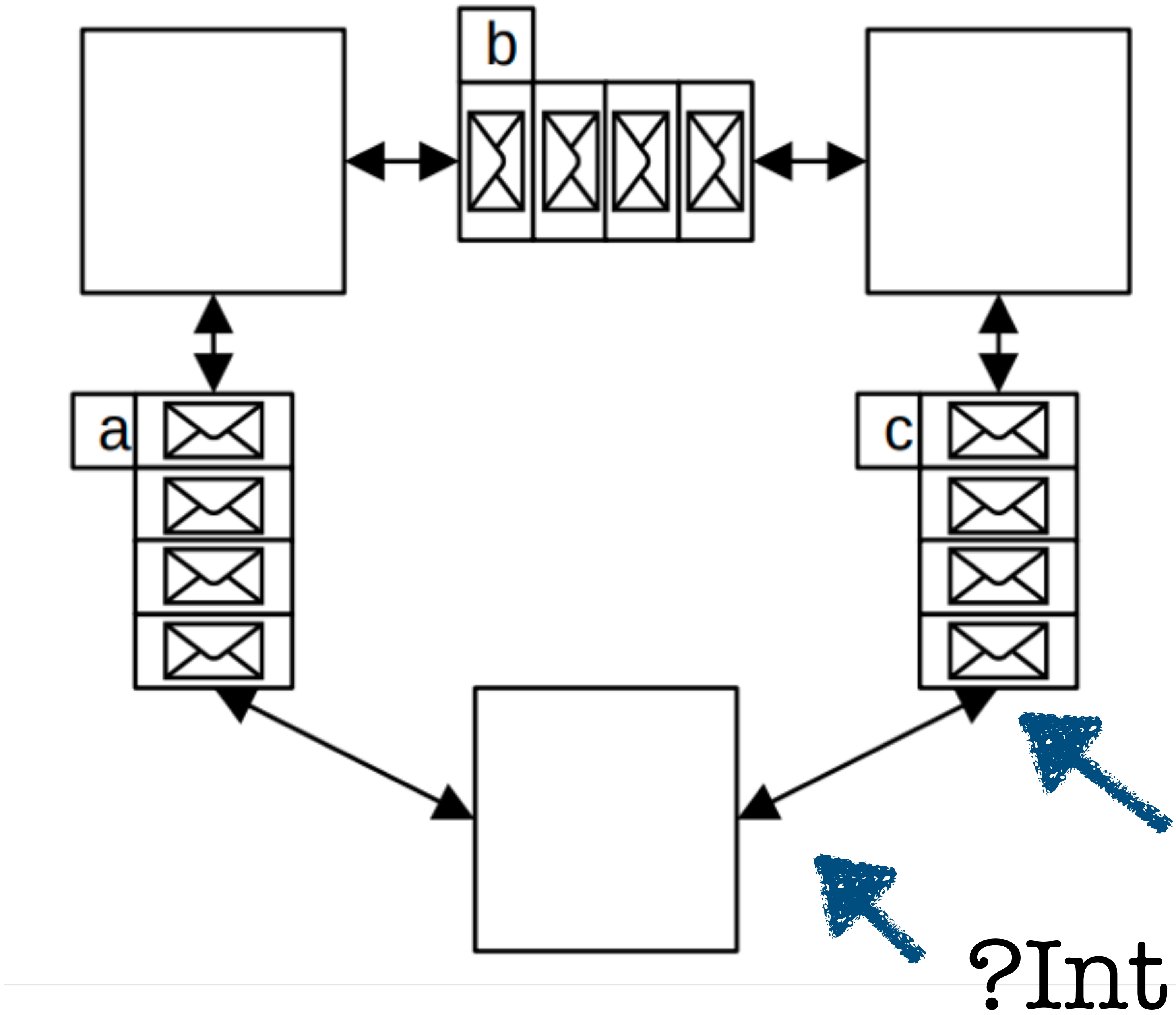
Special Delivery: Programming with Mailbox Types

Simon Fowler, Simon J. Gay, Phil Trinder, and Franciszek Sowul

STARDUST Meeting, University of Kent

12th September 2022

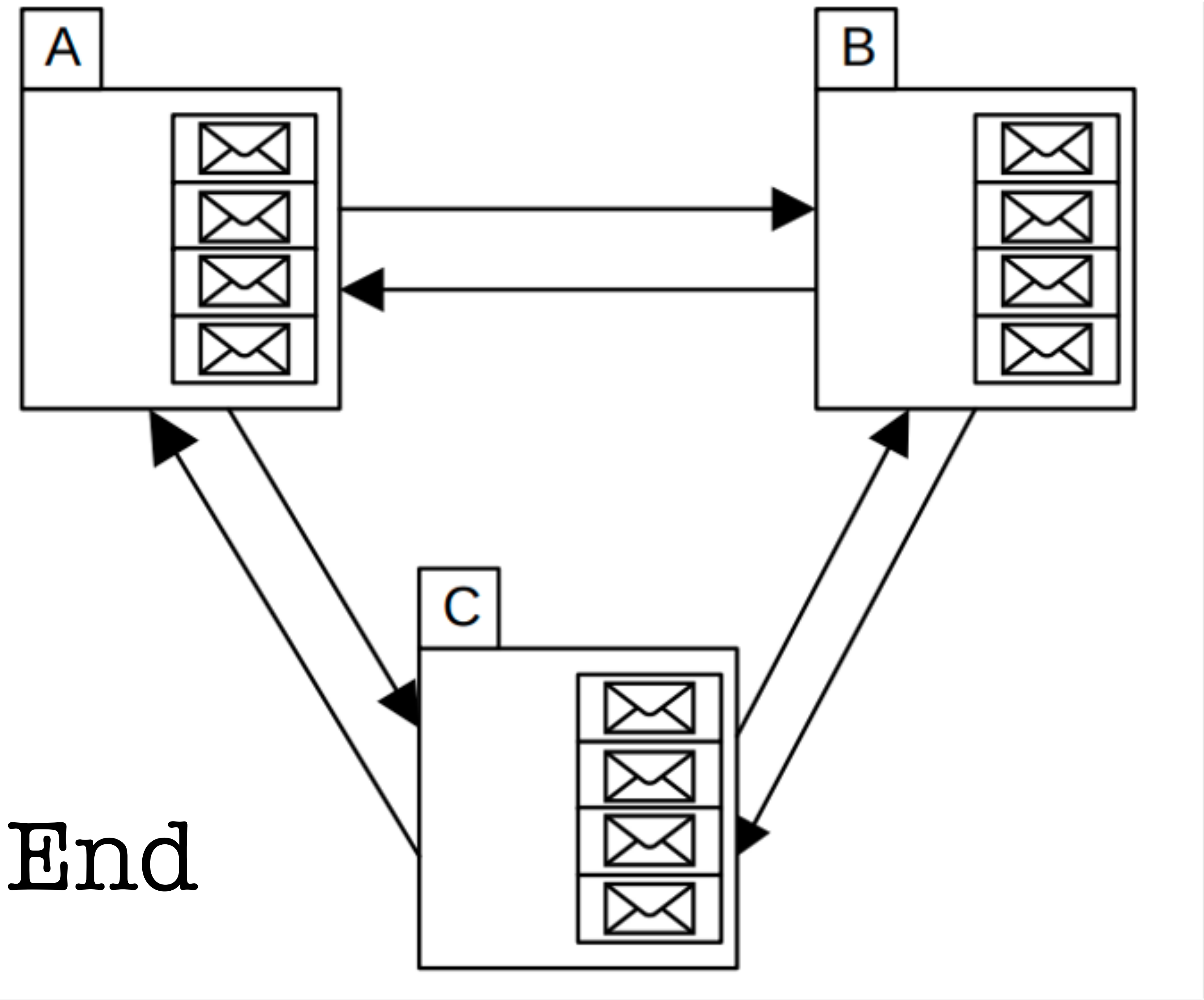


!Int.!Int.?Bool.End

?Int.?Int.!Bool.End

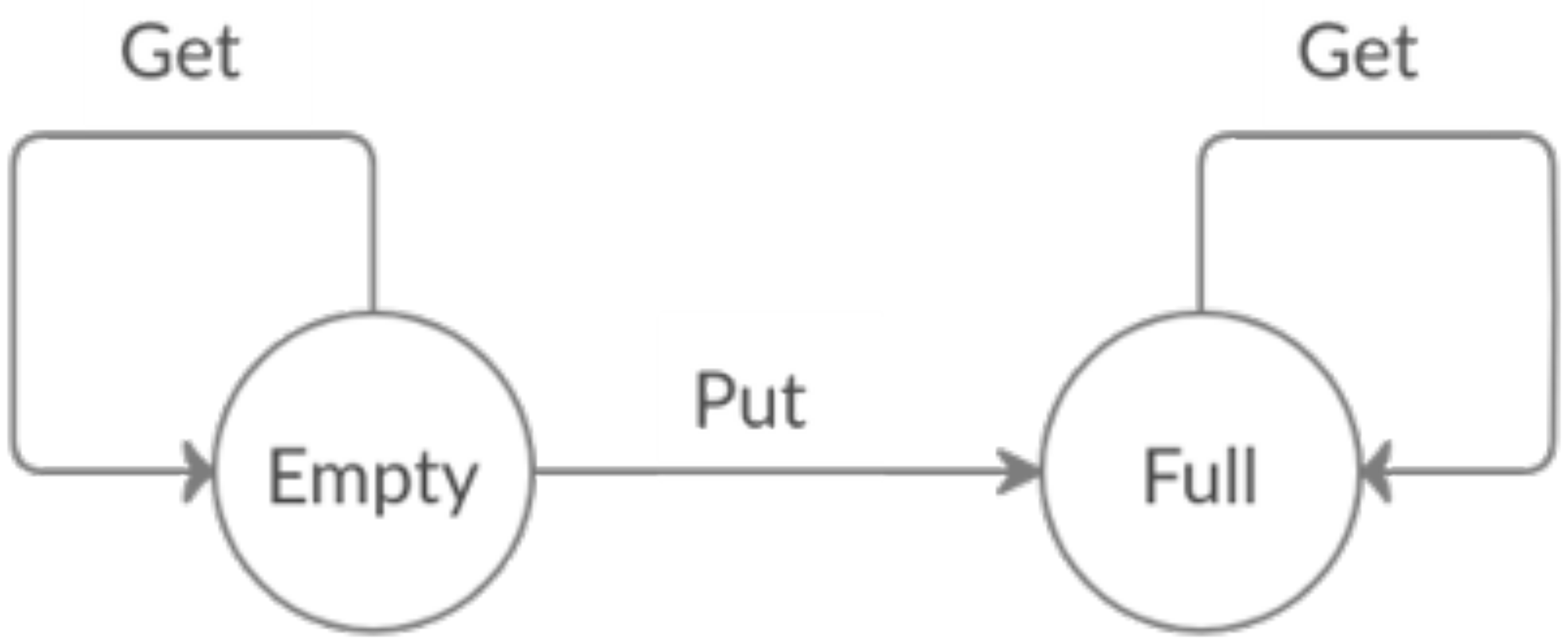
Channels

- Communication is **ordered** and **bidirectional**
- **Anonymous** processes, **multiple, named** channel endpoints
- Easy to type; difficult to distribute



Actors

- Communication is **unidirectional** and **possibly unordered** (selective receive)
- Named processes, associated with incoming **mailbox**
- Easy to distribute; difficult to type



Future: Placeholder variable

Can be written once, read many times

Multiple writes: error

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.
```

```
full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.
```

```
main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, 5 },
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Protocol violation

Two 'put' messages.
Manifests as a runtime error.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, 5 },
  Future ! { put, 10 },
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Protocol violation

No 'put' message.
Future never resolved.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Protocol violation

No 'reply' message.

Requests go unanswered.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, 5 },
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Unexpected Message

Message is never handled.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, 5 },
  Future ! { surprise, 10 },
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Payload Mismatch

Client code expects an integer; gets a string.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, "hello" },
  Future ! { get, self() },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end.
```

Self-deadlock

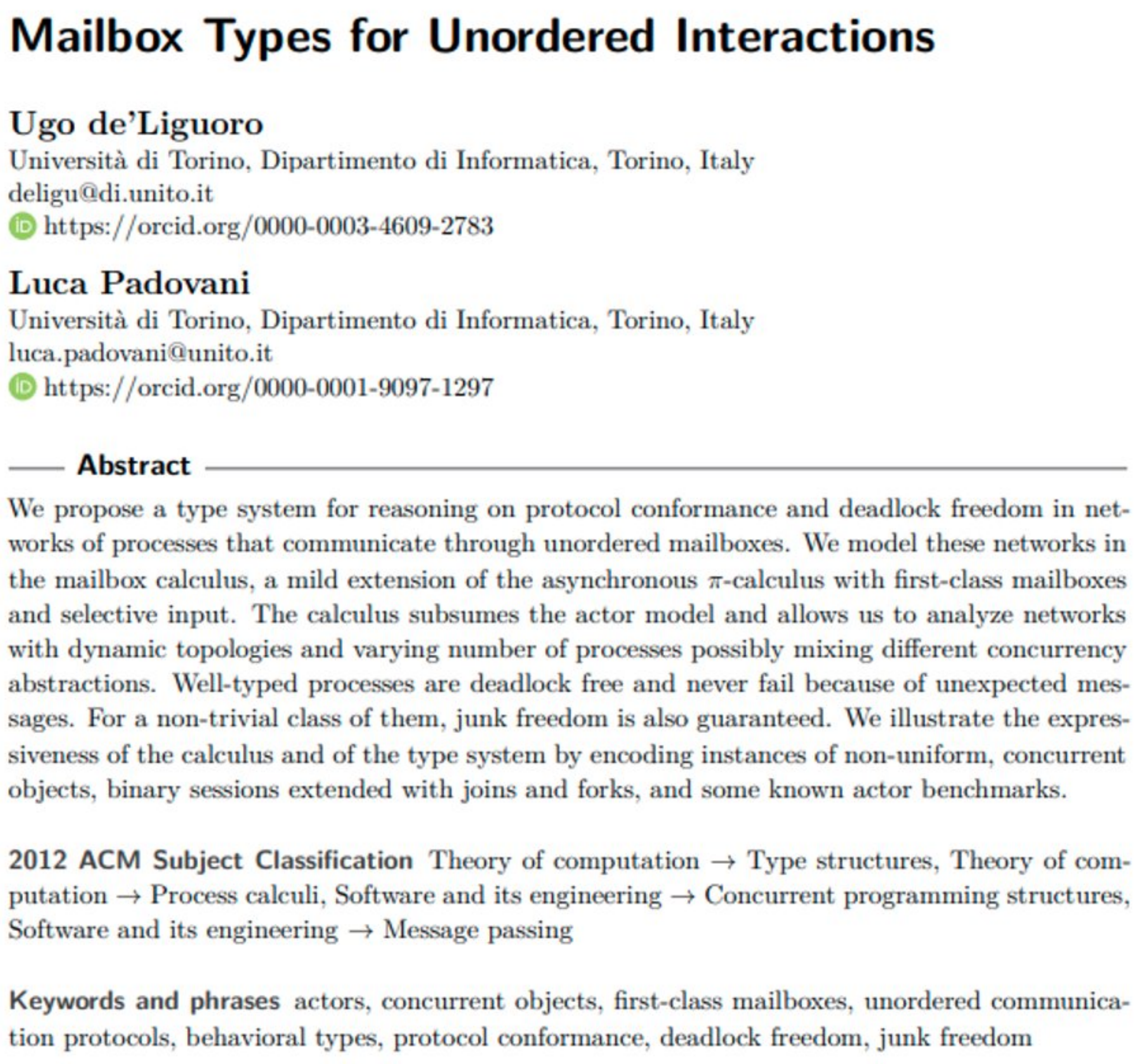
Attempting to read a reply message before sending a request.

```
empty_future() ->
  receive
    { put, X } -> full_future(X)
  end.

full_future(X) ->
  receive
    { get, Pid } ->
      Pid ! { reply, X },
      full_future(X);
    { put, _ } ->
      erlang:error("Multiple writes")
  end.

main() ->
  Future = spawn(future, empty_future, []),
  Future ! { put, 5 },
  receive
    { reply, Result } ->
      io:fwrite("~w~n", [Result + 10])
  end,
  Future ! { get, self() }.
```

Mailbox Types: Type mailboxes with **commutative regular expressions**



Mailbox Types for Unordered Interactions

Ugo de'Liguoro
Università di Torino, Dipartimento di Informatica, Torino, Italy
deligu@di.unito.it
https://orcid.org/0000-0003-4609-2783

Luca Padovani
Università di Torino, Dipartimento di Informatica, Torino, Italy
luca.padovani@unito.it
https://orcid.org/0000-0001-9097-1297

Abstract

We propose a type system for reasoning on protocol conformance and deadlock freedom in networks of processes that communicate through unordered mailboxes. We model these networks in the mailbox calculus, a mild extension of the asynchronous π-calculus with first-class mailboxes and selective input. The calculus subsumes the actor model and allows us to analyze networks with dynamic topologies and varying number of processes possibly mixing different concurrency abstractions. Well-typed processes are deadlock free and never fail because of unexpected messages. For a non-trivial class of them, junk freedom is also guaranteed. We illustrate the expressiveness of the calculus and of the type system by encoding instances of non-uniform, concurrent objects, binary sessions extended with joins and forks, and some known actor benchmarks.

2012 ACM Subject Classification Theory of computation → Type structures, Theory of computation → Process calculi, Software and its engineering → Concurrent programming structures, Software and its engineering → Message passing

Keywords and phrases actors, concurrent objects, first-class mailboxes, unordered communication protocols, behavioral types, protocol conformance, deadlock freedom, junk freedom

$$E, F \quad ::= \quad \mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F \mid E \odot F \mid E^*$$

Unreliable
Empty mailbox
Mailbox containing message with tag 'm'
E or F
Mailbox containing E and F
Mailbox containing many Es

A **mailbox type** is a **capability** associated with a **pattern**:

$$!E \qquad ?E$$

Key ideas:

- Each mailbox has **many** send references, but **precisely one** receive reference
- Sends and receives must balance out
- Subtyping: relies on **pattern inclusion**

$$
\begin{array}{rcl}
\mathsf{EmptyFuture} & \triangleq & ?(\mathtt{Put}[\mathsf{Int}] \odot \mathtt{Get}[\mathsf{ClientSend}]^{*}) \\
\mathsf{FullFuture} & \triangleq & ?\mathtt{Get}[\mathsf{ClientSend}]^{*} \\
\mathsf{ClientRecv} & \triangleq & ?\mathtt{Reply}[\mathsf{Int}] \\
\mathsf{ClientSend} & \triangleq & !\mathtt{Reply}[\mathsf{Int}]
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{emptyFuture}(\mathit{self}) & \triangleq & \mathit{self}?\mathtt{Put}(x)\,.\,\mathsf{fullFuture}(\mathit{self}, x) \\
\mathsf{fullFuture}(\mathit{self}, x) & \triangleq & \mathbf{free}\ \mathit{self}\,.\,\mathbf{done} \\
& + & \mathit{self}?\mathtt{Get}(\mathit{sender})\,.\,(\mathit{sender}!\mathtt{Reply}[x] \parallel \mathsf{fullFuture}(\mathit{self}, x)) \\
& + & \mathit{self}?\,\mathtt{Put}(x)\,.\,\mathbf{fail}\ \mathit{self}
\end{array}
$$

$$
\begin{array}{l}
(\nu \mathit{future})(\mathsf{emptyFuture}(\mathit{future}) \parallel \mathit{future}!\mathtt{Put}[5] \parallel \\
\qquad (\nu \mathit{self})(\mathit{future}!\mathtt{Get}[\mathit{self}] \parallel (\mathit{self}?\mathtt{Reply}(x)\,.\,\mathbf{free}\ \mathit{self}\,.\,\mathsf{print}(\mathsf{intToString}(x))))
\end{array}
$$

A process calculus shows a **snapshot of a concurrent system**

A programming language must be able to describe the **program a user writes**

```
def emptyFuture(self : EmptyFuture): 1 {
  guard self {
    receive Put [ x ] from self ↦ fullFuture(self, x)
  }
}

def fullFuture(self : FullFuture, value : Int): 1 {
  guard self {
    free ↦ ()
    receive Get [ user] from self ↦
      user ! Reply [ value] ;
      fullFuture(self, value)
  }
}
```

```
def client(): 1 {
  let future = new in
  spawn emptyFuture(future);
  let self = new in
  future ! Put [ 5 ] ;
  future ! Get [ self] ;
  guard self {
    receive Reply [ result] from self ↦
      free self;
      print(intToString(result))
  }
}
```

Demo

Language Integration

Challenge 1: Static / Dynamic Distinction

$$\begin{aligned}&(\nu \mathit{future})(\mathsf{emptyFuture}(\mathit{future}) \parallel \mathit{future}\,!\,\texttt{Put}[5] \parallel \\ &\qquad (\nu \mathit{self})(\mathit{future}\,!\,\texttt{Get}[\mathit{self}] \parallel (\mathit{self}\,?\,\texttt{Reply}(x)\,.\,\mathbf{free}\ \mathit{self}\,.\,\mathsf{print}(\mathsf{intToString}(x)))\end{aligned}$$

Names

- Process calculus: know runtime names *a priori* as they are part of a process
- In a PL: only *dynamic*: generated by the semantics.
- Distinction incompatible with alias control / deadlock-freedom techniques used by the mailbox calculus

Sequential composition?

Variable rebinding?

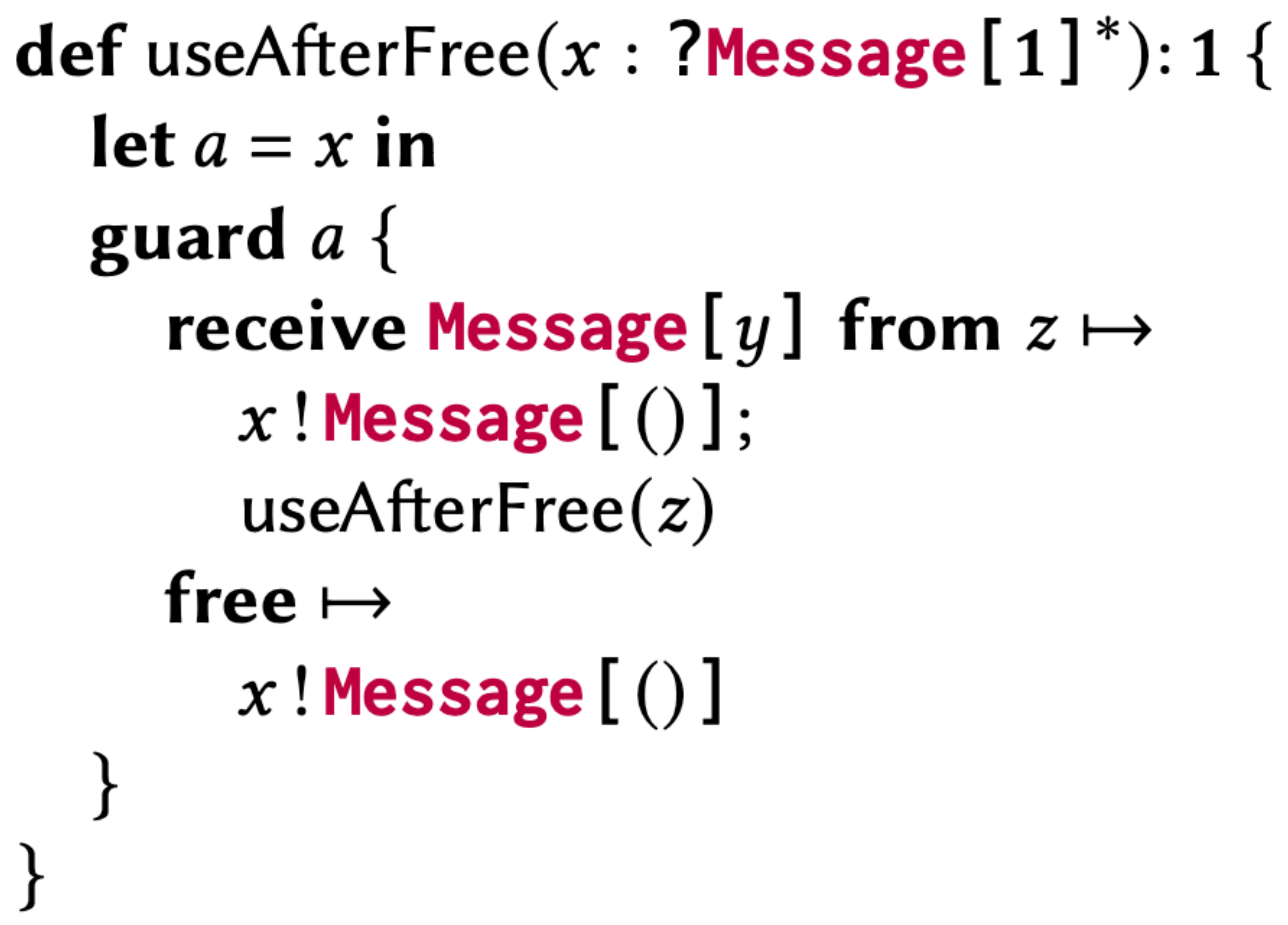
Challenge 2: Name hygiene

```
def useAfterFree(x : ?Message[1]*): 1 {
  guard x {
    receive Message[y] from z ↦
      x ! Message[()];
      useAfterFree(z)
    free ↦
      x ! Message[()]
  }
}
```

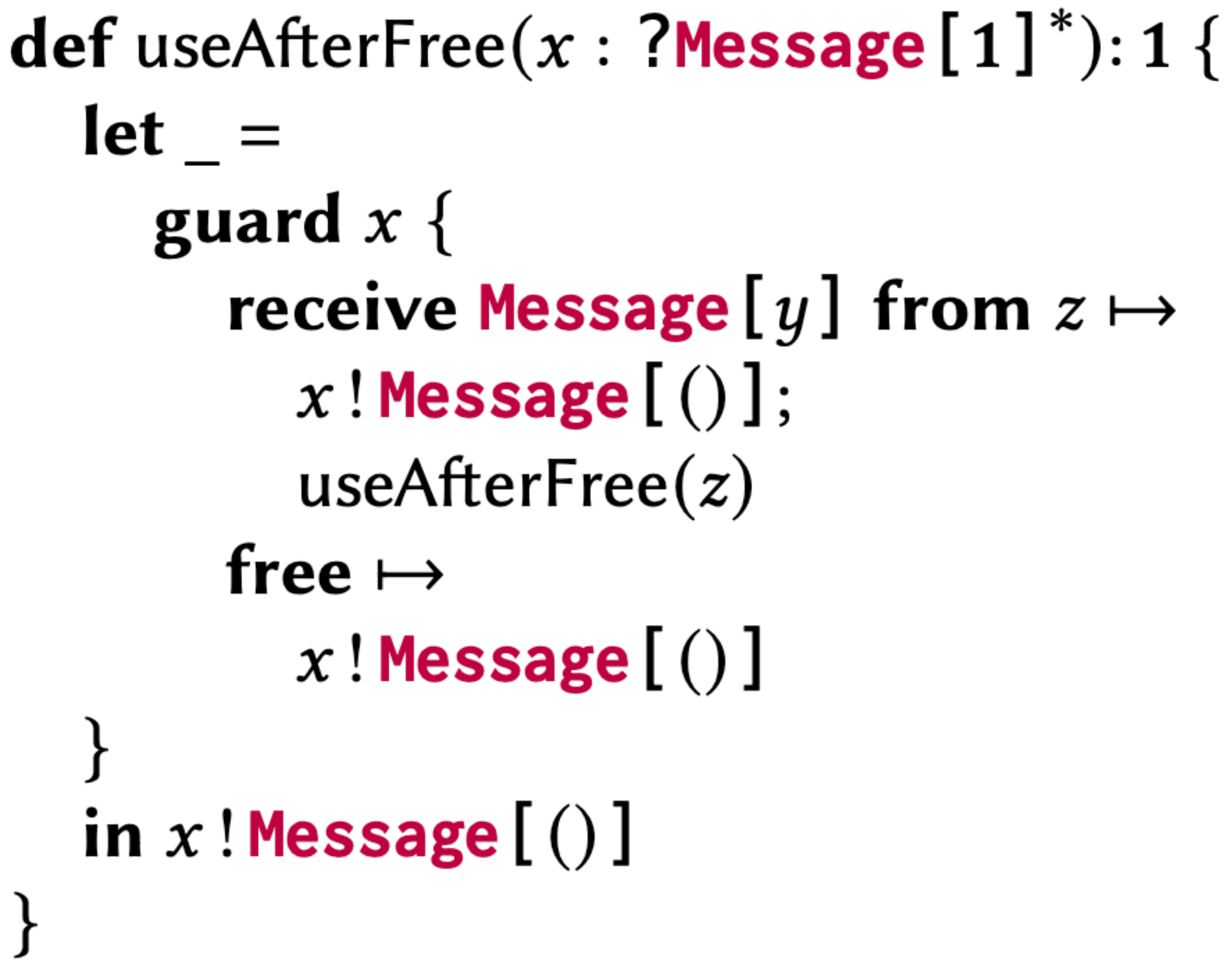
A guard 'uses up' a variable; x must not be in scope afterwards

Easy to do in a linear system; more difficult in a multi-writer system where variables can be used more than once

Challenge 2: Name hygiene



```
def useAfterFree(x : ?Message[1]*): 1 {
  let a = x in
  guard a {
    receive Message[y] from z ↦
      x ! Message[()];
      useAfterFree(z)
    free ↦
      x ! Message[()]
  }
}
```



```
def useAfterFree(x : ?Message[1]*): 1 {
  let _ =
    guard x {
      receive Message[y] from z ↦
        x ! Message[()];
        useAfterFree(z)
      free ↦
        x ! Message[()]
    }
  in x ! Message[()]
}
```

...and must be robust to renaming / aliasing, and evaluation contexts!

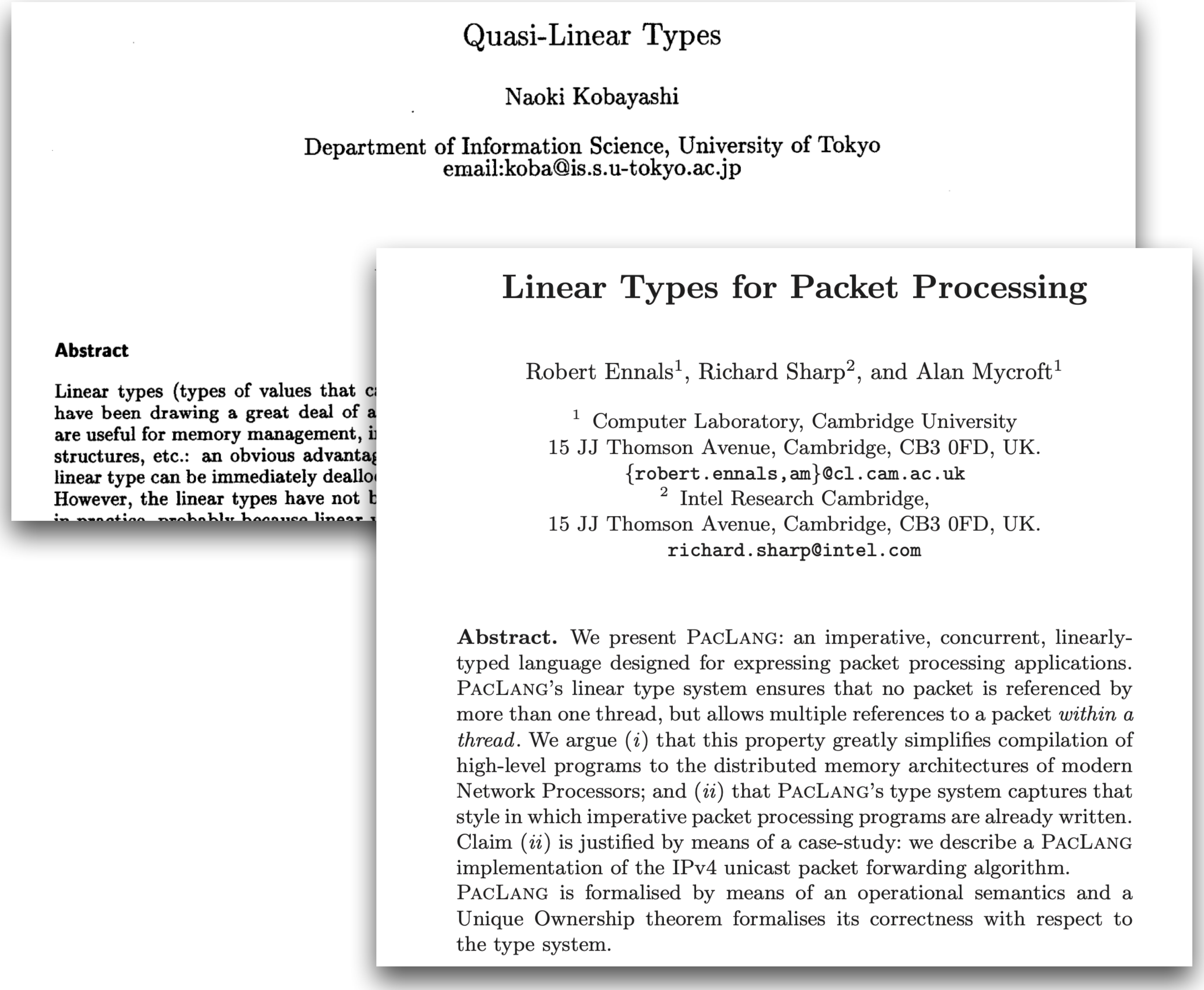
Need: Only one variable name in scope for each runtime name

Challenge 3: Aliasing via Communication

$$a \leftarrow \mathtt{m}[b] \quad \| \quad \begin{array}{l} \textbf{guard}\ a\ \{ \\ \quad \textbf{receive}\ \mathtt{m}[x]\ \textbf{from}\ y \mapsto \\ \quad\quad b\,!\,\mathtt{n}[x]; \\ \quad\quad \textbf{free}\ y \\ \} \end{array} \qquad \longrightarrow \qquad \begin{array}{l} b\,!\,\mathtt{n}[b]; \\ \textbf{free}\ a \end{array}$$

Cannot allow communication to introduce unsafe aliases!

Quasi-Linear Types



Quasi-Linear Types

Naoki Kobayashi

Department of Information Science, University of Tokyo
email:koba@is.s.u-tokyo.ac.jp

Abstract

Linear types (types of values that c
have been drawing a great deal of a
are useful for memory management, i
structures, etc.: an obvious advantag
linear type can be immediately deallo
However, the linear types have not b

Linear Types for Packet Processing

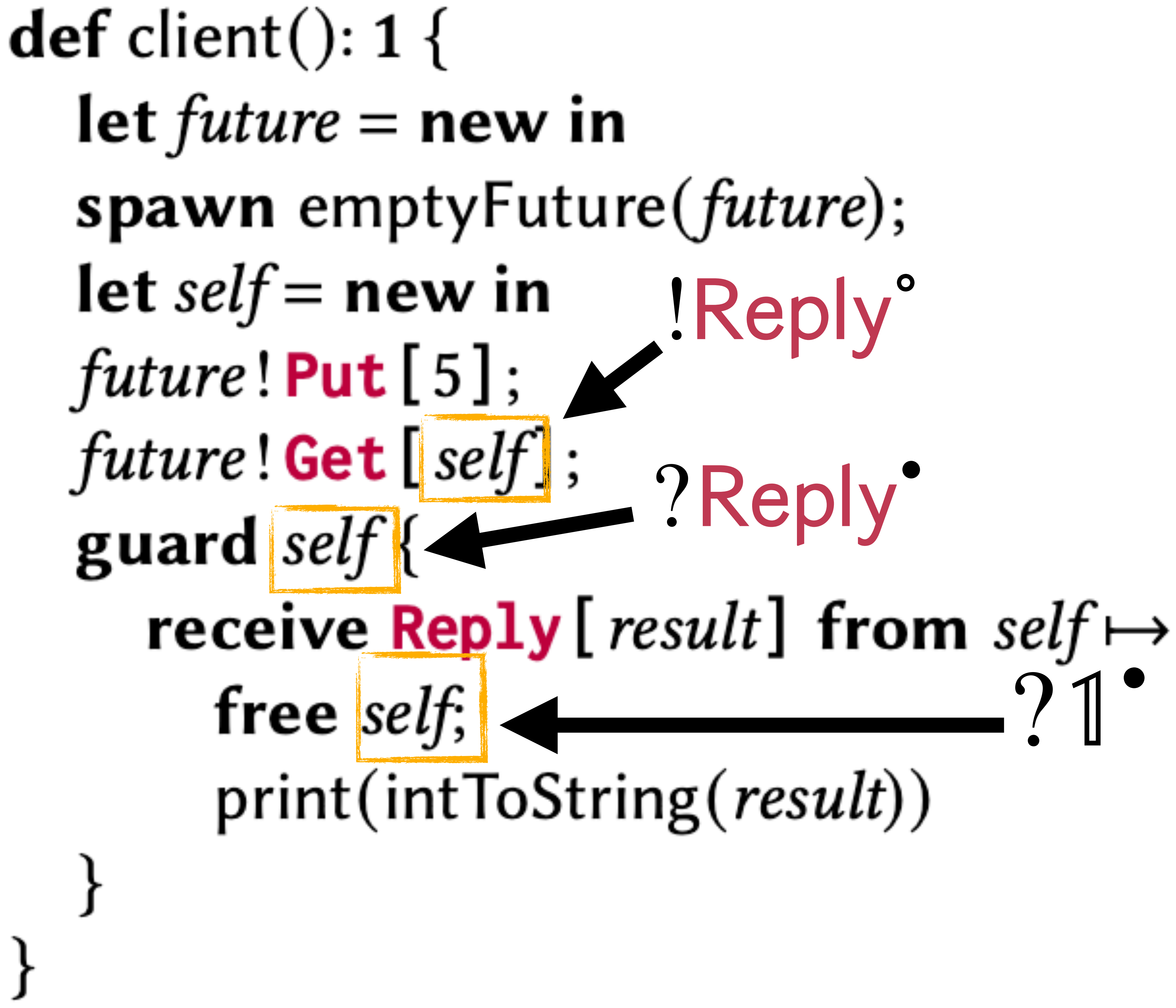
Robert Ennals[1], Richard Sharp[2], and Alan Mycroft[1]

[1] Computer Laboratory, Cambridge University
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK.
`{robert.ennals,am}@cl.cam.ac.uk`
[2] Intel Research Cambridge,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK.
`richard.sharp@intel.com`

Abstract. We present PACLANG: an imperative, concurrent, linearly-typed language designed for expressing packet processing applications. PACLANG's linear type system ensures that no packet is referenced by more than one thread, but allows multiple references to a packet *within a thread*. We argue (*i*) that this property greatly simplifies compilation of high-level programs to the distributed memory architectures of modern Network Processors; and (*ii*) that PACLANG's type system captures that style in which imperative packet processing programs are already written. Claim (*ii*) is justified by means of a case-study: we describe a PACLANG implementation of the IPv4 unicast packet forwarding algorithm. PACLANG is formalised by means of an operational semantics and a Unique Ownership theorem formalises its correctness with respect to the type system.

- Quasi-linear typing: each reference can be used **once per process** as a full ("returnable") reference, but many times as a partial ("usable") reference
- Returnable references can be let-bound; returned as part of an expression; and guarded upon
- Returnable reference must be the **last occurrence** of the name in the thread
- Usable references can only be used as the target of a send

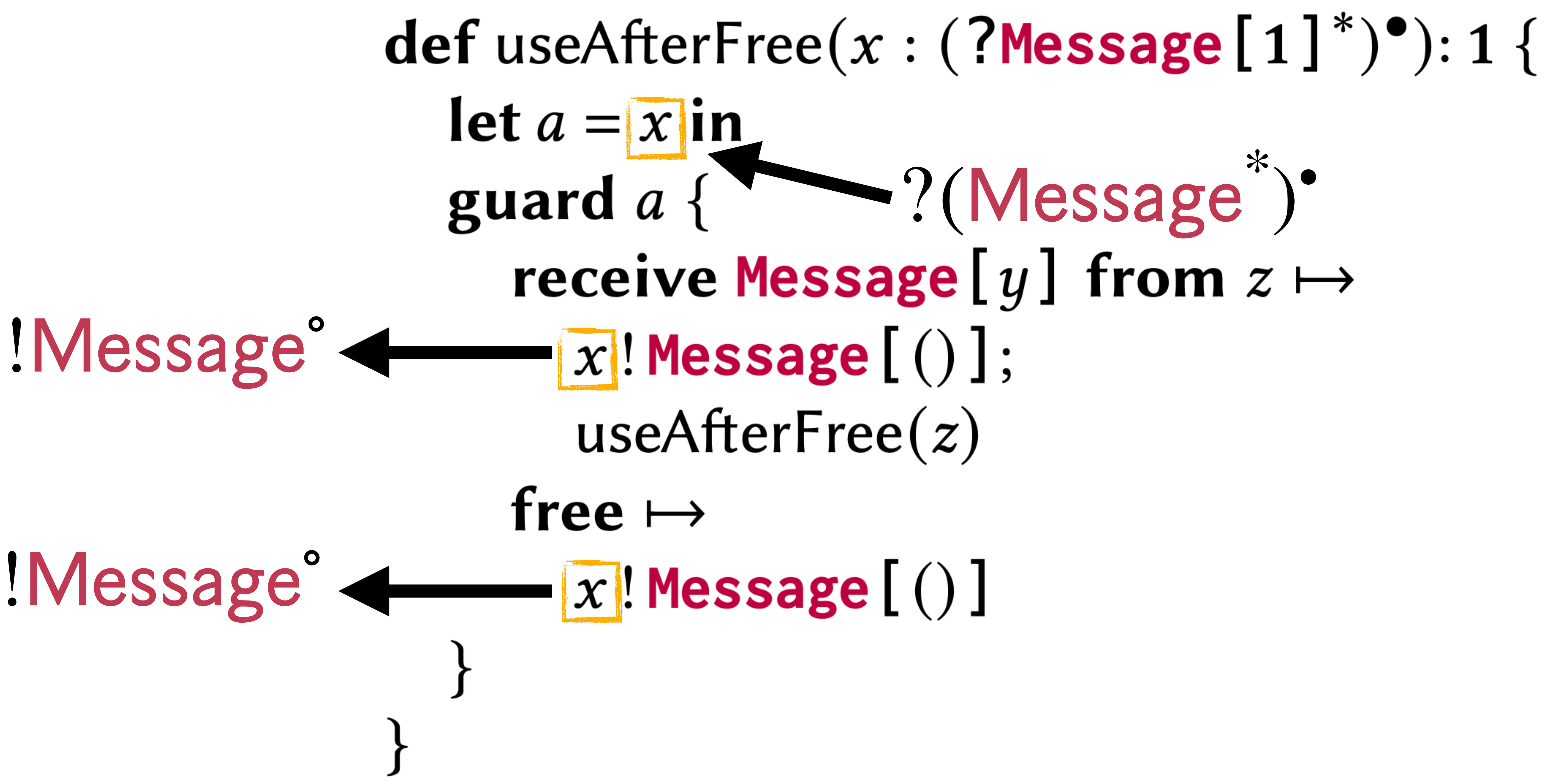
Quasi-Linear Types: Example



```
def client(): 1 {
    let future = new in
    spawn emptyFuture(future);
    let self = new in
    future ! Put [5];
    future ! Get [self];
    guard self {
        receive Reply [result] from self ↦
            free self;
            print(intToString(result))
    }
}
```

Typable: Returnable use **always last in scope** (note that 'self' is consumed by 'guard' and rebound)

Quasi-Linear Types: Example



```
def useAfterFree($x$ : $(?\mathsf{Message}[1]^*)^\bullet$): 1 {
    let $a = \boxed{x}$ in
    guard $a$ {
        receive Message[$y$] from $z$ ↦
            $\boxed{x}$! Message[()];
            useAfterFree($z$)
        free ↦
            $\boxed{x}$! Message[()]
    }
}
```

$?(\mathsf{Message}^*)^\bullet$

$!\mathsf{Message}^\circ$

$!\mathsf{Message}^\circ$

Untypable: variable 'x' appears *after* returnable occurrence

Mailbox types	J, K	$::=$	$!E \mid ?E$
Mailbox patterns	E, F	$::=$	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F \mid E \odot F \mid E^*$
Base types	C	$::=$	$\mathsf{1} \mid \mathsf{Int} \mid \mathsf{String} \mid \cdots$
Types	T, U	$::=$	$C \mid J$
Usage annotations	η	$::=$	$\circ \mid \bullet$
Usage-annotated types	A, B	$::=$	$C \mid J^\eta$
Variables	x, y, z		
Definition names	f		
Definitions	D	$::=$	$\textbf{def } f(\overrightarrow{x : A}) \colon B\ \{M\}$
Values	V, W	$::=$	$x \mid c$
Terms	L, M, N	$::=$	$V \mid \textbf{let } x \colon T = M \textbf{ in } N \mid f(\overrightarrow{V})$
		$\mid$	$\textbf{spawn } M \mid \textbf{new} \mid V\,!\,\mathtt{m}[\overrightarrow{W}] \mid \textbf{guard } V\ \{\overrightarrow{G}\}$
Guards	G	$::=$	$\textbf{fail} \mid \textbf{free} \mapsto M \mid \textbf{receive } \mathtt{m}[\overrightarrow{x}] \textbf{ from } y \mapsto M$

Selected Typing Rules (Send)

Message with tag **m** has payload types $\overrightarrow{T}$

Target must have usable mailbox type ! **m**

$$\dfrac{\begin{array}{c}\boxed{\mathcal{S}(\mathbf{m}) = \overrightarrow{T}} \qquad \boxed{\Gamma \vdash V : !\,\mathbf{m}^{\circ}} \\ \boxed{(\Gamma'_i \vdash W_i : \lceil T_i \rceil)_{i \in 1..n}}\end{array}}{\boxed{\Gamma + \Gamma'_1 + \ldots + \Gamma'_n \vdash V\,!\,\mathbf{m}[\overrightarrow{W}] : \mathbf{1}}}$$

Payload types must match (usable)

Send term has **unit type**, no shared linear variables between target and each payload

Selected Typing Rules (Let)

Ensure subject of **let** has **returnable** type

$$\frac{\boxed{\Gamma_1 \vdash M : \lfloor T \rfloor} \qquad \Gamma_2, x : \lfloor T \rfloor \vdash N : B}{\boxed{\Gamma_1 \triangleright \Gamma_2 \vdash \mathbf{let}\; x : T = M \;\mathbf{in}\; N : B}}$$

Sequencing of environments: ensures mailbox types combine correctly, ensure quasilinear well-formedness properties

Note: Type annotation **optional**

Type Combination

$$!E \boxplus !F = !(E \odot F)$$

$$!E \boxplus ?(E \odot F) = ?F \qquad ?(E \odot F) \boxplus !E = ?F$$

Combining two send mailbox types: send **both**

Combining a send and receive: types **balance out**

$$\frac{}{\circ \rhd \circ = \circ} \qquad \frac{}{\circ \rhd \bullet = \bullet}$$

Can combine two usable types, or a usable and a returnable type

Note: Not reflexive, nor symmetric

Selected Typing Rules (New and Spawn)

$$\frac{}{\cdot \vdash \mathbf{new} : \boxed{?\mathbb{1}^{\bullet}}}$$

New mailbox: Returnable, empty receive capability.
Ensures all sends balanced by receives

$$\frac{\Gamma \vdash M : \mathbf{1}}{\boxed{\lceil \Gamma \rceil} \vdash \mathbf{spawn}\ M : \mathbf{1}}$$

Spawn: environment treated as **usable** (quasi-linearity is **thread-local)**

Metatheory

Theorem (Preservation):
If Γ is reliable, $\Gamma \vdash \mathscr{C}$, and $\mathscr{C} \longrightarrow \mathscr{D}$, then $\Gamma \vdash \mathscr{D}$.

Corollary (Mailbox Conformance):
If Γ is reliable and $\Gamma \vdash \mathscr{C}$, then $\mathscr{C} \not\longrightarrow^{*} \mathscr{G}[\mathbf{fail}\, V]$.

Nontrivial: requires extensive reasoning about contexts and quasi-linearity

Algorithmic Typing & Implementation

