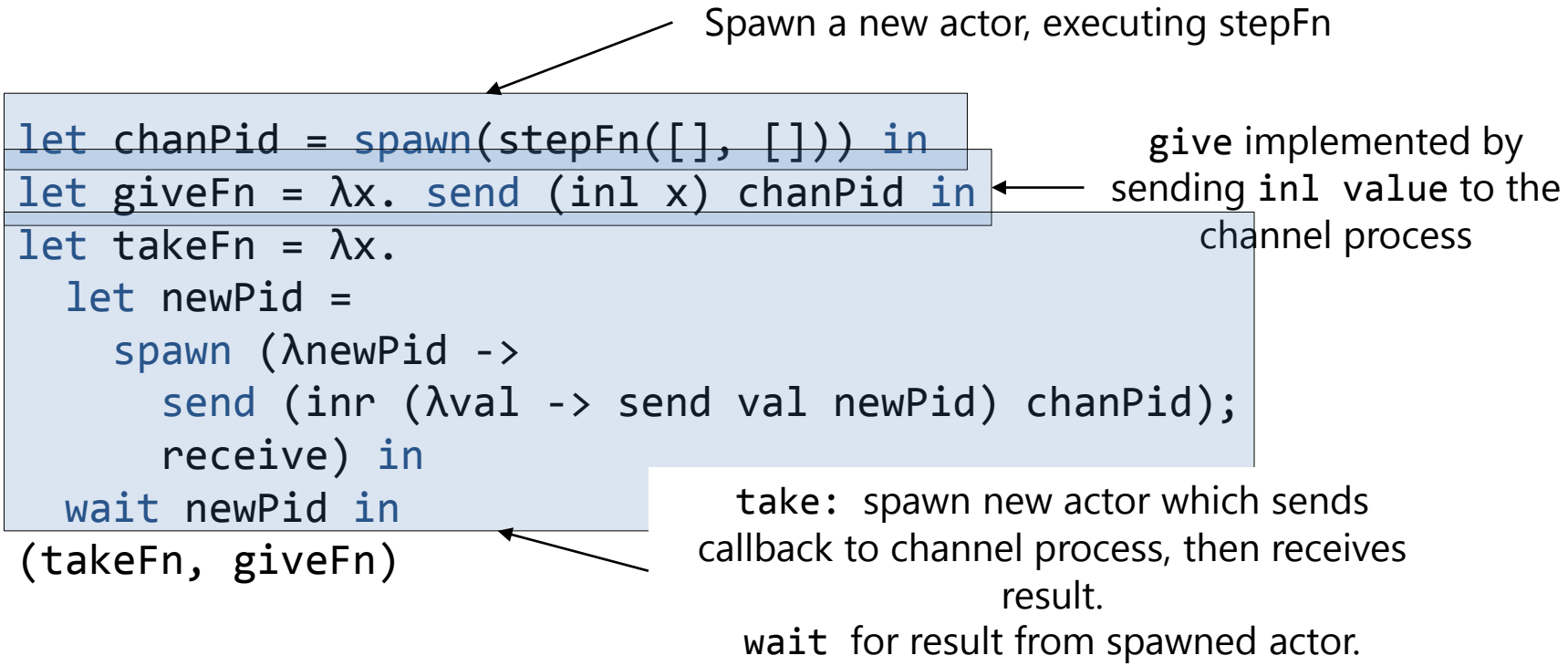
```
let stepFn = rec step(state: (List(C) * List(C ->C 1)): 1.
  let (vals, readers) = drainBu
  let msg = receive in
  case msg of
      inl v |-> step (vals ++ [v], readers)
      inr sendFn |-> step (vals, readers ++ [sendFn]) in ...
```

Message can either be inl value or

If a callback, add to callback list

Seriously though:

3. Spawn a new actor with empty state; define functions for `give` and `take`



Still to do / the future

- Goal: A minimal behaviourally typed actor calculus
  - Conjecture: there exists a minimal session-typed actor calculus
    - …which we can do analogous translations to / from asynchronous GV
    - …not needing recursion for the channel -> actor translation
    - …with simpler session types (no ! / ? required?)
  - Influence a design for session-typed actors in Links
- A better solution to type pollution
  - Subtyping?
- Lots of proving to do!

Extra slides

Actor Configuration Typing

Mailbox typing

$$\textsc{Empty}\ \frac{}{\Gamma \vdash \epsilon : \overrightarrow{A}} \qquad \textsc{Mailbox}\ \frac{\Gamma; B \vdash W : A \quad \Gamma; B \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma; B \vdash W\,\overrightarrow{V} : \overrightarrow{A}}$$

Configuration typing

$$\textsc{Par}\ \frac{\Gamma \vdash \mathcal{C} \quad \Gamma \vdash \mathcal{C}'}{\Gamma \vdash \mathcal{C} \parallel \mathcal{C}'} \qquad \textsc{Pid}\ \frac{\Gamma, x : \mathsf{Pid}(A, B) \vdash \mathcal{C}}{\Gamma \vdash (\nu x)\mathcal{C}}$$

$$\textsc{Actor}\ \frac{\Gamma; D \vdash x : \mathsf{Pid}(A, B) \quad \Gamma; A \vdash M : B \quad \Gamma; A \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma \vdash \langle x; M; \overrightarrow{V} \rangle}$$

