Our sponsors make OPLSS possible.

Diamond Sponsor

Jane Street

https://www.janestreet.com/

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

SIGPLAN

https://www.sigplan.org/

Silver Sponsors

Lecture 4: Mailbox Types

Simon Gay

School of Computing Science, University of Glasgow, UK

ICFP 2023

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

The asynchronous and unidirectional communication model supported by *mailboxes* is a key reason for the success of actor languages like Erlang and Elixir for implementing reliable and scalable distributed systems. While many actors may *send* messages to some actor, only the actor may (selectively) *receive* from its mailbox. Although actors eliminate many of the issues stemming from shared memory concurrency, they remain vulnerable to communication errors such as protocol violations and deadlocks.

Mailbox types are a novel behavioural type system for mailboxes first introduced for a process calculus by de'Liguoro and Padovani in 2018, which capture the contents of a mailbox as a commutative regular expression. Due to aliasing and nested evaluation contexts, moving from a process calculus to a programming language is challenging. This paper presents Pat, the first programming language design incorporating mailbox types, and describes an algorithmic type system. We make essential use of quasi-linear typing to tame some of the complexity introduced by aliasing. Our algorithmic type system is necessarily co-contextual, achieved through a novel use of backwards bidirectional typing, and we prove it sound and complete with respect to our declarative type system. We implement a prototype type checker, and use it to demonstrate the expressiveness of Pat on a factory automation case study and a series of examples from the Savina actor benchmark suite.

Special Delivery: Programming with Mailbox Types (Extended Version)

Submitted to JFP

SIMON FOWLER

University of Glasgow, UK

(email: Simon.Fowler@glasgow.ac.uk*)*

DUNCAN PAUL ATTARD

University of Malta, Malta

(email: Duncan.Attard@um.edu.mt*)*

DANIELLE MARSHALL

University of Glasgow, UK

(email: Danielle.Marshall@glasgow.ac.uk*)*

SIMON J. GAY

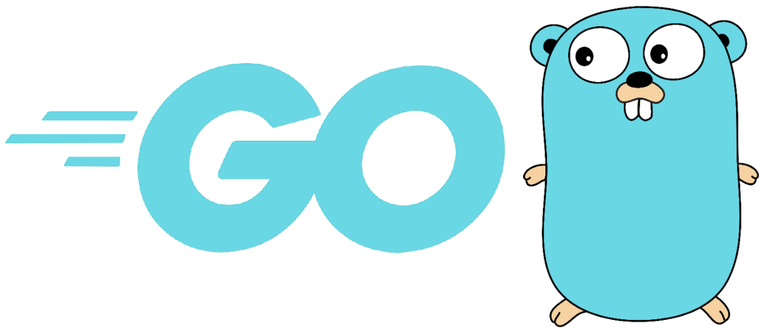
University of Glasgow, UK

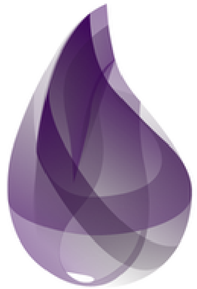
(email: Simon.Gay@glasgow.ac.uk*)*

PHIL TRINDER

University of Glasgow, UK

(email: Phil.Trinder@glasgow.ac.uk*)*

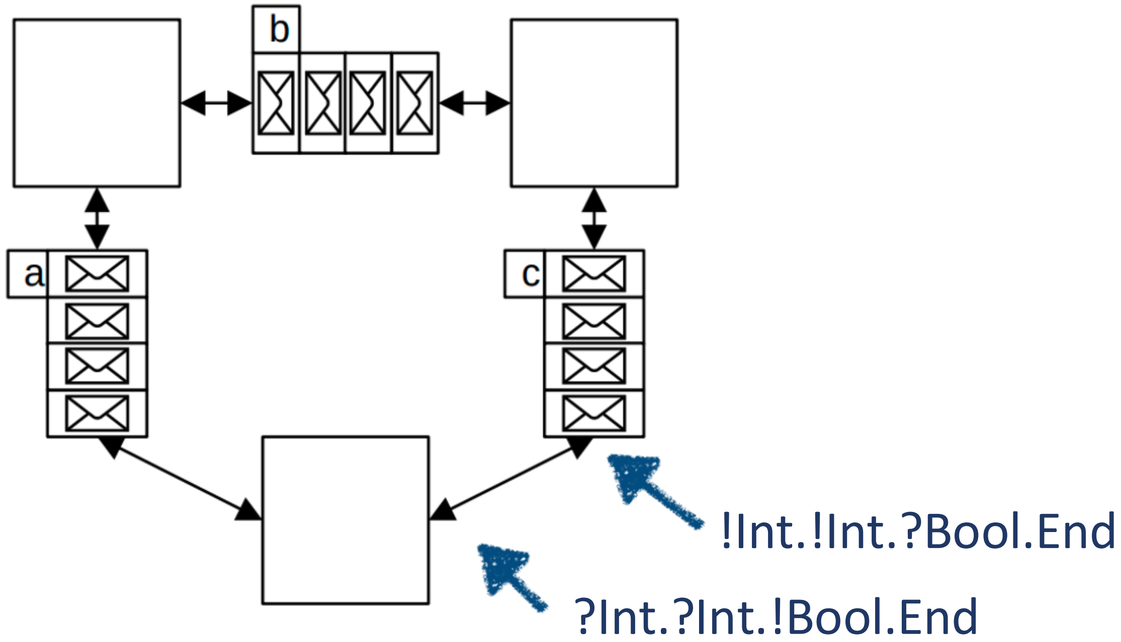


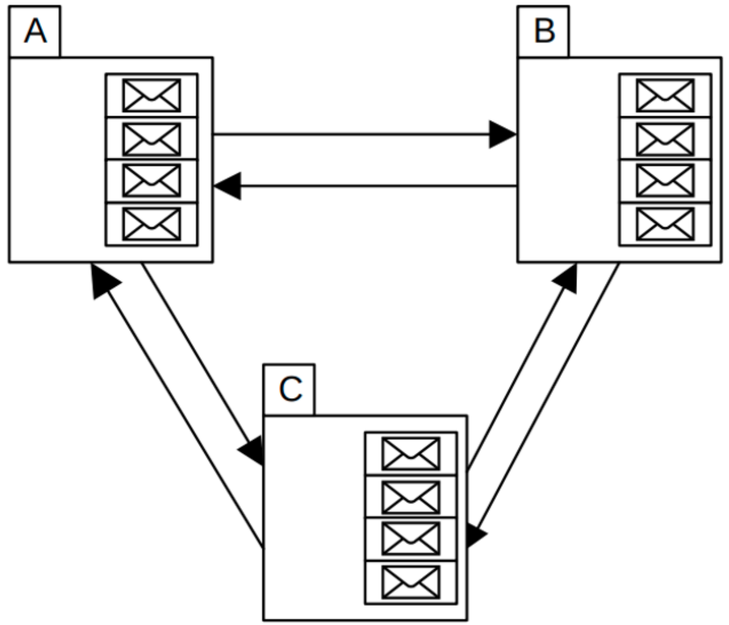


elixir

Communication-centric programming languages: **isolated threads** which communicate using **explicit message passing**

Avoids many pitfalls of shared memory, but there are still **potential communication errors to eliminate.**





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

Session types for actors

A few researchers have worked on session type systems for actor communication:

Dimitris Mostrous and Vasco Vasconcelos. Session Typing for a Featherweight Erlang. COORDINATION 2011.

Simon Fowler. An Erlang Implementation of Multiparty Session Actors. ICE 2016.

Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. LMCS 2017.

Paul Harvey, Simon Fowler, Ornela Dardha and Simon Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. ECOOP 2021.

It’s difficult to develop a good session type system that is faithful to the actor model.

Session types for actors

We (Glasgow, Kent, Oxford) had a funded project (2020-2024) to try do fault-tolerant session types for actors.

Distributed software systems are an essential part of the infrastructure of modern society. Such systems typically comprise diverse software components deployed across networks of hosts. Ensuring their reliability is challenging, as software components must correctly communicate and synchronise with each other, and any of the hardware or software components may fail. Failure and service "outage" is extremely costly, with worldwide financial losses due to software failures in 2017 estimated at US$1.7tn, up from US$1.1tn in 2016.

Failures can occur at all levels of the system stack: hardware, operating systems, networks, software, and users. Here we focus on using advanced programming language technologies to enable the software level to handle failures that arise from any level of the stack. Our aim is to provide software-level reliability for distributed systems by combining fault prevention with fault tolerance. *The key objective is to combine the communication-structuring mechanism of session types with the scalability and fault-tolerance of actor-based software architectures.*

The result will be a well-founded theory of reliable actor programming, supported by a collection of libraries and tools, and validated on a range of case studies. Key aims are to deliver tools that provide lightweight support for developers – e.g. warning of potential issues – and to allow developers to continue to use established idioms. By doing so we aim to deliver a step change in the engineering of reliable distributed software systems.

STARDUST is funded by EPSRC (grants EP/T014512/1, EP/T014628/1, EP/T014709/1) between 1st October 2020 and 30th September 2024.

UKRI Engineering and Physical Sciences Research Council

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

ECOOP 2018

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.

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

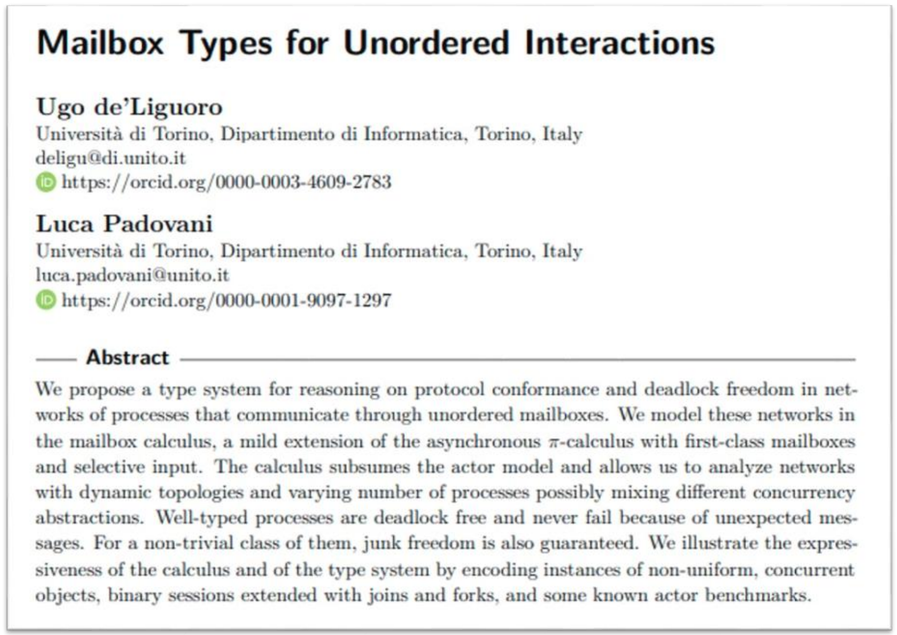
Attempting to read a reply message before sending a request.

```
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() }.
```

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



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.

$!E$

Send capability: obligation to send messages with pattern E
Possibly **many** send references per mailbox

$?E$

Receive capability: mailbox contains messages with pattern E
Only **one** receive reference per mailbox

Goal: sends and receives should **balance out**

m Message with tag m

$E \odot F$ Patterns E **and** F

$E \oplus F$ Pattern E **or** F

$\star E$ 0 or more copies of E

$\mathbb{1}$ Empty mailbox

$\mathbb{0}$ Unreliable mailbox

$$\mathsf{EmptyFuture} \triangleq ?(\mathtt{Put}[\mathsf{Int}] \odot \star\mathtt{Get}[\mathsf{ClientSend}])$$

Receive capability where mailbox contains at most one Put message, many Get messages

$$\mathsf{FullFuture} \triangleq ?\star\mathtt{Get}[\mathsf{ClientSend}]$$

Receive capability where mailbox contains many Get messages (but no Put messages)

$$\mathsf{ClientSend} \triangleq !\mathtt{Reply}[\mathsf{Int}]$$

Send capability: obligation to send a Reply message

$$\mathsf{ClientRecv} \triangleq ?\mathtt{Reply}[\mathsf{Int}]$$

Receive capability: mailbox contains single Reply message

$$
\begin{aligned}
\mathsf{emptyFuture}(\mathit{self}) &\triangleq \mathit{self}?\texttt{Put}(x)\,.\,\mathsf{fullFuture}(\mathit{self}, x)\\
\mathsf{fullFuture}(\mathit{self}, x) &\triangleq \textbf{free}\ \mathit{self}\,.\,\textbf{done}\\
&\quad + \mathit{self}?\texttt{Get}(\mathit{sender})\,.\,(\mathit{sender}!\texttt{Reply}[x] \parallel \mathsf{fullFuture}(\mathit{self}, x))\\
&\quad + \mathit{self}?\,\texttt{Put}(x)\,.\,\textbf{fail}\ \mathit{self}
\end{aligned}
$$

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

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

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

From process calculus to programming language

This turns out to be complicated!

We propose Pat, a functional programming language that supports actor-based communication with mailbox types.

Pat has first-class mailboxes – good for functional programming, but it creates challenges for applications to Erlang.

Postman Pat (and his black-and-white cat)

The Future example in Pat

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

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

Demo

Language Integration

Challenge 1: Static / Dynamic Distinction

$$(\nu future)(\text{emptyFuture}(future) \parallel future!\texttt{Put}[5] \parallel (\nu self)(future!\texttt{Get}[self] \parallel (self?\texttt{Reply}(x).\,\textbf{free}\ self.\,\text{print}(\text{intToString}(x)))$$

Names

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

Sequential composition?

Variable rebinding?

Challenge 2: Name hygiene

```
def useAfterFree1(x : ?★Msg[1]): 1 {
  guard x : ★Msg {
    receive Msg[y] from z ↦
      x ! Msg[];
      useAfterFree1(z)
    free ↦ x ! Msg[]
  }
}
```

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 useAfterFree2(x : ?★Msg[1]): 1 {
  let a = x in
  guard a : ★Msg {
    receive Msg[y] from z ↦
      x ! Msg[];
      useAfterFree2(z)
    free ↦ x ! Msg[]
  }
}
```

```
def useAfterFree3(x : ?★Msg[1]): 1 {
  let _ =
    guard x : ★Msg {
      receive Msg[y] from z ↦
        x ! Msg[];
        useAfterFree3(z)
      free ↦ x ! Msg[]
    } in x ! Msg[]
}
```

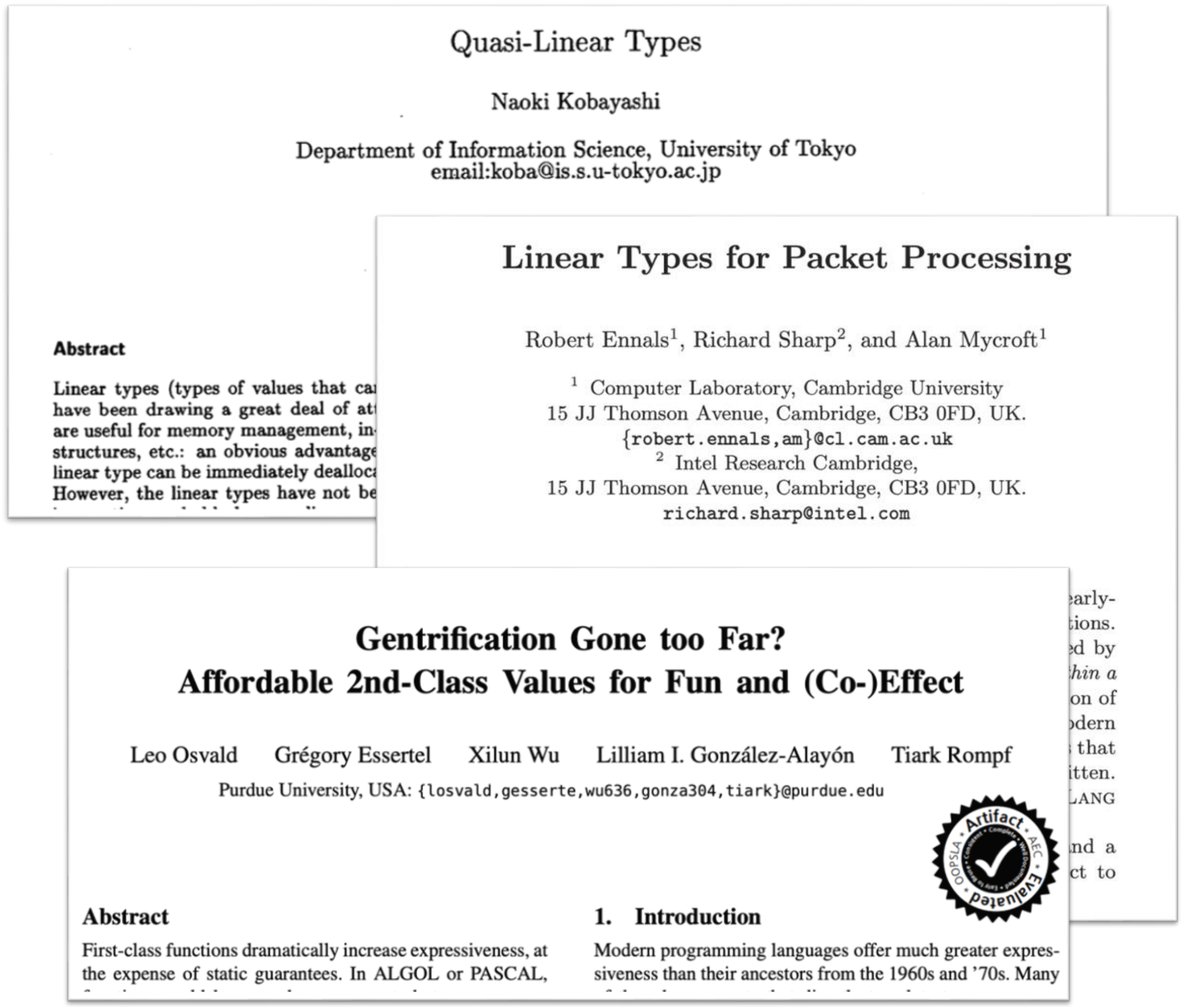
…and must be robust to renaming / aliasing, and evaluation contexts.

Challenge 3: Aliasing via Communication

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

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 ca
have been drawing a great deal of att
are useful for memory management, in
structures, etc.: an obvious advantage
linear type can be immediately dealloca
However, the linear types have not be

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

Gentrification Gone too Far?
Affordable 2nd-Class Values for Fun and (Co-)Effect

Leo Osvald Grégory Essertel Xilun Wu Lilliam I. González-Alayón Tiark Rompf

Purdue University, USA: {losvald,gesserte,wu636,gonza304,tiark}@purdue.edu

Artifact Consistent Complete Well Documented Easy to Reuse OOPSLA Evaluated

Abstract

First-class functions dramatically increase expressiveness, at the expense of static guarantees. In ALGOL or PASCAL,

1. Introduction

Modern programming languages offer much greater expressiveness than their ancestors from the 1960s and '70s. Many

- Quasi-linear typing: each reference can be used **once per thread** as a full ("returnable") reference, but many times as a 2nd class reference
- Returnable references can be let-bound; returned as part of an expression; and guarded upon
- 2nd class references cannot be guarded on or escape their scope
- Returnable reference must be the **last occurrence** of the name in thread

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))
  }
}
```

! Reply°

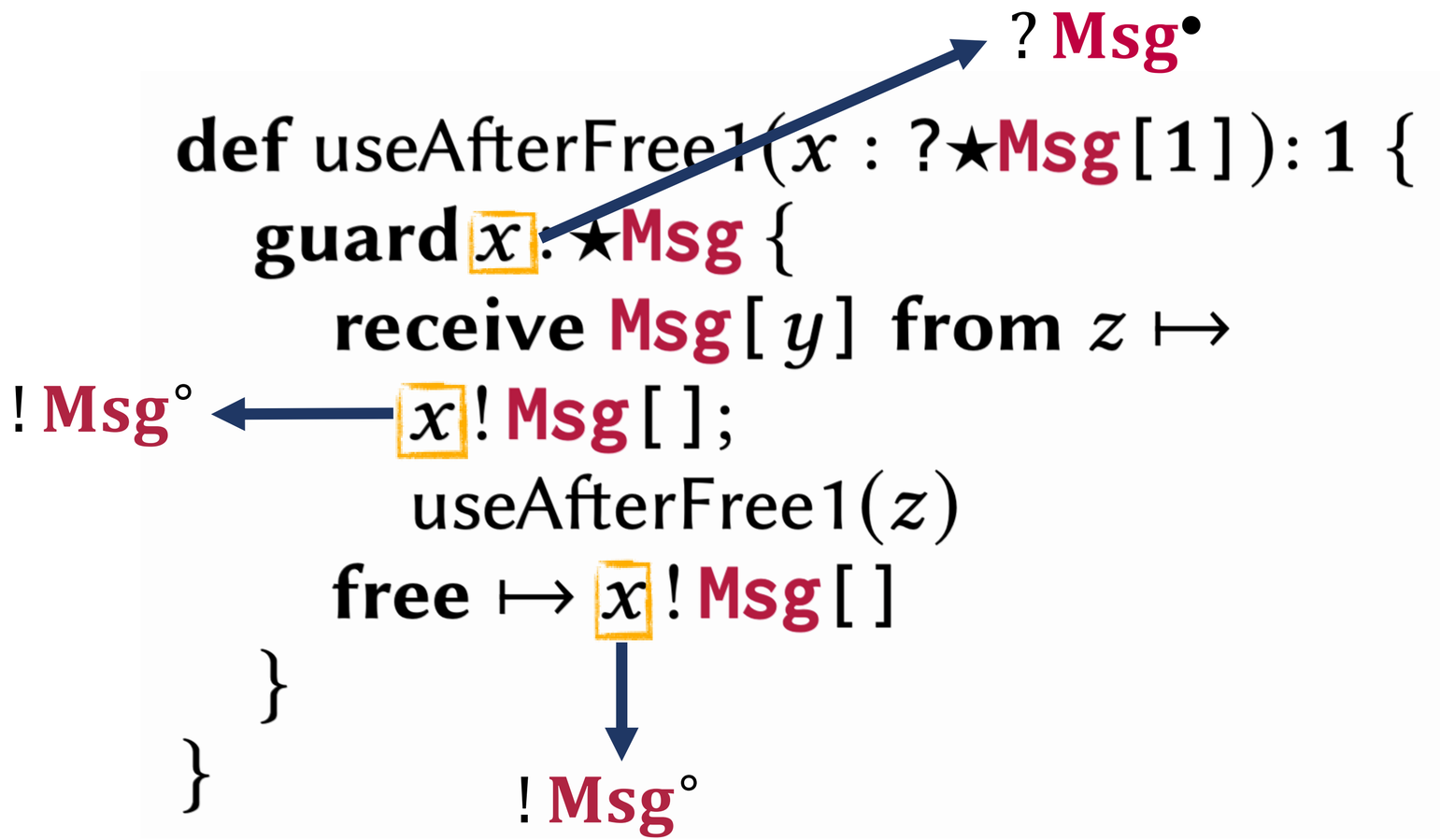
? Reply•

? 𝟙•

Note: **binding** occurrence

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

Quasi-Linear Types: Example



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

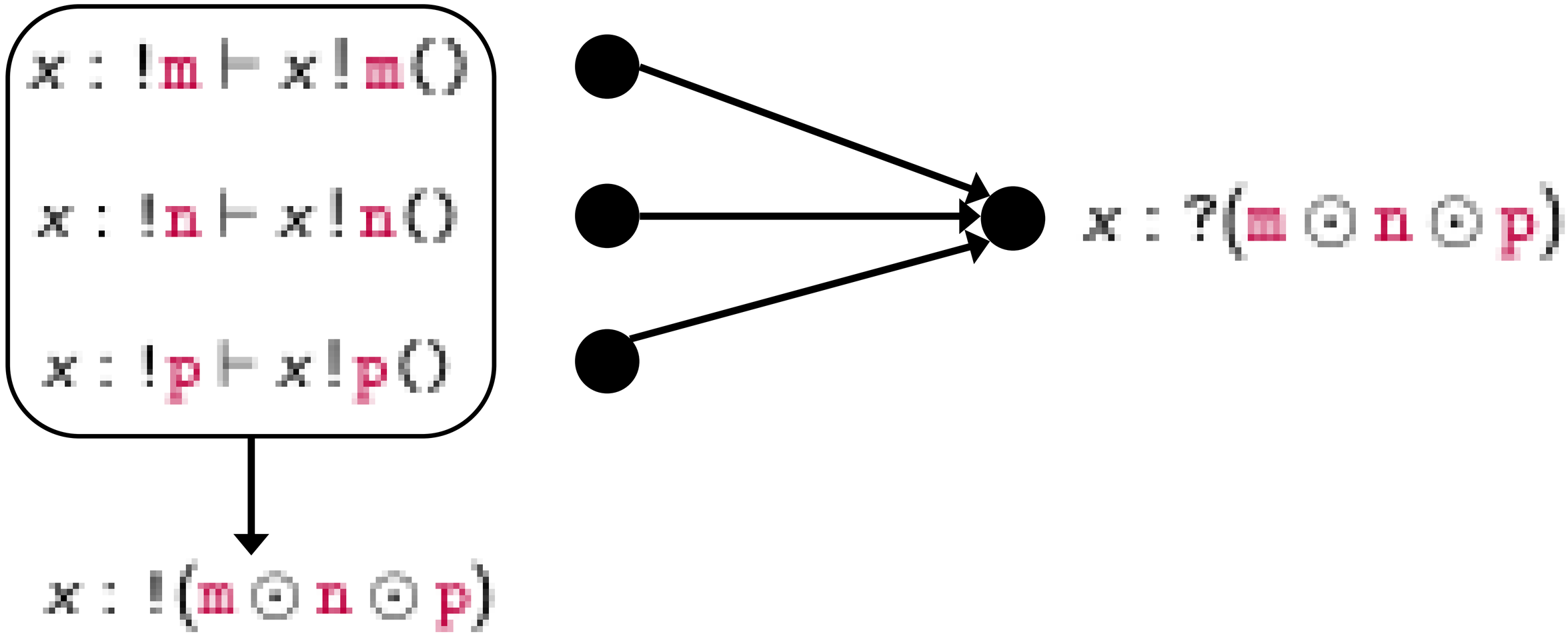
$$!E \boxplus !F = !(E \odot F)$$

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

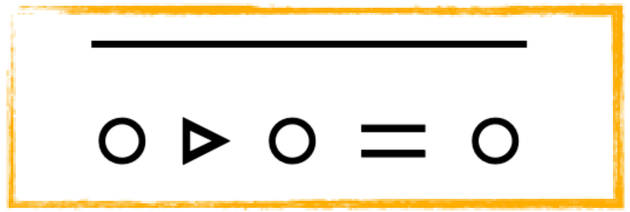
$$!E \boxplus ?(E \odot F) = ?F \qquad ?(E \odot F) \boxplus !E = ?F$$

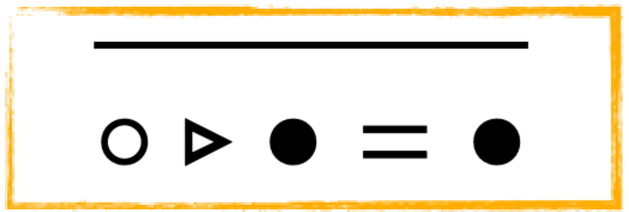
Combining a send and receive: types **balance out**

Idea: Use **environment subtyping** to rewrite into semantically-equivalent forms where types can be combined



$$!(\mathtt{m} \odot \mathtt{n} \odot \mathtt{p}) \boxplus ?(\mathtt{m} \odot \mathtt{n} \odot \mathtt{p}) = ?1$$





Can combine two 2nd class usages, or a 2nd class and a returnable usage

Note: Not commutative, and cannot combine two returnable types

Selected Typing Rules (Let)

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

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

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

Note: Type annotation **optional**

Selected Typing Rules (New and Spawn)

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

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

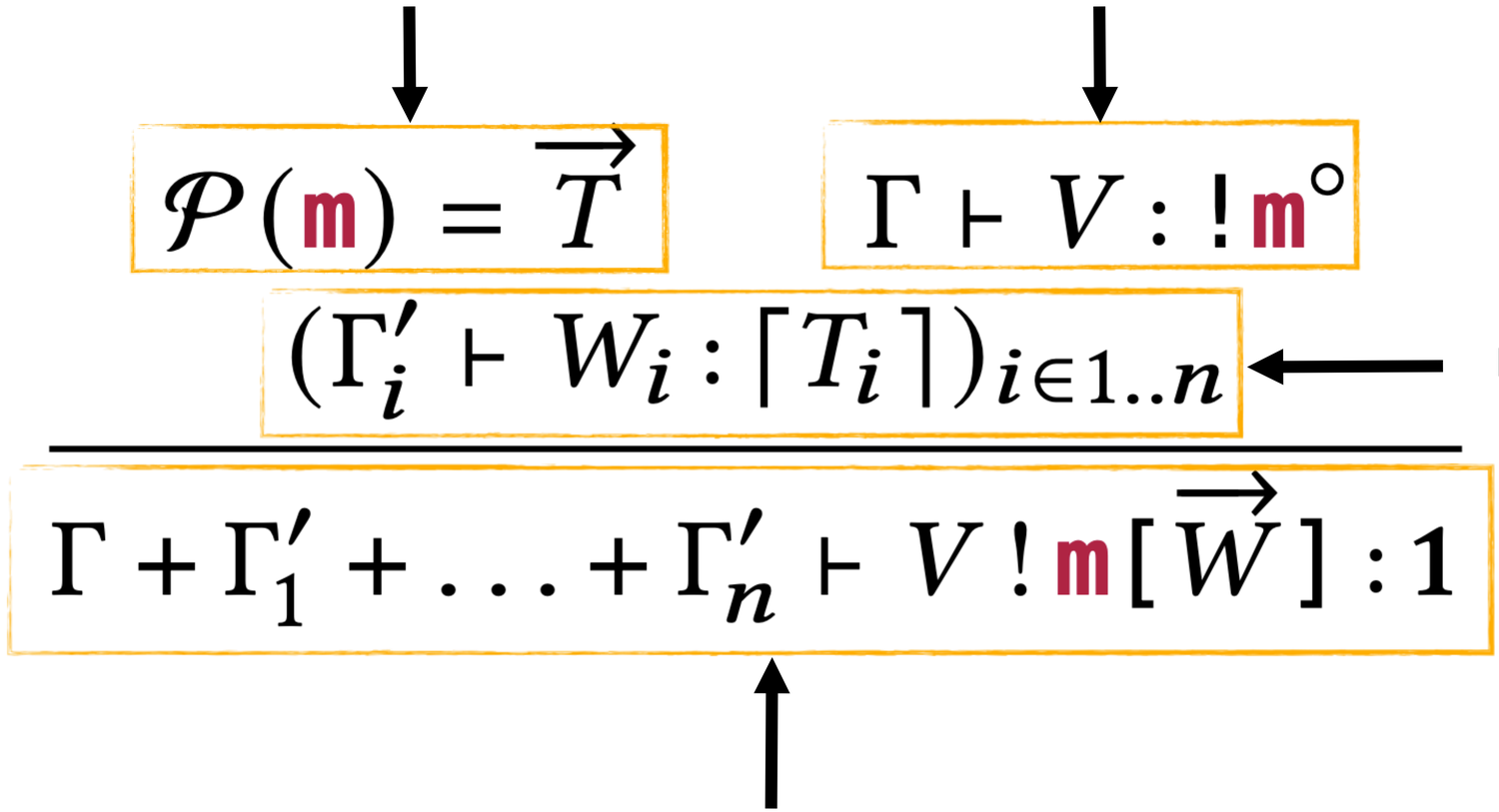
$$\frac{\Gamma \vdash M : \mathbf{1}}{\lceil \Gamma \rceil \vdash \mathbf{spawn}\ M : \mathbf{1}}$$

Spawn: environment treated as **2nd class** (quasi-linearity is **thread-local)**

Selected Typing Rule (Send)

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

Target must have 2nd-class mailbox type !**m**



Payload types must match (2nd class)

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

Example Derivation

```
let x = new in
x ! m[];
guard x : m {
    receive m[] from x → free x
}
```

Example Derivation

Metatheory

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

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

Nontrivial: frame stack representation; also 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

Instead take a **co-contextual** approach:
produce a type environment and a set of pattern inclusion constraints

Bidirectional Typing

$$\Gamma \vdash M \Rightarrow A$$

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

$$\Gamma \vdash M \Leftarrow A$$

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

Backwards Bidirectional Typing

$$M \Rightarrow_P \tau \blacktriangleright \Theta; \Phi$$

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

$$M \Leftarrow_P \tau \blacktriangleright \Theta; \Phi$$

"Check that term M has type type τ *under program P,* ***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)

Algorithmic Type Combination

$$\frac{}{!\,\gamma^{\eta_1} \fatsemi !\,\delta^{\eta_2} \blacktriangleright !\,(\gamma \odot \delta)^{\eta_1 \triangleright \eta_2}; \emptyset}$$

Algorithmic combination of send types: nothing special needed

Algorithmic combination of $!\,\gamma$ and $?\,\delta$:

$$\frac{\alpha \text{ fresh}}{!\,\gamma^{\eta_1} \fatsemi ?\,\delta^{\eta_2} \blacktriangleright ?\,\alpha^{\eta_1 \triangleright \eta_2}; \{(\gamma \odot \alpha) <: \delta\}}$$

- Generate **fresh pattern variable** α
- Generate **constraint** which says that γ concatenated with α is contained in δ
- Combined type is $?\,\alpha$ (i.e., **residual** after γ has been received

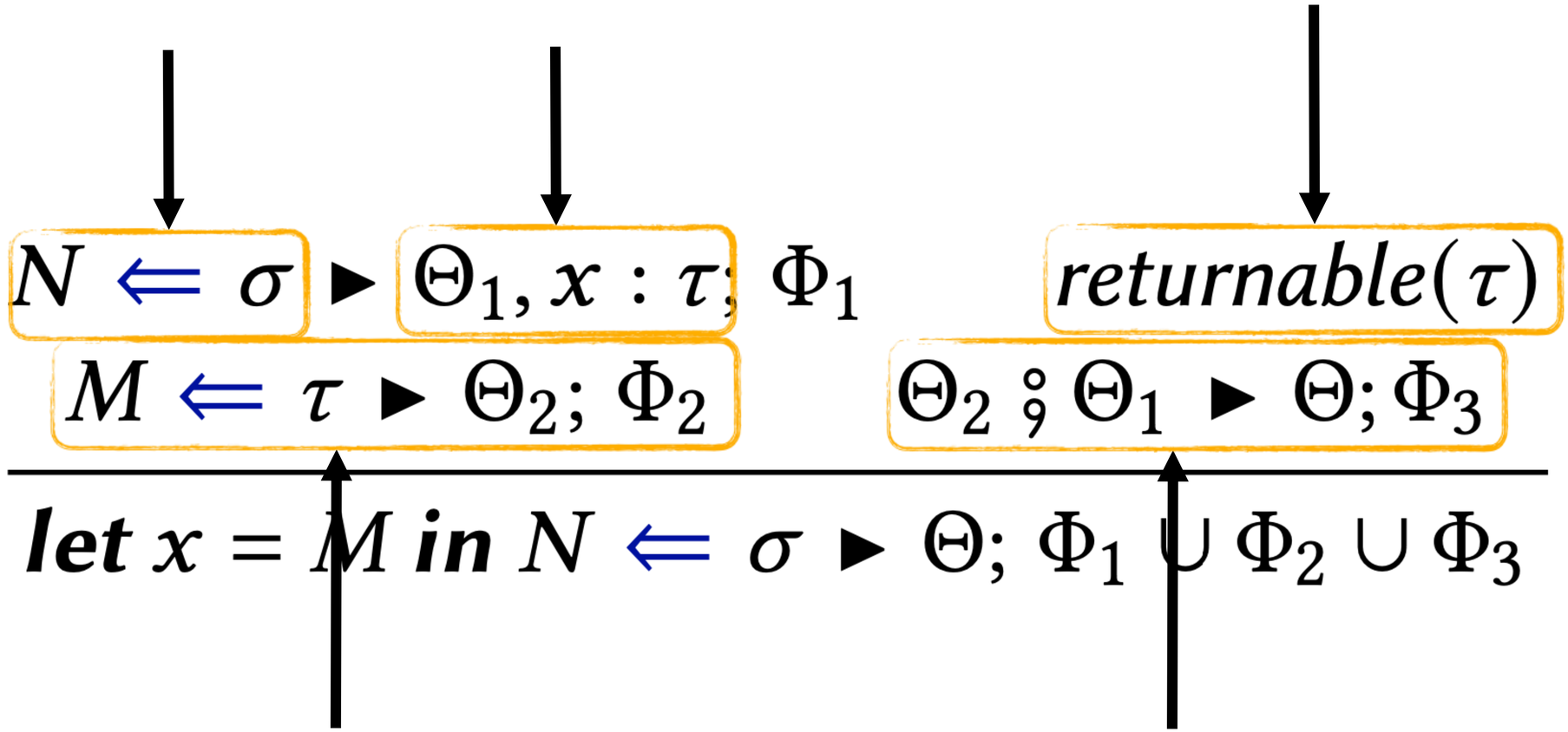
$$\textsc{TC-Var}\ \frac{}{x \Leftarrow \tau \blacktriangleright x : \tau;\ \emptyset}$$

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

Check that the body has type σ

Inspect produced type environment, see that x has type τ

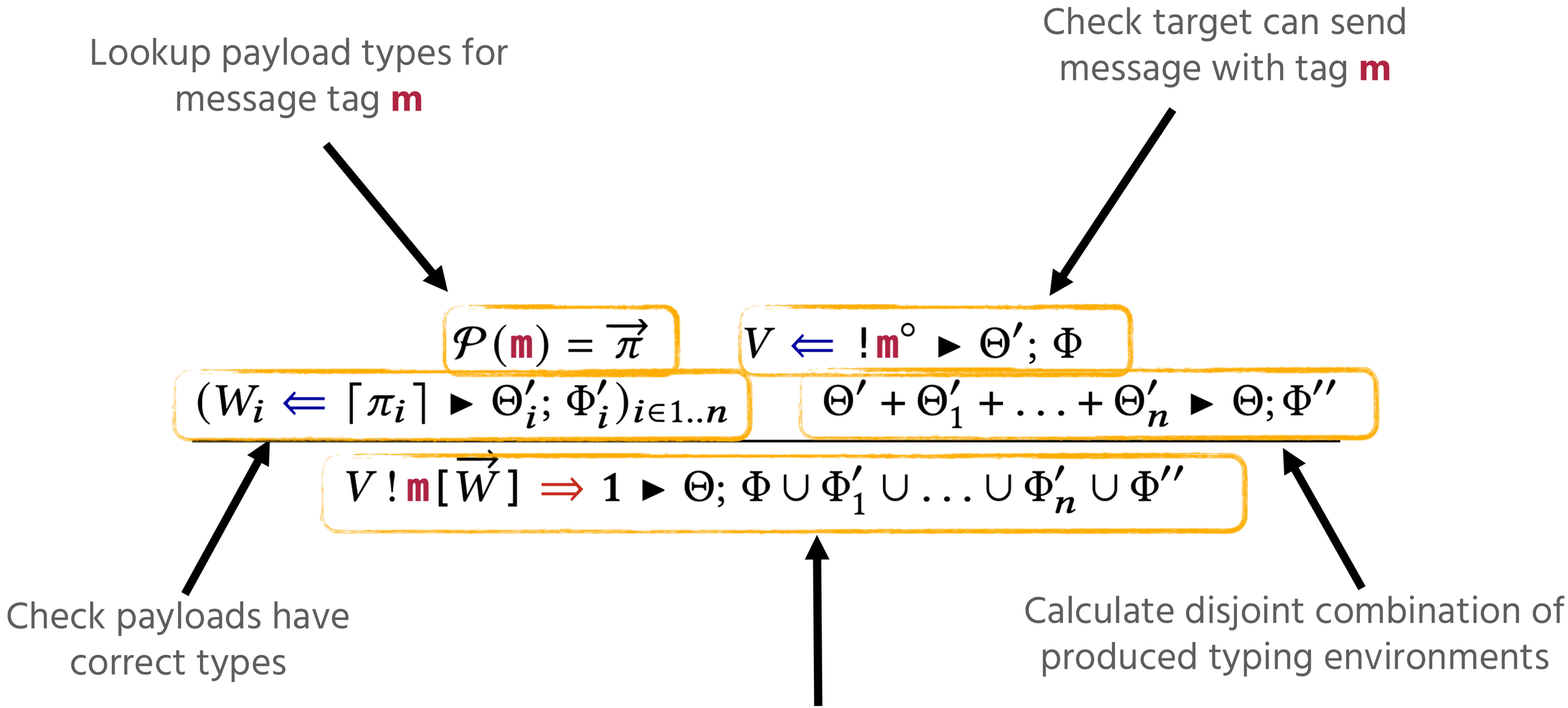
Ensure that τ is returnable



Check that M has type τ

Calculate algorithmic sequencing of environments

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


Lookup payload types for message tag m
Check target can send message with tag m
$\mathcal{P}(\mathtt{m}) = \overrightarrow{\pi}$
$V \Leftarrow \,!\mathtt{m}^{\circ} \blacktriangleright \Theta'; \Phi$
$(W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta'_i; \Phi'_i)_{i \in 1..n}$
$\Theta' + \Theta'_1 + \ldots + \Theta'_n \blacktriangleright \Theta; \Phi''$
$V\,!\mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi \cup \Phi'_1 \cup \ldots \cup \Phi'_n \cup \Phi''$
Check payloads have correct types
Calculate disjoint combination of produced typing environments
Synthesise unit type for the send term, producing combined environment and the union of all constraint sets

Metatheory

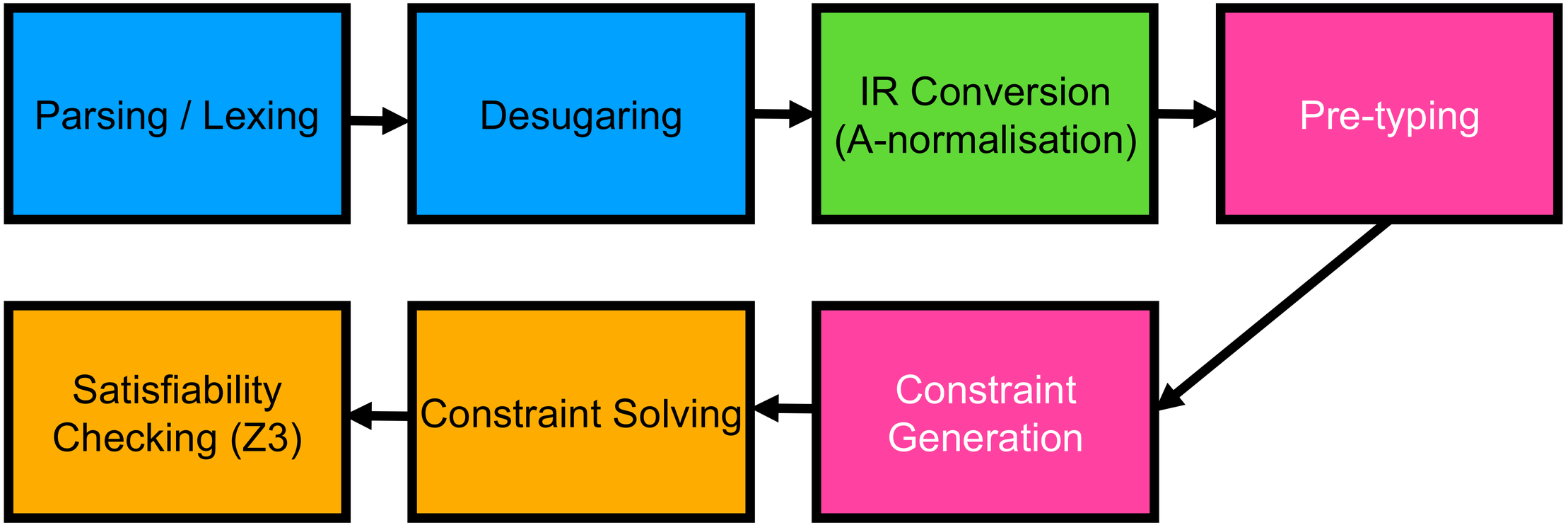
Algorithmic Soundness (Don’t accept nonsensical terms)

If a term is typable in the algorithmic system, then it is typable in the declarative system.

Algorithmic Completeness (Don’t reject sensible terms)

If a term is typable in the declarative system, then it is **checkable** in the algorithmic system

Implementation & Evaluation

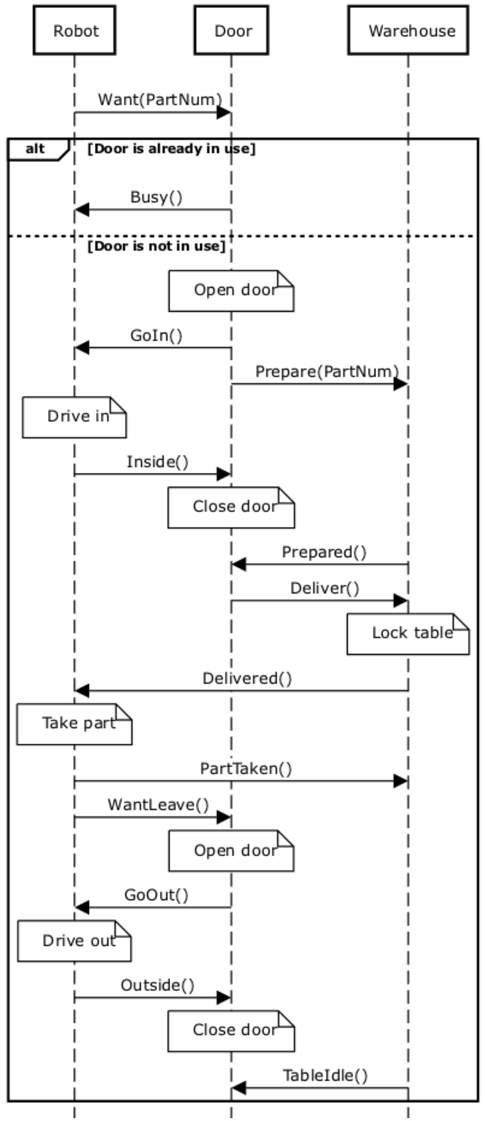


Pattern inclusion: closed form solution thanks to Hopkins & Kozen (1999)
Check **consistency** by translating into Presburger formulae & offloading to Z3

Idea due to ***A Type Checking Algorithm for Concurrent Object Protocols,*** Padovani (2018).

Global constraints + many layers => error messages can be difficult to interpret.

Factory Use Case



- Collaboration with Actyx AG
- Factory floor: multiple robots attempting to access single warehouse (mutual exclusion)
- Outside the capabilities of most session type systems
- Programming style similar to Akka Typed

Savina Benchmarks

#	Name	Description	Strict	Time (ms)
Original mailbox calculus models taken from de'Liguoro and Padovani [12]				
1	Lock	Concurrent lock modelling mutual exclusion	•	28.5
2	Future	Future variable that is written to once and read multiple times	•	22.5
3	Account	Concurrent accounts exchanging debit and credit instructions	•	19.5
4	AccountF	Concurrent accounts where debit instructions are effected via futures	•	33.5
5	Master-Worker	Master-worker parallel network	•	29.0
6	Session Types	Session-typed communicating actors using one arbiter	○	75.5
Selected micro-benchmarks adapted from Imam and Sarkar [34], based on Neykova and Yoshida [42]				
7	Ping Pong	Process pair exchanging k ping and pong messages	•	24.6
8	Thread Ring	Ring network where actors cyclically relay one token with counter k	○	37.2
9	Counter	One actor sending messages to a second that sums the count, k	○	29.8
10	K-Fork	Fork-join pattern where a central actor delegates k requests to workers	•	7.1
11	Fibonacci	Fibonacci server delegating terms $(k-1)$ and $(k-2)$ to parallel actors	•	27.1
12	Big	Peer-to-peer network where actors exchange k messages randomly	○	62.8
13	Philosopher	Dining philosophers problem	○	57.1
14	Smokers	Centralised network where one arbiter allocates k messages to actors	○	31.3
15	Log Map	Computes the term $x_{k+1} = r \cdot x_k(1 - x_k)$ by delegating to parallel actors	○	57.9
16	Transaction	Request-reply actor communication initiated by a central teller actor	○	46.7

- Can encode all examples from original Mailbox Calculus paper
- Can also check Savina benchmarks
- All typecheck in <100ms

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
- Constraint-based co-contextual typing algorithm
- More sophisticated alias analysis
- Tool integration (Erlang, Elixir…)
- Other many-writer paradigms (Publish-subscribe? Typestate?)

Our sponsors make OPLSS possible.

Diamond Sponsor

Jane Street

https://www.janestreet.com/

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

SIGPLAN

https://www.sigplan.org/

Silver Sponsors

Bonus Slides

Mailbox types	J, K	::=	$!E \mid ?E$
Mailbox patterns	E, F	::=	$\mathbb{0} \mid \mathbb{1} \mid \mathtt{m} \mid E \oplus F$
		$\mid$	$E \odot F \mid \star E$

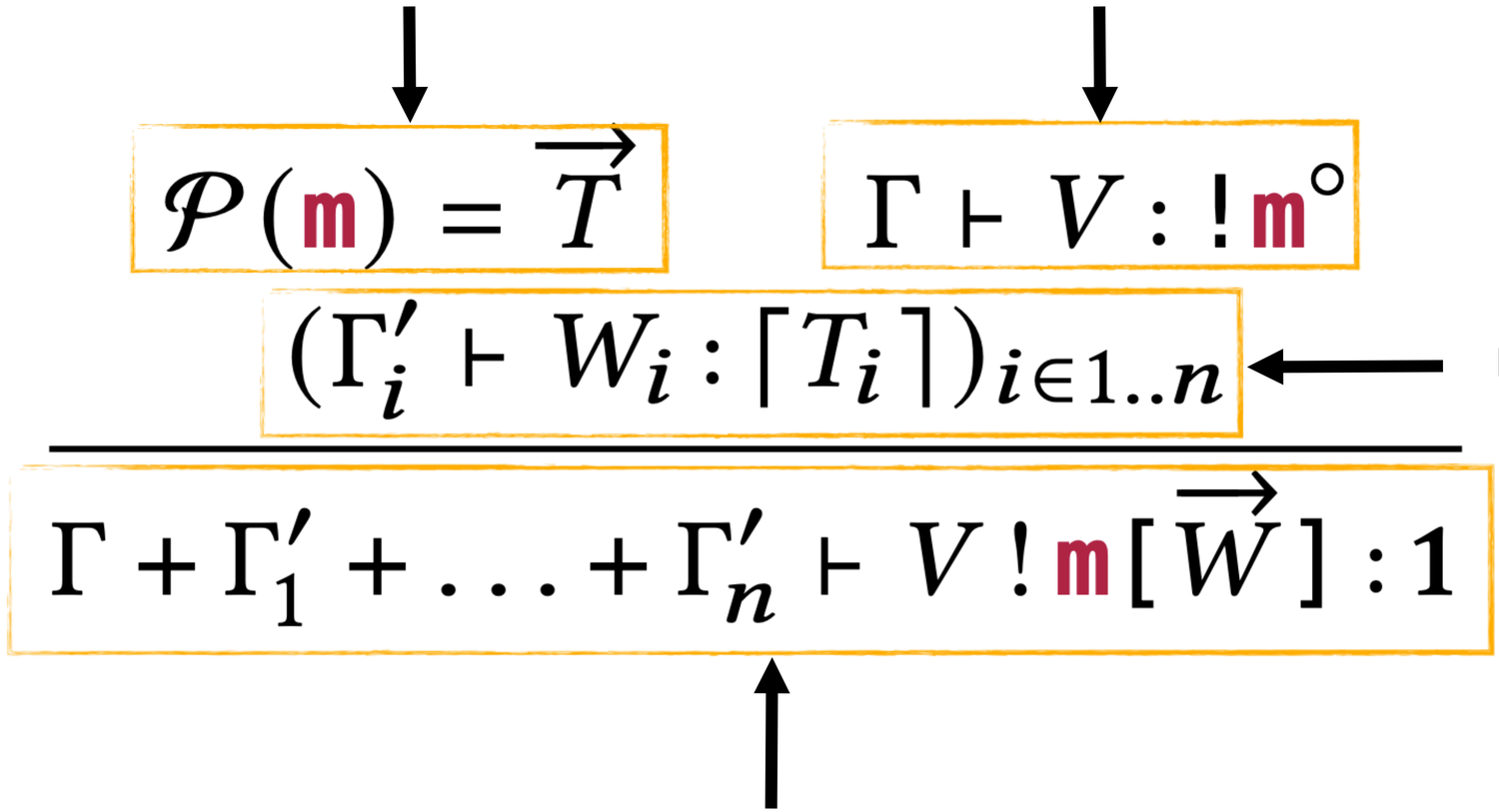
Base types	C	::=	$\mathbf{1} \mid \mathsf{Int} \mid \mathsf{String} \mid \cdots$
Types	T, U	::=	$C \mid J$
Usage annotations	η	::=	$\circ \mid \bullet$
Usage-annotated types	A, B	::=	$C \mid J^\eta$

Variables	x, y, z		
Definition names	f		
Definitions	D	::=	$\textbf{def}\ f(\overrightarrow{x : A}) : B\ \{M\}$
Values	V, W	::=	$x \mid c$
Terms	L, M, N	::=	$V \mid \textbf{let}\ x : T = M\ \textbf{in}\ N \mid f(\overrightarrow{V})$
		$\mid$	$\textbf{spawn}\ M \mid \textbf{new} \mid V\,!\,\mathtt{m}[\overrightarrow{W}] \mid \textbf{guard}\ V : E\ \{\overrightarrow{G}\}$
Guards	G	::=	$\textbf{fail} \mid \textbf{free} \mapsto M \mid \textbf{receive}\ \mathtt{m}[\overrightarrow{x}]\ \textbf{from}\ y \mapsto M$
Type environments	Γ	::=	$\cdot \mid \Gamma, x : A$

Selected Typing Rule (Send)

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

Target must have 2nd-class mailbox type !**m**



Payload types must match (2nd class)

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

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

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

Note: Type annotation **optional**

Selected Typing Rules (New and Spawn)

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

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

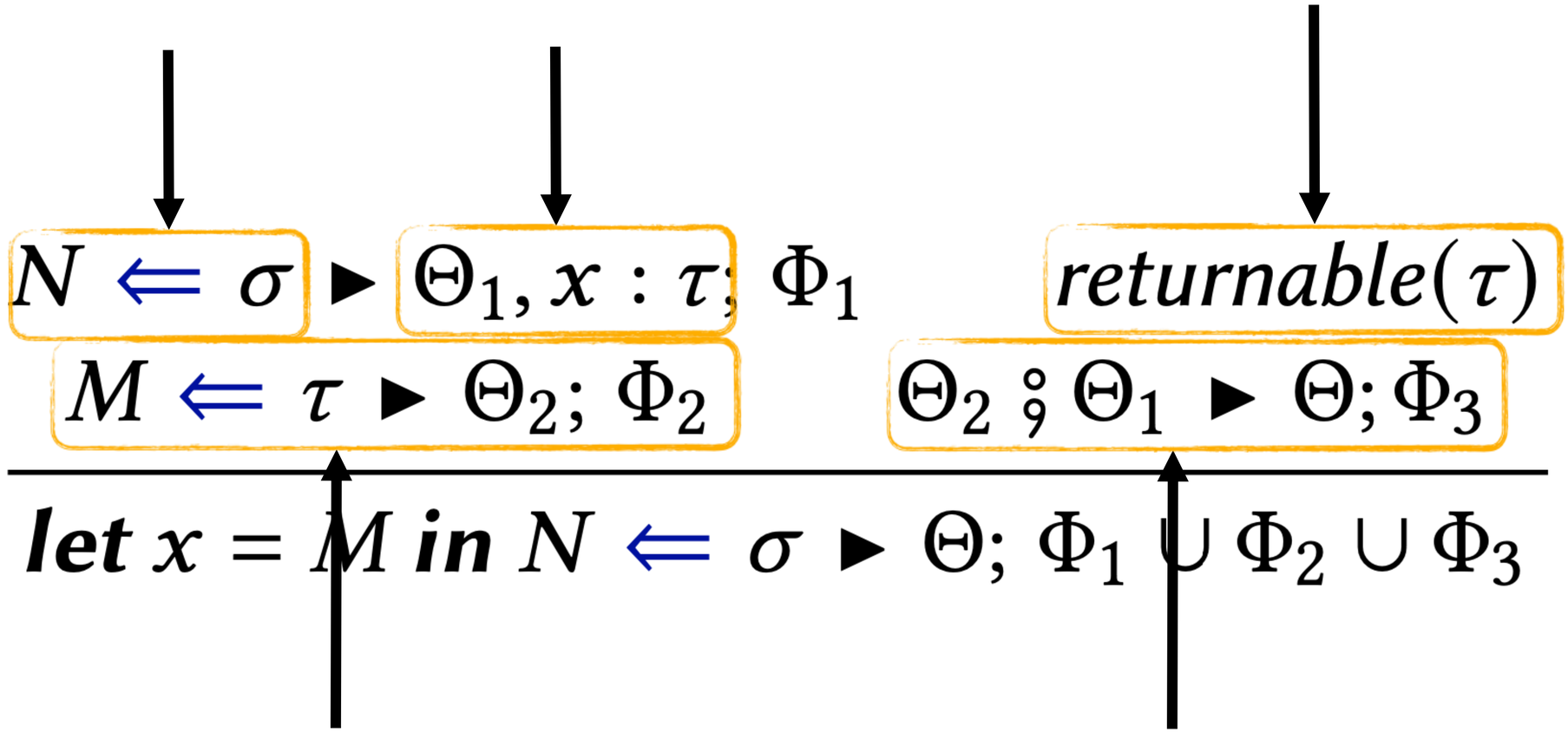
$$\frac{\Gamma \vdash M : \mathbf{1}}{\lceil \Gamma \rceil \vdash \mathbf{spawn}\ M : \mathbf{1}}$$

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

Check that the body has type σ

Inspect produced type environment, see that x has type τ

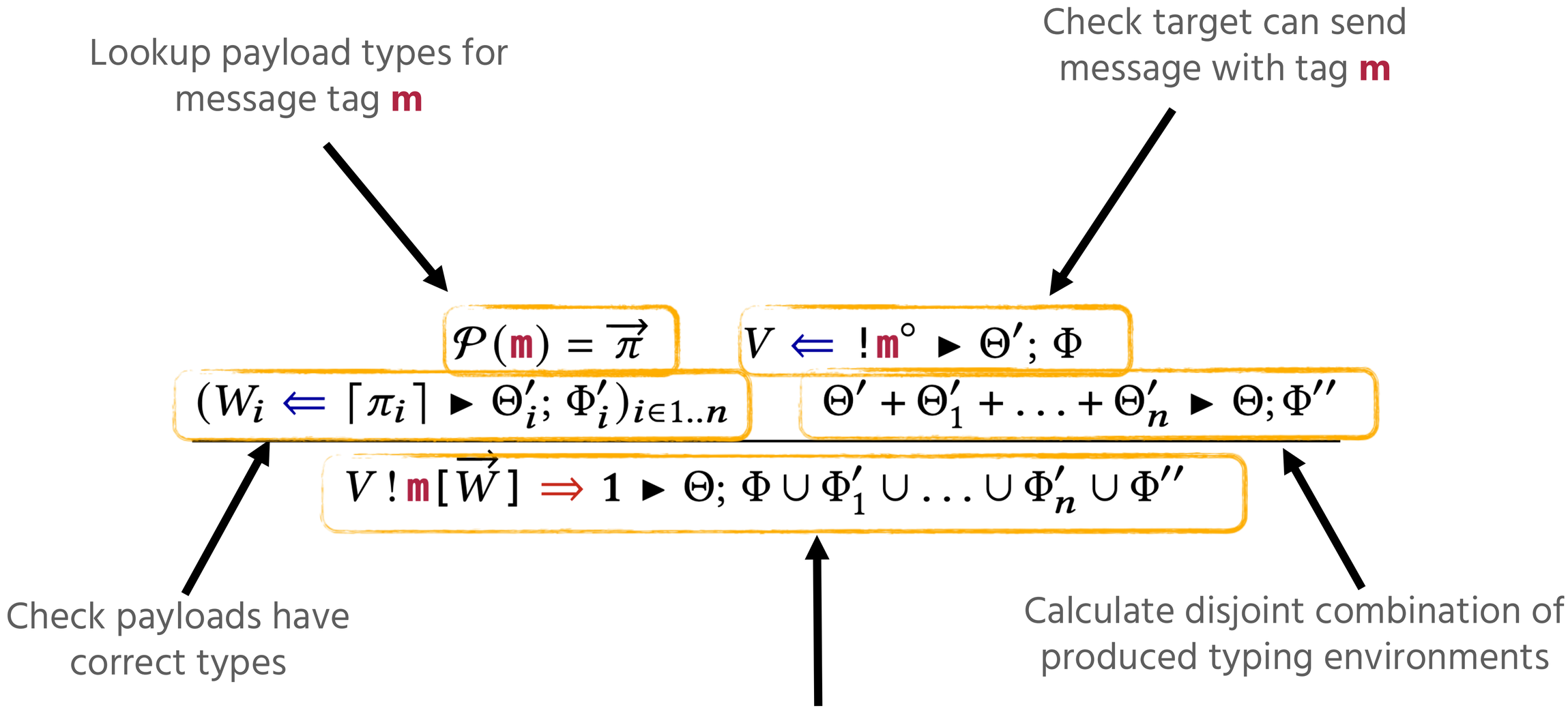
Ensure that τ is returnable



Check that M has type τ

Calculate algorithmic sequencing of environments

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


Lookup payload types for message tag m
Check target can send message with tag m
$\mathcal{P}(\mathtt{m}) = \overrightarrow{\pi}$
$V \Leftarrow \,!\mathtt{m}^{\circ} \blacktriangleright \Theta'; \Phi$
$(W_i \Leftarrow \lceil \pi_i \rceil \blacktriangleright \Theta'_i; \Phi'_i)_{i \in 1..n}$
$\Theta' + \Theta'_1 + \ldots + \Theta'_n \blacktriangleright \Theta; \Phi''$
$V\,!\mathtt{m}[\overrightarrow{W}] \Rightarrow \mathbf{1} \blacktriangleright \Theta;\ \Phi \cup \Phi'_1 \cup \ldots \cup \Phi'_n \cup \Phi''$
Check payloads have correct types
Calculate disjoint combination of produced typing environments
Synthesise unit type for the send term, producing combined environment and the union of all constraint sets

Metatheory (Declarative)

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

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

Nontrivial: frame stack representation; also requires extensive reasoning about contexts and quasi-linearity

Metatheory (Algorithmic)

Theorem (Algorithmic Soundness)

- If Ξ is a covering solution for $\mathrm{M} \Leftarrow_P \tau \blacktriangleright \Theta; \Phi$, then $\Xi(\Theta) \vdash_{\Xi(P)} M : \Xi(\tau)$
- If Ξ is a covering solution for $\mathrm{M} \Rightarrow_P \tau \blacktriangleright \Theta; \Phi$, then $\Xi(\Theta) \vdash_{\Xi(P)} M : \Xi(\tau)$

Theorem (Algorithmic Completeness)

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

Our sponsors make OPLSS possible.

Diamond Sponsor

Jane Street

https://www.janestreet.com/

Platinum Sponsors

National Science Foundation
WHERE DISCOVERIES BEGIN

https://www.nsf.gov/

SIGPLAN

https://www.sigplan.org/

Silver Sponsors