How do we write a typechecker?

The declarative system cannot be implemented as-is:

- Nondeterministic environment & type splitting
- Environment subtyping
- Pattern inclusion

Key idea: **Produce** a type environment and a set of pattern inclusion constraints

Bidirectional Typing

$$\boxed{\Gamma \vdash M \Rightarrow A}$$

"Under type environment Γ, we can ***synthesise*** *type A for term M"*

$$\boxed{\Gamma \vdash M \Leftarrow A}$$

"Under type environment Γ, we can ***check*** *that term M has type A"*

Backwards Bidirectional Typing

$$\boxed{P \Rightarrow \tau \blacktriangleright \Theta; \Phi}$$

"Synthesise type τ for term P,
***producing** type environment Θ and*
pattern inclusion constraints Φ"

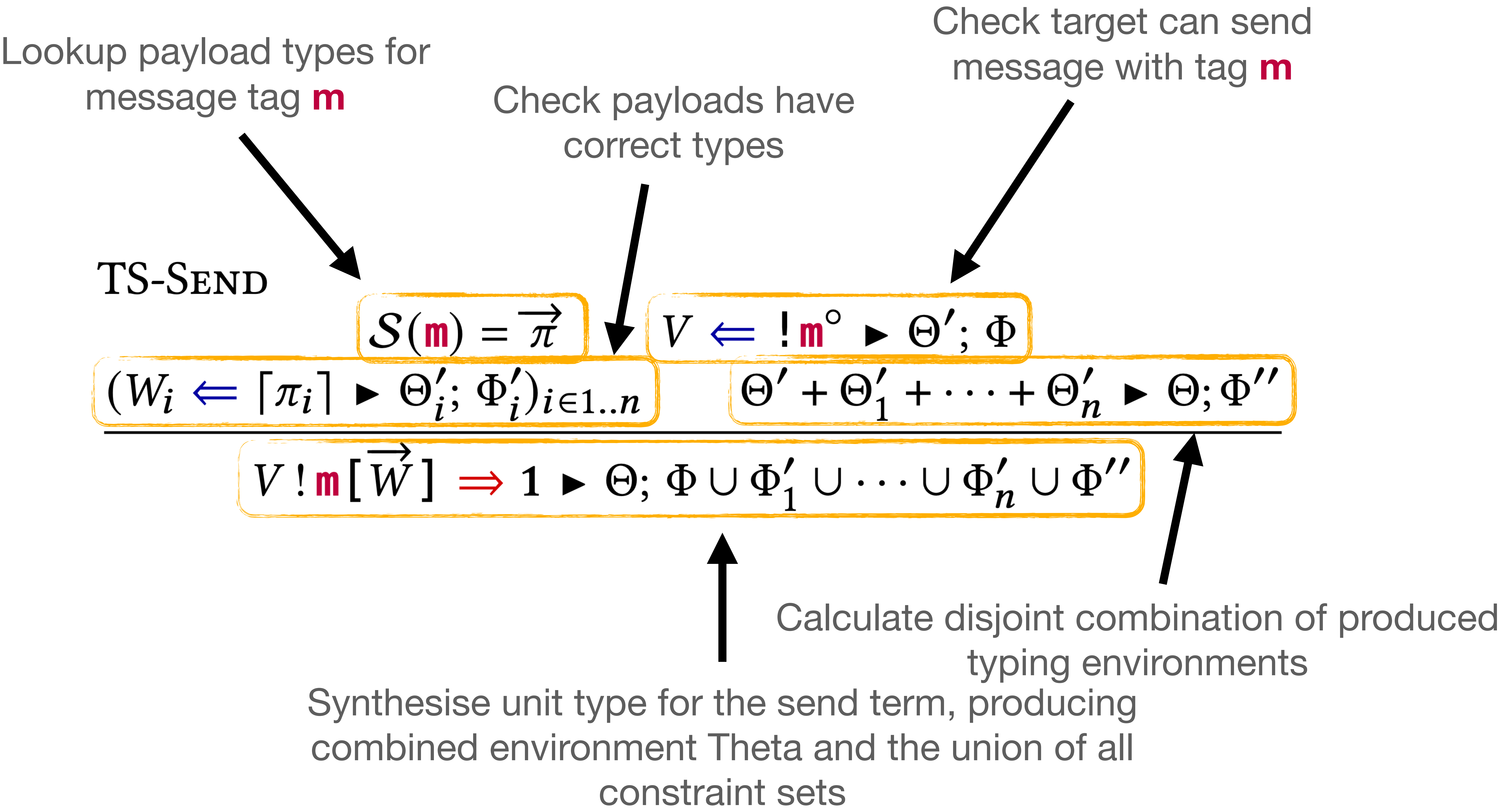
$$\boxed{P \Leftarrow \tau \blacktriangleright \Theta; \Phi}$$

"Check that term P has type type τ,
***producing** type environment Θ and*
pattern inclusion constraints Φ"

Key idea: Stay in checking mode as much as possible to preserve type information and propagate to variables
(originally introduced by Zeilberger, 2015)

$$\frac{\textsc{TC-Var}}{x \Leftarrow \tau \blacktriangleright x : \tau;\ \emptyset}$$

Variable rule: a **checking** case, constructing a singleton environment



Lookup payload types for message tag **m**

Check payloads have correct types

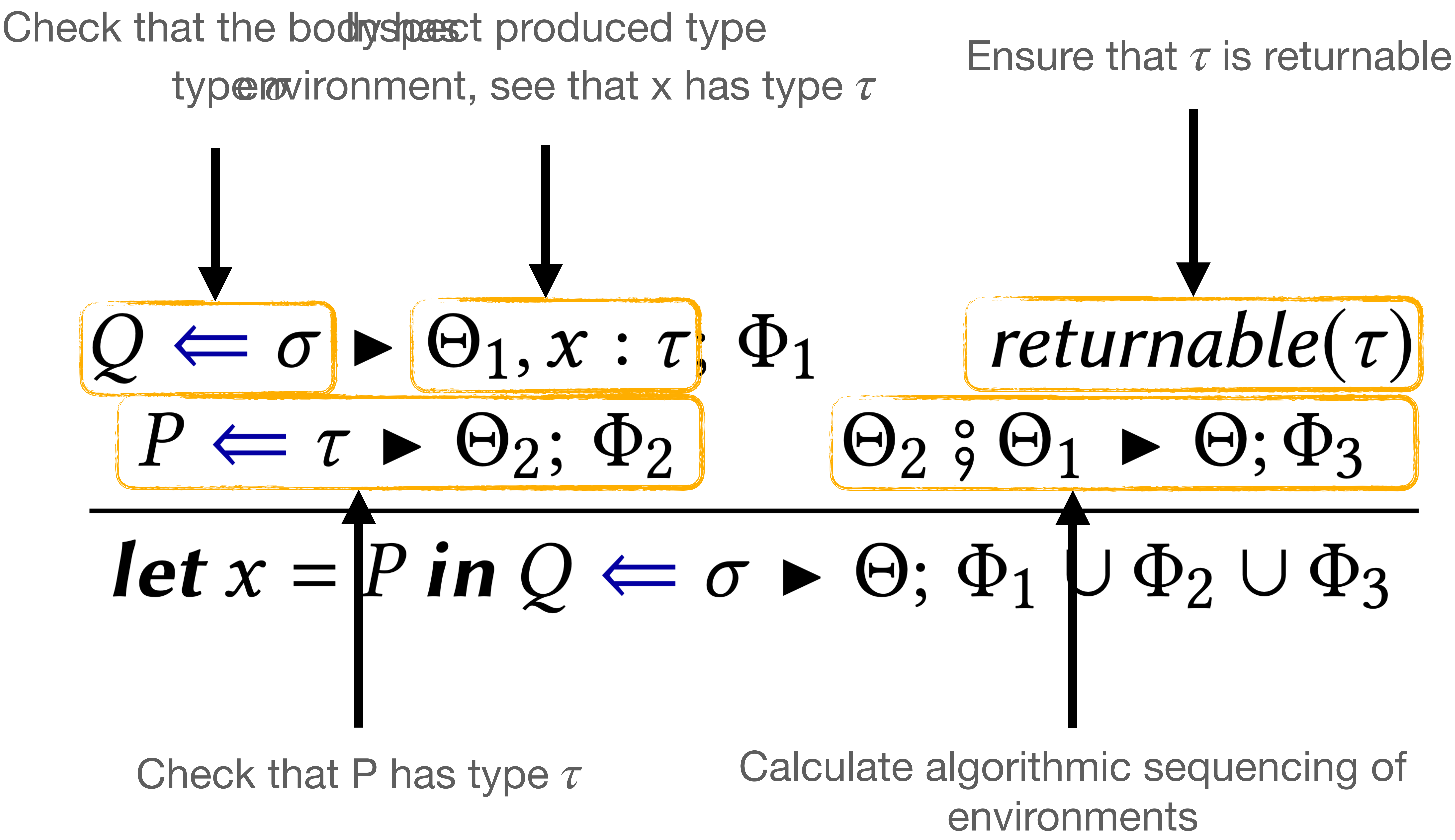
Check target can send message with tag **m**

TS-Send

$$\frac{\boxed{\mathcal{S}(\mathtt{m}) = \overrightarrow{\pi}} \quad \boxed{V \Leftarrow\, !\,\mathtt{m}^{\circ} \blacktriangleright \Theta'; \Phi} \\ \boxed{(W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta'_i; \Phi'_i)_{i \in 1..n}} \quad \boxed{\Theta' + \Theta'_1 + \cdots + \Theta'_n \blacktriangleright \Theta; \Phi''}}{\boxed{V\,!\,\mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta; \Phi \cup \Phi'_1 \cup \cdots \cup \Phi'_n \cup \Phi''}}$$

Calculate disjoint combination of produced typing environments

Synthesise unit type for the send term, producing combined environment Theta and the union of all constraint sets



Check that the body has type σ

Inspect produced type environment, see that x has type τ

Ensure that τ is returnable

$$\frac{\begin{array}{cc} Q \Leftarrow \sigma \blacktriangleright \Theta_1, x : \tau;\ \Phi_1 & \mathit{returnable}(\tau) \\ P \Leftarrow \tau \blacktriangleright \Theta_2;\ \Phi_2 & \Theta_2 \fatsemi \Theta_1 \blacktriangleright \Theta; \Phi_3 \end{array}}{\textbf{\textit{let}}\ x = P\ \textbf{\textit{in}}\ Q \Leftarrow \sigma \blacktriangleright \Theta;\ \Phi_1 \cup \Phi_2 \cup \Phi_3}$$

Check that P has type τ

Calculate algorithmic sequencing of environments

Note: Revert to synthesis if x is not used in Q

Metatheory

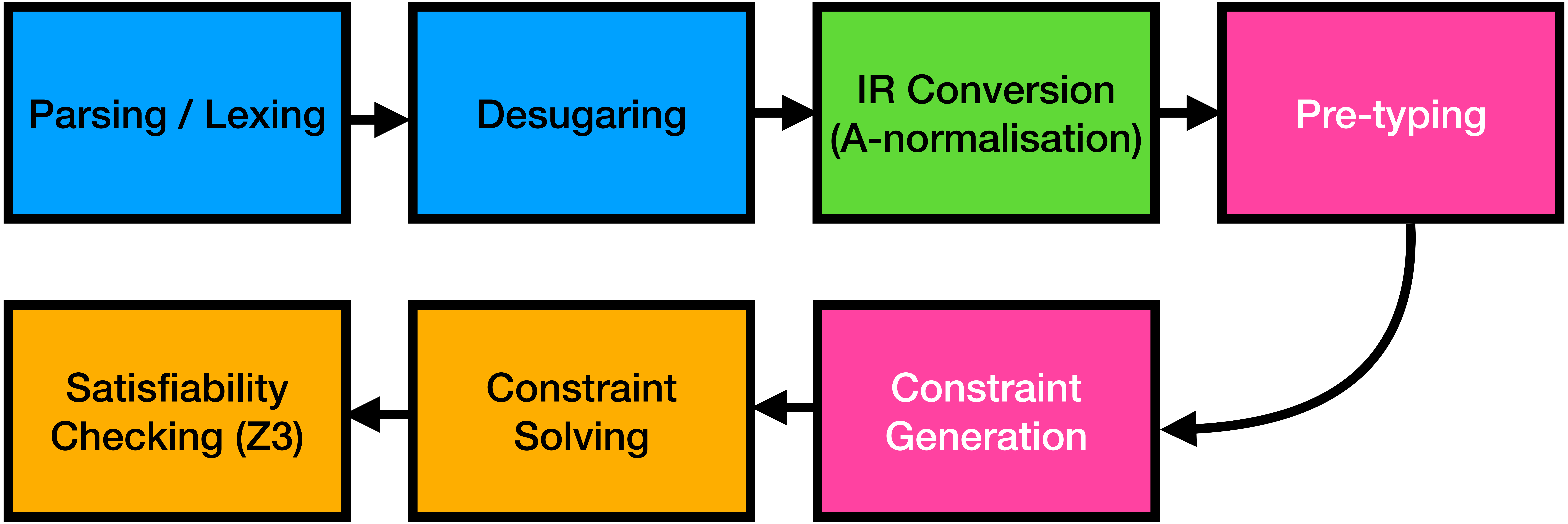
Theorem (Algorithmic Soundness)

- If $P \Leftarrow \tau \blacktriangleright \Theta; \Phi$ and Ξ is a usable solution for Φ, then $\Xi(\Theta) \vdash \Xi(P) : \Xi(\tau)$
- If $P \Rightarrow \tau \blacktriangleright \Theta; \Phi$ and Ξ is a usable solution for Φ, then $\Xi(\Theta) \vdash \Xi(P) : \Xi(\tau)$

Conjecture (Algorithmic Completeness)

If $\Gamma \vdash M : A \rightsquigarrow P$, then there exist some Θ, Φ and a usable solution Ξ of Φ such that $P \Leftarrow A \blacktriangleright \Theta; \Phi$ where $\Gamma \leq \Xi(\Theta)$.

Implementation



Pattern inclusion: closed form solution thanks to Hopkins & Kozen (1999)

Check **consistency** by translating into Presburger formulae & offloading to Z3

Wrapping up

Mailbox Types: Type the **mailbox**, not the **process**

First integration of mailbox types into a programming language:

- Vital use of quasi-linear types to handle many-writer, single-reader pattern

Sound and complete algorithmic type system based on **backwards bidirectional typing**

Future work

- Compilation (Ongoing, with Franek Sowul)
- Constraint-based co-contextual typing algorithm
- More sophisticated alias analysis
- Tool integration (Erlang, Elixir...)
- Other many-writer paradigms (Publish-subscribe? Typestate?)