Figure 1: Configuration typing

Session Actor Calculus Sketch

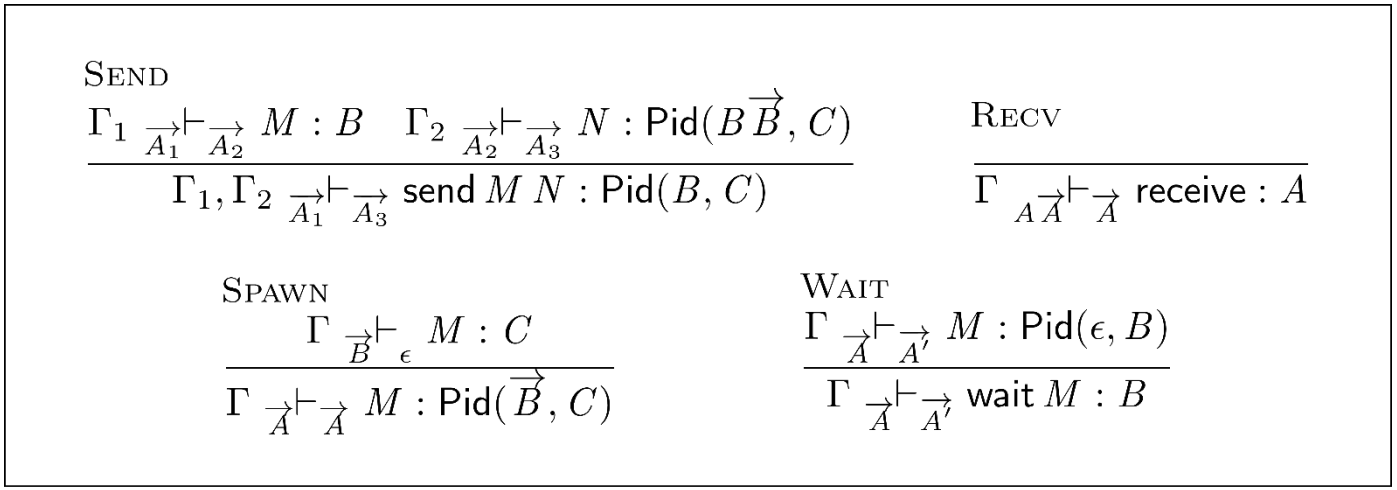
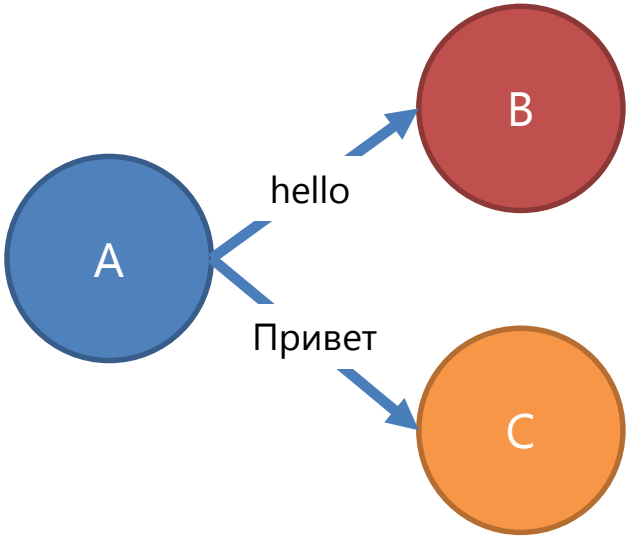


Figure 1: Possible typing rules for session actor calculus communication primitives

Actor-based Functional Languages vs. the Actor Model

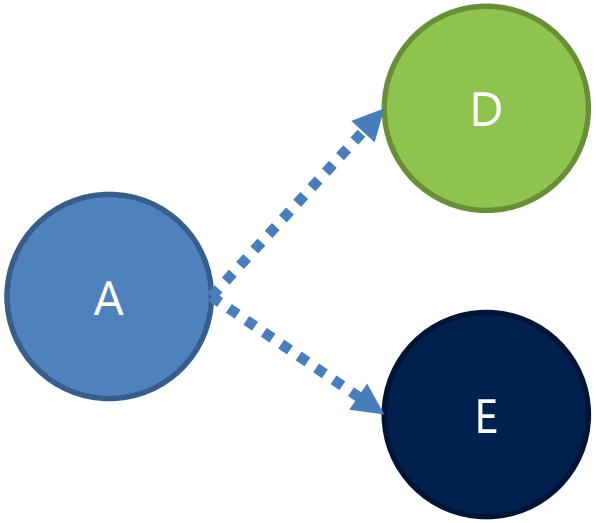
- A word of caution regarding terminology!
- Actors: a minimal concurrency model
  - Unforgeable PID, message queue (mailbox)
  - Behaviour



1. Send a finite set of messages to another actor

Actor-based Functional Languages vs. the Actor Model

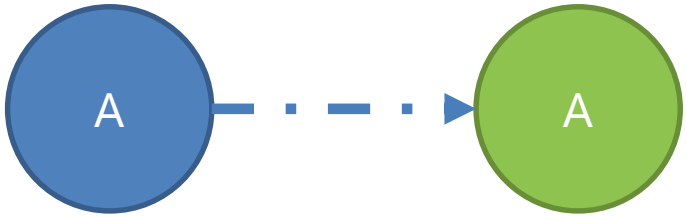
- A word of caution regarding terminology!
- Actors: a minimal concurrency model
  - Unforgeable PID, message queue (mailbox)
  - Behaviour



2. Spawn a finite set of new actors

Actor-based Functional Languages vs. the Actor Model

- A word of caution regarding terminology!
- Actors: a minimal concurrency model
  - Unforgeable PID, message queue (mailbox)
  - Behaviour



3. Change behaviour: react differently when processing next message

Actor-based Functional Languages vs. the Actor Model

- Agha (1985) introduces minimal actor languages SAL and Act, which stay very true to the core actor model: