Mixing Metaphors: Actors as Channels and Channels as Actors

Abstract

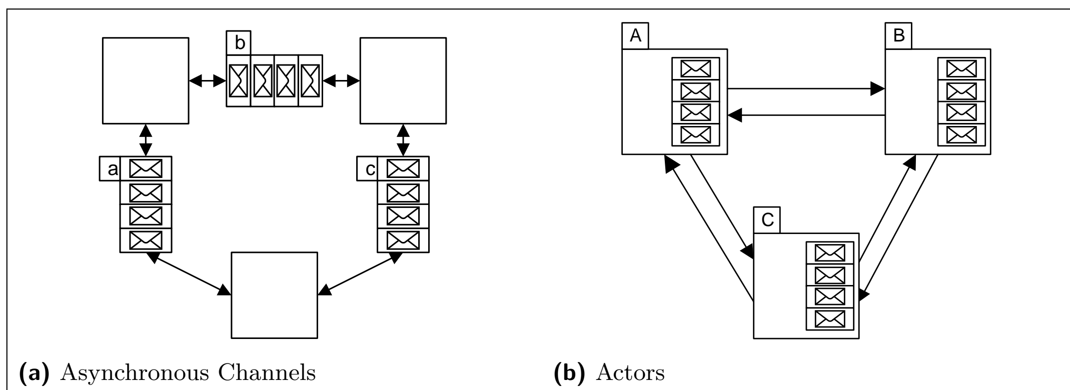
Channel- and actor-based programming languages are both used in practice, but the two are often confused. Languages such as Go provide anonymous processes which communicate using typed buffers—known as channels—while languages such as Erlang provide addressable processes each with a single incoming message queue—known as actors. The lack of a common representation makes it difficult to reason about the translations that exist in the folklore. We define a calculus λ_{ch} for typed asynchronous channels, and a calculus λ_{act} for typed actors. We define translations from λ_{act} into λ_{ch} and λ_{ch} into λ_{act} and prove that both translations are type- and semantics-preserving. We show that our approach accounts for synchronisation and selective receive in actor systems and discuss future extensions to support guarded choice and behavioural types.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

When comparing channels (as used by Go) and actors (as used by Erlang), one runs into an immediate mixing of metaphors. The words themselves do not refer to comparable entities!

In languages such as Go, anonymous processes pass messages via named channels, whereas in languages such as Erlang, named processes accept messages from an associated mailbox. A channel is a buffer, whereas an actor is a process. We should really be comparing named processes (actors) with anonymous processes, and buffers tied to a particular process (mailboxes) with buffers that can link any process to any process (channels). Nonetheless, we will stick with the popular names, even if it is as inapposite as comparing TV channels with TV actors.



(a) Asynchronous Channels **(b)** Actors

Figure 1 Channels and Actors

Figure 1 compares channels with actors. On the left, three anonymous processes communicate via channels named a, b, c. On the right, three processes named A, B, C send messages to each others' associated mailboxes. A common misunderstanding is that channels are synchronous but actors are asynchronous [33], however while asynchrony *is* required by actor systems, channels may be either synchronous or asynchronous; to ease comparison, we consider asynchronous channels. A more significant difference is that each actor has a single buffer, its mailbox, which can be read only by that actor, whereas channels are free-floating buffers that can be read by any process with a reference to the channel.

licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Channel-based languages such as Go enjoy a firm basis in process calculi such as CSP [21] and the π-calculus [32]. It is easy to type channels, either with simple types (see [40], p. 231) or more complex systems such as session types [15, 22, 23]. Actor-based languages such as Erlang are seen by many as the "gold standard" for distributed computing due to their support for fault tolerance through supervision hierarchies [5, 7].

Both models are popular with developers, with channel-based languages and frameworks such as Go, Concurrent ML [39], and Hopac [24]; and actor-based languages and frameworks such as Erlang, Elixir, and Akka.

There is often confusion over the differences between channels and actors. For example, two questions about this topic are:

> "If I wanted to port a Go library that uses Goroutines, would Scala be a good choice because its inbox/[A]kka framework is similar in nature to coroutines?" [26], and

> "I don't know anything about [the] actor pattern however I do know goroutines and channels in Go. How are [the] two related to each other?" [25]

The success of actor-based languages is largely due to their support for *supervision hierarchies*: processes are arranged in trees, where *supervisor* processes restart child processes should they fail. Projects such as Proto Actor [38] emulate actor-style programming in a channel-based language in an attempt to gain some of the benefits. Hopac [24] is a channel-based library for F#, based on Concurrent ML [39]. The documentation [1] contains a comparison with actors, including an implementation of a simple actor-based communication model using Hopac-style channels, as well as an implementation of Hopac-style channels using an actor-based communication model. By comparing the two, this paper provides a formal model for the implementation technique used by Proto Actor, and a formal model for an asynchronous variant of the translation from channels into actors as specified by the Hopac documentation.

Putting Practice into Theory. We seek to characterise the core features of channel- and actor-based models of concurrent programming, and distil them into minimal concurrent λ-calculi. In doing so, we:

- Obtain concise and expressive core calculi, which can be used as a basis to explore more advanced features such as behavioural typing, and;
- Make the existing folklore about the two models explicit, gaining formal guarantees about the correctness of translations. In turn, we give a formal grounding to implementations based on the translations, such as Proto Actor.

Our common framework is that of a concurrent λ-calculus: that is, a λ-calculus with a standard term language equipped with primitives for communication and concurrency, as well as a language of *configurations* to model concurrent behaviour. We choose a λ-calculus rather than a process calculus as our starting point because we are ultimately interested in actual programming languages, and in particular functional programming languages.

While actor-based languages must be asynchronous by design, channels may be either synchronous (requiring a rendezvous between sender and receiver) or asynchronous (where sending always happens immediately). We base λ_{ch} on asynchronous channels since actors are naturally asynchronous, and since it is possible to emulate asynchronous channels using synchronous channels [39]. By working in the asynchronous setting, we can concentrate on the more fundamental differences between the two models.

A central difference between the two models is depicted in Figure 2. Figure 2a shows the communication patterns allowed by a single (full-duplex) channel: each process P_i can

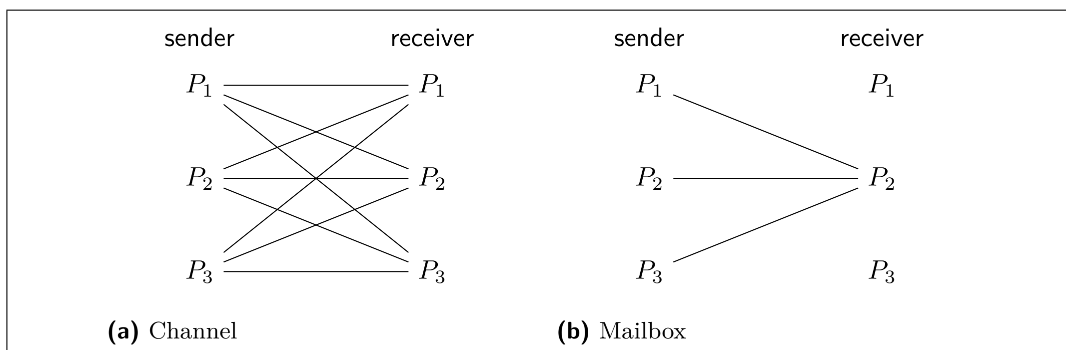


Figure 2 Mailboxes as pinned channels

use the channel to communicate with every other process. Conversely, Figure 2b shows the communication patterns allowed by a mailbox associated with process P_2: while any process can send to the mailbox, only P_2 can read from it. Viewed this way, it is apparent that the restrictions imposed on the communication behaviour of actors are exactly those captured by Merro and Sangiorgi's localised π-calculus [31].

Readers familiar with actor-based programming may be wondering at this point whether such a characterisation is too crude, as it does not take into account the possibility of processing messages out-of-order. Fear not—we show in Section 7 that our basic actor calculus can in fact simulate this functionality. The key idea is to simulate a mailbox supporting selective receive by an actual mailbox augmented with a secondary queue of values that have been received but not yet matched.

Outline and Contributions.

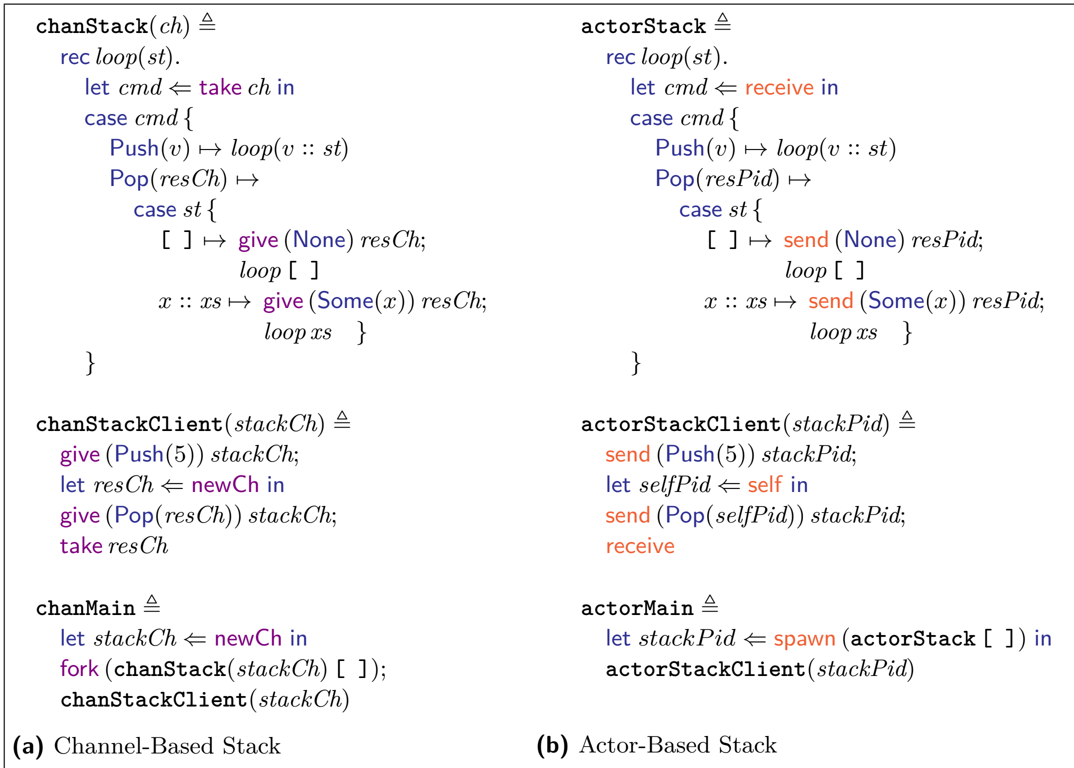
In §2 we present side-by-side implementations of a concurrent stack using channels and using actors. The main contributions of this paper are as follows.

- We define a calculus λ_{ch} with typed asynchronous channels (§3), and a calculus λ_{act} with type-parameterised actors (§4), by extending the simply-typed λ-calculus with communication primitives specialised to each model. We give a type system and operational semantics for each calculus, and precisely characterise the notion of progress that each calculus enjoys.
- We define a simple translation from λ_{act} into λ_{ch}, prove that the translation is type-preserving, and prove that λ_{ch} can simulate λ_{act} (§5).
- We define a more involved translation from λ_{ch} into λ_{act}, again proving that the translation is type-preserving, and that λ_{act} can simulate λ_{ch} (§6).
- We introduce an extension of λ_{act} to support synchronous communication calls, and show how this can simplify the translation from λ_{ch} into λ_{act} (§7.1).
- We introduce an extension of λ_{act} to support Erlang-style selective receive, and prove that it can be simulated by λ_{act} without selective receive (§7.2).
- We outline extension of λ_{ch} with input-guarded nondeterministic choice (§7.3) and consider how λ_{act} might be extended with behavioural types (§7.4).

In §8 we discuss related work and §9 concludes.

2 Channels and Actors Side-by-Side

Let us consider the example of a concurrent stack. A concurrent stack carrying values of type A can receive a command to push a value onto the top of the stack, or to pop a value from



(a) Channel-Based Stack **(b)** Actor-Based Stack

Figure 3 Concurrent Stacks using Channels and Actors

the stack and return it to the process making the request. Assuming a standard encoding of algebraic datatypes using binary sums, we define a type Operation(A) = Push(A) | Pop(B) (where B = ChanRef(A) for channels, and ActorRef(A) for actors) to describe operations on the stack, and Option(A) = Some(A) | None to handle the possibility of popping from an empty stack.

Figure 3 shows the stack implemented using channels (Figure 3a) and using actors (Figure 3b). Each implementation uses a common core language based on the simply-typed λ-calculus extended with recursion, lists, and sums.

At first glance, the two stack implementations seem remarkably similar. Each:

1. Waits for a command
2. Case splits on the command, and either:
   - Pushes a value onto the top of the stack, or;
   - Takes the value from the head of the stack and returns it in a response message
3. Loops with an updated state.

The main difference is that **chanStack** is parameterised over a channel *ch*, and retrieves a value from the channel using take *ch*. Conversely, **actorStack** retrieves a value from its mailbox using the nullary primitive receive.

Let us now consider functions which interact with the stacks. The **chanStackClient** function sends commands over the *stackCh* channel, and begins by pushing 5 onto the stack. Next, the function creates a channel *resCh* to be used to receive the result and sends this in a request, before retrieving the result from the result channel using take. In contrast, **actorStackClient** performs a similar set of steps, but sends its process ID (retrieved using

$\texttt{chanClient2}(\mathit{intStackCh}, \mathit{stringStackCh}) \triangleq$	$\texttt{actorClient2}(\mathit{intStackPid}, \mathit{stringStackPid}) \triangleq$
$\mathsf{let}\ \mathit{intResCh} \Leftarrow \mathsf{newCh}\ \mathsf{in}$	$\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}$
$\mathsf{let}\ \mathit{strResCh} \Leftarrow \mathsf{newCh}\ \mathsf{in}$	$\mathsf{send}\ (\mathsf{Pop}(\mathit{selfPid}))\ \mathit{intStackPid};$
$\mathsf{give}\ (\mathsf{Pop}(\mathit{intResCh}))\ \mathit{intStackCh};$	$\mathsf{let}\ \mathit{res1} \Leftarrow \mathsf{receive}\ \mathsf{in}$
$\mathsf{let}\ \mathit{res1} \Leftarrow \mathsf{take}\ \mathit{intResCh}\ \mathsf{in}$	$\mathsf{send}\ (\mathsf{Pop}(\mathit{selfPid}))\ \mathit{stringStackPid};$
$\mathsf{give}\ (\mathsf{Pop}(\mathit{strResCh}))\ \mathit{stringStackCh};$	$\mathsf{let}\ \mathit{res2} \Leftarrow \mathsf{receive}\ \mathsf{in}$
$\mathsf{let}\ \mathit{res2} \Leftarrow \mathsf{take}\ \mathit{strResCh}\ \mathsf{in}$	$(\mathit{res1}, \mathit{res2})$
$(\mathit{res1}, \mathit{res2})$	

Figure 4 Clients Interacting with Multiple Stacks

self) in the request instead of creating a new channel; the result is then retrieved from the mailbox using receive.

Type Pollution. The differences become more prominent when we consider clients which interact with multiple stacks containing different types of values, as shown in Figure 4. Here, **chanStackClient2** creates new result channels for integers and strings, sends requests for the results, and creates a pair of type $(\mathsf{Option}(\mathsf{Int}) \times \mathsf{Option}(\mathsf{String}))$. The **actorStackClient2** function attempts to do something similar, but cannot create separate result channels. Consequently, the actor must be able to handle messages either of type $\mathsf{Option}(\mathsf{Int})$ *or* type $\mathsf{Option}(\mathsf{String})$, meaning that the final pair has type $(\mathsf{Option}(\mathsf{Int}) + \mathsf{Option}(\mathsf{String})) \times (\mathsf{Option}(\mathsf{Int}) + \mathsf{Option}(\mathsf{String}))$.

Additionally, it is necessary to modify **actorStack** to use the correct injection into the actor type when sending the result; for example an integer stack would have to send a value $\mathsf{inl}\ (\mathsf{Some}(5))$ instead of simply $\mathsf{Some}(5)$. The requirement of knowing all message types received by another actor is known as the *type pollution* problem, which can be addressed neatly through the use of subtyping [19], or synchronisation abstractions such as futures [10].

3 λ_{ch}: A Concurrent λ-calculus for Channels

In this section we introduce λ_{ch}, a concurrent λ-calculus extended with asynchronous channels. To concentrate on the core differences between channel- and actor-style communication, we begin with minimal calculi; note that these do not contain all features (such as lists, sums, and recursion) needed to express the examples in Section 2.

3.1 Syntax and Typing of Terms

Figure 5 gives the syntax and typing rules of λ_{ch}, a lambda calculus based on fine-grain call-by-value [29]: terms are partitioned into values and computations. Key to this formulation are two constructs: $\mathsf{return}\, V$ represents a computation that has completed, whereas $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$ evaluates M to $\mathsf{return}\, V$, substituting V for x in M. Fine-grain call-by-value is convenient because it makes evaluation-order completely explicit and (unlike A-normal form, for instance) is closed under reduction.

Types consist of the unit type $\mathbf{1}$, function types $A \to B$, and channel reference types $\mathsf{ChanRef}(A)$ which can be used to communicate along a channel of type A. We let α range over variables x and run time names a. We write $\mathsf{let}\, x = V\, \mathsf{in}\, M$ for $(\lambda x.M)\, V$ and $M; N$ for $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$, where x is fresh.

Communication and Concurrency for Channels. The $\mathsf{give}\, V\, W$ operation sends value V along channel W, while $\mathsf{take}\, V$ retrieves a value from a channel V. Assuming an extension of the language with integers and arithmetic operators, we can define a function $\mathbf{neg}(c)$ which

Syntax

Types	$A, B ::= \mathbf{1} \mid A \to B \mid \mathsf{ChanRef}(A)$
Variables and names	$\alpha ::= x \mid a$
Values	$V, W ::= \alpha \mid \lambda x.M \mid ()$
Computations	$L, M, N ::= V\,W$
	$\mid\ \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V$
	$\mid\ \mathsf{fork}\ M \mid \mathsf{give}\ V\ W \mid \mathsf{take}\ V \mid \mathsf{newCh}$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{App}\ \frac{\Gamma \vdash V : A \to B \quad \Gamma \vdash W : A}{\Gamma \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B} \qquad \textsc{Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{return}\ V : A}$$

$$\textsc{Give}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{give}\ V\ W : \mathbf{1}} \qquad \textsc{Take}\ \frac{\Gamma \vdash V : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathsf{take}\ V : A} \qquad \textsc{Fork}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathsf{fork}\ M : \mathbf{1}} \qquad \textsc{NewCh}\ \frac{}{\Gamma \vdash \mathsf{newCh} : \mathsf{ChanRef}(A)}$$

Figure 5 Syntax and typing rules for λ_{ch} terms and values

receives a number n along channel c and replies with the negation of n as follows:

$$\mathbf{neg}(c) \triangleq \mathsf{let}\ n \Leftarrow \mathsf{take}\ c\ \mathsf{in}\ \mathsf{let}\ negN \Leftarrow (-n)\ \mathsf{in}\ \mathsf{give}\ negN\ c$$

The operation newCh creates a new channel. The operation fork M spawns a new process that performs computation M. Firstly, note that fork returns the unit value; the spawned process is anonymous and therefore it is not possible to interact with it directly. Secondly, note that channel creation is completely decoupled from process creation, meaning that a process can have access to multiple channels.

3.2 Operational Semantics

Configurations. The concurrent behaviour of λ_{ch} is given by a nondeterministic reduction relation on *configurations*, ranged over by $\mathcal{C}$ and $\mathcal{D}$ (Figure 6). Configurations consist of parallel composition ($\mathcal{C} \parallel \mathcal{D}$), restrictions ($(\nu a)\mathcal{C}$), computations ($M$), and buffers ($a(\overrightarrow{V})$, where $\overrightarrow{V} = V_1 \cdot \ldots \cdot V_n$).

Evaluation Contexts. Reduction is defined in terms of evaluation contexts E, which are simplified due to fine-grain call-by-value. We also define configuration contexts, allowing reduction modulo parallel composition and name restriction.

Reduction. Figure 7 shows the reduction rules for λ_{ch}. Reduction is defined as a deterministic reduction on terms ($\longrightarrow_{\mathsf{M}}$) and a nondeterministic reduction relation on configurations ($\longrightarrow$). Reduction on configurations is defined modulo structural congruence rules which capture commutativity and associativity of parallel composition, scope extrusion, and that structural congruence extends to configuration contexts.

Typing of Configurations. We wish to ensure well-formedness of configurations—namely that buffers are well-scoped and contain values of the correct type—so it is convenient to

Syntax of evaluation contexts and configurations

Evaluation contexts	$E ::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid a(\overrightarrow{V}) \mid M$
Configuration contexts	$G ::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Chan}\ \frac{\Gamma, a : \mathsf{ChanRef}(A); \Delta, a{:}A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}} \qquad \textsc{Buf}\ \frac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})} \qquad \textsc{Term}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma; \cdot \vdash M}$$

Figure 6 λ_{ch} configurations and evaluation contexts

define typing rules on configurations as shown in Figure 6. The judgement $\Gamma; \Delta \vdash \mathcal{C}$ states that under environments Γ and Δ, $\mathcal{C}$ is well-typed; Γ is a typing environment for terms, whereas Δ is a linear typing environment for configurations, mapping names a to channel types A. Linearity in Δ is purely a technical device to ensure that under a name restriction $(\nu a)\mathcal{C}$, that $\mathcal{C}$ contains exactly one buffer with name a; this should not be confused with channel *references* which are contained in Γ and are unrestricted. Note that Chan extends both Γ *and* Δ, adding a *reference* into Γ and the *capability* to type a buffer into Δ. Par states that two configurations are typeable if they are each typeable under disjoint linear environments, and Buf states that under a term environment Γ and a singleton linear environment $a : A$, it is possible to type a buffer $a(\overrightarrow{V})$ if $\Gamma \vdash V_i : A$ for all $V_i \in \overrightarrow{V}$. As an example, $(\nu a)(a(\overrightarrow{V}))$ is well-typed, but $(\nu a)(a(\overrightarrow{V}) \parallel a(\overrightarrow{W}))$ and $(\nu a)(\mathsf{return}\ ())$ are not.

Relation Notation. Given a relation R, we write R^+ for its transitive closure, and R^* for its reflexive, transitive closure.

Properties of the Term Language. Reduction on terms is standard. It preserves typing and purely-functional terms enjoy progress. We omit proofs in the body of the paper which are mainly straightforward inductions; selected full proofs can be found in Appendix B.

▶ **Lemma 1** (Preservation (λ_{ch} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_{\mathsf{M}} M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 2** (Progress (λ_{ch} terms)).
Assume Γ is either empty or only contains entries of the form $a_i : \mathsf{ChanRef}(A_i)$.
If $\Gamma \vdash M : A$, then either:

1. *$M = \mathsf{return}\ V$ for some value V*
2. *M can be written $E[M']$, where M' is a communication or concurrency primitive (i.e. $\mathsf{give}\ V\ W, \mathsf{take}\ V, \mathsf{fork}\ M$, or newCh)*
3. *There exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$*

Reduction on Configurations. Concurrency and communication is captured by reduction on configurations. The Give rule reduces $\mathsf{give}\ W\ a$ in parallel with a buffer $a(\overrightarrow{V})$ by adding the value W onto the end of the buffer. The Take rule reduces $\mathsf{take}\ a$ in parallel with a non-empty buffer by returning the first value in the buffer. The Fork rule reduces $\mathsf{fork}\ M$ by spawning a new thread M in parallel with the parent process. The NewCh rule reduces newCh by creating an empty buffer and returning a fresh name for that buffer.

Typeability of configurations is preserved by structural congruence, and reduction preserves the typeability of configurations.

▶ **Lemma 3.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$ for some configuration $\mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 4** (Preservation (λ_{ch} configurations)). *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

Reduction on terms

$$
\begin{array}{rcll}
(\lambda x.M)\,V & \longrightarrow_{\mathsf{M}} & M\{V/x\} & \\
\mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M & \longrightarrow_{\mathsf{M}} & M\{V/x\} & \\
E[M_1] & \longrightarrow_{\mathsf{M}} & E[M_2] & (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)
\end{array}
$$

Structural congruence

$$
\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel C \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel D) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})
$$

$$
G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}
$$

Reduction on configurations

$$
\begin{array}{llcll}
\textsc{Give} & E[\mathsf{give}\, W\, a] \parallel a(\overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\ ()] \parallel a(\overrightarrow{V} \cdot W) & \\
\textsc{Take} & E[\mathsf{take}\, a] \parallel a(W \cdot \overrightarrow{V}) & \longrightarrow & E[\mathsf{return}\, W] \parallel a(\overrightarrow{V}) & \\
\textsc{Fork} & E[\mathsf{fork}\, M] & \longrightarrow & E[\mathsf{return}\ ()] \parallel M & \\
\textsc{NewCh} & E[\mathsf{newCh}] & \longrightarrow & (\nu a)(E[\mathsf{return}\, a] \parallel a(\epsilon)) & (a \text{ is a fresh name}) \\
\textsc{LiftM} & G[M_1] & \longrightarrow & G[M_2] \quad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2) & \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] \quad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) &
\end{array}
$$

Figure 7 Reduction on λ_{ch} terms and configurations

3.3 Progress and Canonical Forms

While it is possible to prove deadlock-freedom in systems with more discerning type systems based on linear logic (such as those of Wadler [42], and Lindley and Morris [30]) or those using channel priorities (for example, the calculus of Padovani and Novara [36]), more liberal calculi such as λ_{ch} and λ_{act} allow deadlocked configurations. We thus define a form of progress which does not preclude deadlock; to help with proving a progress result, it is useful to consider the notion of a *canonical form* in order to allow us to reason about the configuration as a whole.

▶ **Definition 5** (Canonical form (λ_{ch})). A configuration $\mathcal{C}$ is in *canonical form* if it can be written $(\nu a_1) \ldots (\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots \parallel a_n(\overrightarrow{V_n}))$.

The following lemma states that well-typed open configurations can be written in a form similar to canonical form, but without bindings for names already in the environment. An immediate corollary is that well-typed closed configurations can always be written in a canonical form.

▶ **Lemma 6.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *with* $\Delta = a_1 : A_1, \ldots, a_k : A_k$*, then there exists a* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}' = (\nu a_{k+1}) \ldots (\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots \parallel a_n(\overrightarrow{V_n}))$.

▶ **Corollary 7.** *If* $\cdot; \cdot \vdash \mathcal{C}$*, then there exists some* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}'$ *is in canonical form.*

Armed with a canonical form, we can now capture precisely the intuition that the only situation in which a well-typed closed configuration $\mathcal{C}$ cannot reduce further is if all threads are either blocked or fully evaluated. Consider a *leaf configuration* to be a configuration without subconfigurations: in this case, either a term or a buffer.

▶ **Theorem 8** (Weak progress (λ_{ch} configurations)).
Let $\cdot; \cdot \vdash \mathcal{C}$*,* $\mathcal{C} \not\longrightarrow$*, and let* $\mathcal{C}' = (\nu a_1) \ldots (\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots a_n(\overrightarrow{V_n}))$ *be a canonical form of* $\mathcal{C}$*. Then every leaf of* $\mathcal{C}$ *is either:*

1. *A buffer* $a_i(\overrightarrow{V_i})$*;*
2. *A fully-reduced term of the form* return V*, or;*
3. *A term of the form* $E[\mathsf{take}\, a_i]$*, where* $\overrightarrow{V_i} = \epsilon$*.*

4 λ_{act}: A Concurrent λ-calculus for Actors

In this section, we introduce λ_{act}, a core language providing actor-like concurrency.

Actors were originally introduced by Hewitt et al. [20] as a formalism for artificial intelligence, where an actor is an entity endowed with an unforgeable address known as a process ID, and a single incoming message queue known as a mailbox. There are many variations of actor-based languages (the work of De Koster et al. [11] provides a detailed taxonomy—our calculus firmly falls into the category of *process-based* actors), but the main common feature is that each provide lightweight processes which are associated with a mailbox. We follow the model of Erlang by providing an explicit `receive` operation to allow an actor to retrieve a message from its mailbox, as opposed to making an event loop implicit.

While it is common to parameterise channels over types, parameterising actors over types is more recent, in particular due to type pollution. Type-parameterised actors were introduced by He [18] and He et al. [19], and more recently implemented in libraries such as Akka Typed [4] and Typed Actors [41]. Indeed, Akka Typed hopes to replace untyped Akka actors, so it seems useful to consider a typed calculus. A key difference between λ_{ch} and λ_{act} is that receive (unlike take) is a nullary operation to receive a value from the actor's mailbox. Consequently, it is necessary to use a simple *type-and-effect system* (as inspired by Gifford and Lucassen [16]) to type terms with respect to the mailbox type of the enclosing actor.

λ_{act} is designed to be as minimal as possible, in order to concentrate on the core differences with channel-based programming. Consequently, we treat the mailbox as a FIFO queue as opposed to considering actor behaviours or selective, out-of-order reception of messages. This is orthogonal to the core model of communication—we show in §7.2 that after extending the term language of λ_{act}, it is possible to encode selective receive in the base calculus.

4.1 Syntax and Typing of Terms

Figure 8 shows the syntax and typing rules for λ_{act}. As with λ_{ch}, α ranges over variables and names. ActorRef(A) is an *actor reference* or process ID, and allows messages to be sent to an actor. As for communication and concurrency primitives, spawn M spawns a new actor to evaluate a computation M; send $V\, W$ sends a value V to an actor referred to by reference W; receive receives a value from the actor's mailbox; and self returns an actor's own process ID.

Function arrows $A \to^C B$ are annotated with a type C which denotes the type of the mailbox of the actor evaluating the term. As an example, consider a function which multiplies a received number by a given value:

$$\texttt{recvAndMult} \triangleq \lambda n.\mathsf{let}\ x \Leftarrow \mathsf{receive}\ \mathsf{in}\ (x \times n)$$

Such a function would have type $Int \to^{Int} Int$, and as an example would not be typeable for an actor that could only receive strings. Again, we work in the setting of fine-grain call-by-value; the distinction between values and computations is helpful when reasoning about the metatheory. We have two typing judgements: the standard judgement on values $\Gamma \vdash V : A$, and a judgement $\Gamma \mid B \vdash M : A$ which states that a term M has type A under typing context Γ, and can receive values of type B. The typing of receive and self depends on the type of the actor's mailbox.

Syntax

Types	$A, B, C ::= \mathbf{1} \mid A \to^C B \mid \mathsf{ActorRef}(A)$
Variables and names	$\alpha ::= x \mid a$
Values	$V, W ::= \alpha \mid \lambda x.M \mid ()$
Computations	$L, M, N ::= V\,W$
	$\mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid \mathsf{return}\ V$
	$\mid \mathsf{spawn}\ M \mid \mathsf{send}\ V\,W \mid \mathsf{receive} \mid \mathsf{self}$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Var}\ \frac{\alpha : A \in \Gamma}{\Gamma \vdash \alpha : A} \qquad \textsc{Abs}\ \frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^C B} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \mid B \vdash M : A}$

$$\textsc{App}\ \frac{\Gamma \vdash V : A \to^C B \quad \Gamma \vdash W : A}{\Gamma \mid C \vdash V\,W : B} \qquad \textsc{EffLet}\ \frac{\Gamma \mid C \vdash M : A \quad \Gamma, x : A \mid C \vdash N : B}{\Gamma \mid C \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B}$$

$$\textsc{EffReturn}\ \frac{\Gamma \vdash V : A}{\Gamma \mid C \vdash \mathsf{return}\ V : A} \qquad \textsc{Send}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ActorRef}(A)}{\Gamma \mid C \vdash \mathsf{send}\ V\,W : \mathbf{1}}$$

$$\textsc{Recv}\ \frac{}{\Gamma \mid A \vdash \mathsf{receive} : A} \qquad \textsc{Spawn}\ \frac{\Gamma \mid A \vdash M : \mathbf{1}}{\Gamma \mid C \vdash \mathsf{spawn}\ M : \mathsf{ActorRef}(A)} \qquad \textsc{Self}\ \frac{}{\Gamma \mid A \vdash \mathsf{self} : \mathsf{ActorRef}(A)}$$

Figure 8 Syntax and typing rules for λ_{act}

4.2 Operational Semantics

Figure 9 shows the syntax of λ_{act} evaluation contexts, as well as the syntax and typing rules of λ_{act} configurations. Evaluation contexts for terms and configurations are similar to λ_{ch}. The primary difference from λ_{ch} is the actor configuration $\langle a, M, \overrightarrow{V} \rangle$, which can be read as "an actor with name a evaluating term M, with a mailbox consisting of values $\overrightarrow{V}$". Whereas a term M is itself a configuration in λ_{ch}, a term in λ_{act} *must* be evaluated as part of an actor configuration. The typing rules for λ_{act} configurations ensure that all values contained in an actor mailbox are well-typed with respect to the mailbox type, and that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains an actor with name a.

Figure 10 shows the reduction rules for λ_{act}. Again, reduction on terms preserves typing, and the functional fragment of λ_{act} enjoys progress.

▶ **Lemma 9** (Preservation (λ_{act} terms)). *If $\Gamma \vdash M : A$ and $M \longrightarrow_M M'$, then $\Gamma \vdash M' : A$.*

▶ **Lemma 10** (Progress (λ_{act} terms)).
Assume Γ is either empty or only contains entries of the form $a_i : \mathsf{ActorRef}(A_i)$.
If $\Gamma \mid B \vdash M : A$, then either:

1. *$M = \mathsf{return}\ V$ for some value V*
2. *M can be written as $E[M']$, where M' is a communication or concurrency primitive (i.e. $\mathsf{spawn}\ N$, $\mathsf{send}\ V\,W$, $\mathsf{receive}$, or self)*
3. *There exists some M' such that $M \longrightarrow_M M'$*

Reduction on Configurations. While λ_{ch} makes use of separate constructs to create new processes and channels, λ_{act} uses a single construct $\mathsf{spawn}\ M$ to spawn a new actor with an empty mailbox to evaluate term M. Communication happens directly between actors

Syntax of evaluation contexts and configurations

Evaluation contexts	$E ::= [\,] \mid \mathsf{let}\ x \Leftarrow E\ \mathsf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid \langle a, M, \overrightarrow{V} \rangle$
Configuration contexts	$G ::= [\,] \mid G \parallel \mathcal{C} \mid (\nu a)G$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

Par
$$\frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2}$$

Pid
$$\frac{\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

Actor
$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : \mathbf{1} \quad (\Gamma, a : \mathsf{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

Figure 9 λ_{act} evaluation contexts and configurations

instead of through an intermediate entity: as a result of evaluating $\mathsf{send}\ V\ a$, the value V will be appended directly to the end of the mailbox of actor a. SendSelf allows reflexive sending; an alternative would be to decouple mailboxes from the definition of actors, but this complicates both the configuration typing rules and the intuition. Self returns the name of the current process, and Receive retrieves the head value of a non-empty mailbox.

As before, typing is preserved modulo structural congruence and under reduction.

▶ **Lemma 11.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and there exists a $\mathcal{D}$ such that $\mathcal{C} \equiv \mathcal{D}$, then $\Gamma; \Delta \vdash \mathcal{D}$.*

▶ **Theorem 12** (Preservation (λ_{act} configurations))**.** *If $\Gamma; \Delta \vdash \mathcal{C}_1$ and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, then $\Gamma; \Delta \vdash \mathcal{C}_2$.*

4.3 Progress and Canonical Forms

Again, we cannot guarantee deadlock-freedom for λ_{act}. Instead, we characterise the exact form of progress that λ_{act} enjoys: a well-typed configuration can always reduce unless all leaves of the configuration typing judgement are actors which have either fully evaluated their terms, or are blocked waiting for a message from an empty mailbox. Defining a canonical form again aids us in reasoning about progress.

▶ **Definition 13** (Canonical form (λ_{act}))**.** A λ_{act} configuration $\mathcal{C}$ is in *canonical form* if $\mathcal{C}$ can be written $(\nu a_1) \ldots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$.

▶ **Lemma 14.** *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\Delta = a_1 : A_1, \ldots a_k : A_k$, then there exists $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}' = (\nu a_{k+1}) \ldots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$.*

As before, it follows as a corollary of Lemma 14 that closed configurations can be written in canonical form, and with canonical forms defined, we can classify the notion of progress enjoyed by λ_{act}.

▶ **Corollary 15.** *If $\cdot; \cdot \vdash \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\mathcal{C}'$ is in canonical form.*

▶ **Theorem 16** (Weak progress (λ_{act} configurations))**.**
Let $\cdot; \cdot \vdash \mathcal{C}$, $\mathcal{C} \not\longrightarrow$, and let $\mathcal{C}' = (\nu a_1) \ldots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$ be a canonical form of $\mathcal{C}$. Each actor with name a_i is either of the form:

1. *$\langle a_i, \mathsf{return}\ W, \overrightarrow{V_i} \rangle$ for some value W, or;*
2. *$\langle a, E[\mathsf{receive}], \epsilon \rangle$.*

Reduction on terms

$$
\begin{array}{rcll}
(\lambda x.M)\,V & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\
\mathsf{let}\; x \Leftarrow \mathsf{return}\, V \;\mathsf{in}\; M & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\
E[M] & \longrightarrow_{\mathsf{M}} & E[M'] & \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}
$$

Structural congruence

$$
\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})
$$

$$
G[\mathcal{C}] \equiv G[\mathcal{D}] \text{ if } \mathcal{C} \equiv \mathcal{D}
$$

Reduction on configurations

$$
\begin{array}{rrcl}
\textsc{Spawn} & \langle a, E[\mathsf{spawn}\; M], \overrightarrow{V} \rangle & \longrightarrow & (\nu b)(\langle a, E[\mathsf{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \epsilon \rangle) \\
& & & \quad (b \text{ is fresh}) \\
\textsc{Send} & \langle a, E[\mathsf{send}\; V'\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle & \longrightarrow & \langle a, E[\mathsf{return}\,()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle \\
\textsc{SendSelf} & \langle a, E[\mathsf{send}\; V'\, a], \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\mathsf{return}\,()], \overrightarrow{V} \cdot V' \rangle \\
\textsc{Self} & \langle a, E[\mathsf{self}], \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\mathsf{return}\, a], \overrightarrow{V} \rangle \\
\textsc{Receive} & \langle a, E[\mathsf{receive}], W \cdot \overrightarrow{V} \rangle & \longrightarrow & \langle a, E[\mathsf{return}\, W], \overrightarrow{V} \rangle \\
\textsc{Lift} & G[\mathcal{C}_1] & \longrightarrow & G[\mathcal{C}_2] \qquad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) \\
\textsc{LiftM} & \langle a, M_1, \overrightarrow{V} \rangle & \longrightarrow & \langle a, M_2, \overrightarrow{V} \rangle \qquad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)
\end{array}
$$

Figure 10 Reduction on λ_{act} terms and configurations

5 From λ_{act} to λ_{ch}

With both calculi in place, we are now ready to look at the translation from λ_{act} into λ_{ch}. The key idea is to emulate a mailbox using a channel, and to pass the channel as an argument to each function. The translation on terms is parameterised over the variable referring to the channel, which is used to implement context-dependent operations (i.e. receive and self).

As an example, consider **recvAndDouble**, which is a specialisation of the **recvAndMult** function which doubles the number received from the mailbox.

$$\texttt{recvAndDouble} \triangleq \mathsf{let}\; x \Leftarrow \mathsf{receive}\;\mathsf{in}\;(x \times 2)$$

A possible configuration would be an actor evaluating **recvAndDouble**, with some name a and mailbox with values $\overrightarrow{V}$, under a name restriction for a.

$$(\nu a)(\langle a, \texttt{recvAndDouble}, \overrightarrow{V} \rangle)$$

The translation on terms takes a channel name ch as a parameter. As a result of the translation, we have that:

$$[\![\, \texttt{recvAndDouble} \,]\!]\; ch = \mathsf{let}\; x \Leftarrow \mathsf{take}\, ch\;\mathsf{in}\;(x \times 2)$$

with the corresponding configuration $(\nu a)(a([\![\, \overrightarrow{V} \,]\!]) \parallel [\![\, \texttt{recvAndDouble} \,]\!]\; a)$.

The values from the mailbox are translated pointwise and form the contents of a buffer with name a. The translation of **recvAndDouble** is provided with the name a which is used to emulate receive.

5.1 Translation (λ_{act} to λ_{ch})

Figure 11 shows the formal translation from λ_{act} into λ_{ch}. Of particular note is the translation on terms: $[\![-]\!]\; ch$ translates a λ_{act} term into an λ_{ch} term using a channel with name ch

Translation on types

$$[\![\mathsf{ActorRef}(A)]\!] = \mathsf{ChanRef}([\![A]\!]) \qquad [\![A \to^C B]\!] = [\![A]\!] \to \mathsf{ChanRef}([\![C]\!]) \to [\![B]\!] \qquad [\![\mathbf{1}]\!] = \mathbf{1}$$

Translation on values

$$[\![x]\!] = x \qquad [\![a]\!] = a \qquad [\![\lambda x.M]\!] = \lambda x.\lambda ch.([\![M]\!]\ ch) \qquad [\![()]\!] = ()$$

Translation on computation terms

$$[\![\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N]\!]\ ch = \mathsf{let}\ x \Leftarrow ([\![M]\!]\ ch)\ \mathsf{in}\ [\![N]\!]\ ch$$

$$\begin{array}{rcl} [\![V\ W]\!]\ ch &=& \mathsf{let}\ f \Leftarrow ([\![V]\!]\ [\![W]\!])\ \mathsf{in} f\ ch \\ [\![\mathsf{return}\ V]\!]\ ch &=& \mathsf{return}\ [\![V]\!] \\ [\![\mathsf{self}]\!]\ ch &=& \mathsf{return}\ ch \\ [\![\mathsf{receive}]\!]\ ch &=& \mathsf{take}\ ch \end{array} \qquad \begin{array}{rcl} [\![\mathsf{spawn}\ M]\!]\ ch &=& \mathsf{let}\ chMb \Leftarrow \mathsf{newCh}\ \mathsf{in} \\ && \mathsf{fork}\ ([\![M]\!]\ chMb); \\ && \mathsf{return}\ chMb \\ [\![\mathsf{send}\ V\ W]\!]\ ch &=& \mathsf{give}\ ([\![V]\!])\ ([\![W]\!]) \end{array}$$

Translation on configurations

$$[\![\mathcal{C}_1 \parallel \mathcal{C}_2]\!] = [\![\mathcal{C}_1]\!] \parallel [\![\mathcal{C}_2]\!] \qquad [\![(\nu a)\mathcal{C}]\!] = (\nu a)\ [\![\mathcal{C}]\!] \qquad [\![\langle a, M, \overrightarrow{V} \rangle]\!] = a([\![\overrightarrow{V}]\!]) \parallel ([\![M]\!]\ a)$$

Figure 11 Translation from λ_{act} into λ_{ch}

to emulate a mailbox. An actor reference is represented as a channel reference in λ_{ch}; we emulate sending a message to another actor by writing to the channel emulating the recipient's mailbox. Key to translating λ_{act} into λ_{ch} is the translation of function arrows $A \to^C B$; the effect annotation C is replaced by a second parameter $\mathsf{ChanRef}(C)$, which is used to emulate the mailbox of the actor. Values translate to themselves, with the exception of λ abstractions, whose translation takes an additional parameter denoting the channel used to emulate operations on a mailbox. Given parameter ch, the translation function for terms emulates receive by taking a value from ch, and emulates self by returning ch.

Though the translation is straightforward, it is a *global* translation [12], as all functions must be modified in order to take the channel emulating the mailbox as an additional parameter.

5.2 Properties of the Translation

The translation on terms and values preserves typing. We extend the translation function pointwise to typing environments: $[\![\alpha_1 : A_1, \ldots, \alpha_n : A_n]\!] = \alpha_1 : [\![A_1]\!], \ldots, \alpha_n : [\![A_n]\!]$

▶ **Lemma 17** ($[\![-]\!]$ preserves typing (terms and values)).

1. *If* $\Gamma \vdash V : A$ *in* λ_{act}, *then* $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!]$ *in* λ_{ch}.
2. *If* $\Gamma \mid B \vdash M : A$ *in* λ_{act}, *then* $[\![\Gamma]\!], \alpha : \mathsf{ChanRef}([\![B]\!]) \vdash [\![M]\!]\ \alpha : [\![A]\!]$ *in* λ_{ch}.

To state a semantics preservation result, we also define a translation on configurations; the translations on parallel composition and name restrictions are homomorphic. An actor configuration $\langle a, M, \overrightarrow{V} \rangle$ is translated as a buffer $a([\![\overrightarrow{V}]\!])$, (writing $[\![\overrightarrow{V}]\!] = [\![V_0]\!] \cdot, \ldots, \cdot [\![V_n]\!]$ for each $V_i \in \overrightarrow{V}$), composed in parallel with the translation of M, using a as the mailbox channel. We can now see that the translation preserves typeability of configurations.

▶ **Theorem 18** ($[\![-]\!]$ preserves typeability (configurations)).
If $\Gamma; \Delta \vdash \mathcal{C}$ *in* λ_{act}*, then* $[\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![\mathcal{C}]\!]$ *in* λ_{ch}*.*

We describe semantics preservation in terms of a simulation theorem: should a configuration $\mathcal{C}_1$ reduce to a configuration $\mathcal{C}_2$ in λ_{act}, then there exists some configuration $\mathcal{D}$ in λ_{ch} such that $[\![\mathcal{C}_1]\!]$ reduces in zero or more steps to $\mathcal{D}$, with $\mathcal{D} \equiv [\![\mathcal{C}_2]\!]$. To establish the result, we begin by showing that λ_{act} term reduction can be simulated in λ_{ch}.

▶ **Lemma 19** (Simulation of λ_{act} term reduction in λ_{ch}).
If $\Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_{\mathsf{M}} M_2$ *in* λ_{act}*, then given some* α*,* $[\![M_1]\!]\ \alpha \longrightarrow^*_{\mathsf{M}} [\![M_2]\!]\ \alpha$ *in* λ_{ch}*.*

Finally, we can see that the translation preserves structural congruences, and that λ_{ch} configurations can simulate reductions in λ_{act}.

▶ **Lemma 20.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $[\![\mathcal{C}]\!] \equiv [\![\mathcal{D}]\!]$*.*

▶ **Theorem 21** (Simulation of λ_{act} configurations in λ_{ch}).
If $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $[\![\mathcal{C}_1]\!] \longrightarrow^* \mathcal{D}$*, with* $\mathcal{D} \equiv [\![\mathcal{C}_2]\!]$*.*

6 From λ_{ch} to λ_{act}

The translation from λ_{act} into λ_{ch} emulates an actor mailbox using a channel, using it to implement operations which normally rely on the context of the actor. Though global, the translation is straightforward due to the limited form of communication supported by mailboxes. Translating from λ_{ch} into λ_{act} is more challenging. Each channel in a system may have a different type; each process may have access to multiple channels; and (crucially) channels may be freely passed between processes.

6.1 Extensions to the Core Language

To emulate channels using actors, we require several more term-level language constructs: sums, products, recursive functions, and iso-recursive types. Recursive functions are used to implement an event loop, and recursive types are used to maintain a buffer at the term level in addition to the meta-level. Products are used to emulate the state of a channel, in particular to record both a list of values in the buffer and a list of pending requests. Sum types allow the disambiguation of the two types of messages sent to an actor: one to queue a message (emulating give) and one to dequeue a message and return it to the actor making the request (emulating take). Additionally, sums can be straightforwardly used to encode monomorphic variant types. We write $\langle \ell_1 : A_1, \ldots, \ell_n : A_n \rangle$ for variant types and $\langle \ell_i = V \rangle$ for variant values.

Figure 12 shows the extensions required to the core term language and reduction rules; we omit the reduction rule for case analysis on the right injection of a sum. With products, sums, and recursive types, we can encode lists. The typing rules are shown for λ_{ch} but can be easily adapted for λ_{act}, and it is straightforward to verify that the extended languages still enjoy progress and preservation.

6.2 Translation Strategy (λ_{ch} into λ_{act})

To translate typed actors into typed channels (shown in Figure 13), we emulate each channel using an actor process, which is crucial in retaining the mobility of channel endpoints. Channel types describe the typing of a *communication medium* between communicating

Syntax

$$\begin{array}{lrcl} \text{Types} & A, B, C & ::= & A \times B \mid A + B \mid \mathsf{List}(A) \mid \mu X.A \mid X \mid \ldots \\ \text{Values} & V, W & ::= & \mathsf{rec}\, f(x)\,.\,M \mid (V, W) \mid \mathsf{inl}\, V \mid \mathsf{inr}\, W \mid \mathsf{roll}\, V \mid \ldots \\ \text{Terms} & L, M, N & ::= & \mathsf{let}\,(x, y) = V\,\mathsf{in}\, M \mid \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} \mid \mathsf{unroll}\, V \ldots \end{array}$$

Additional value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{Rec}\ \frac{\Gamma, x : A, f : A \to B \vdash M : B}{\Gamma \vdash \mathsf{rec}\, f(x)\,.\,M : A \to B} \qquad \textsc{Pair}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : B}{\Gamma \vdash (V, W) : A \times B} \qquad \textsc{Inl}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathsf{inl}\, V : A + B} \qquad \textsc{Inr}\ \frac{\Gamma \vdash V : B}{\Gamma \vdash \mathsf{inr}\, V : A + B}$$

$$\textsc{Roll}\ \frac{\Gamma \vdash V : A\{\mu X.A/X\}}{\Gamma \vdash \mathsf{roll}\, V : \mu X.A}$$

Additional term typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{Let}\ \frac{\Gamma \vdash V : A \times A' \qquad \Gamma, x : A, y : A' \vdash M : B}{\Gamma \vdash \mathsf{let}\,(x, y) = V\,\mathsf{in}\, M : B} \qquad \textsc{Case}\ \frac{\Gamma \vdash V : A + A' \qquad \Gamma, x : A \vdash M : B \quad \Gamma, y : A' \vdash N : B}{\Gamma \vdash \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} : B} \qquad \textsc{Unroll}\ \frac{\Gamma \vdash V : \mu X.A}{\Gamma \vdash \mathsf{unroll}\, V : A\{\mu X.A/X\}}$$

Additional term reduction rules $\boxed{M \longrightarrow_{\mathsf{M}} M'}$

$$\begin{array}{rcl} (\mathsf{rec}\, f(x)\,.\,M\ V) & \longrightarrow_{\mathsf{M}} & M\{(\mathsf{rec}\, f(x)\,.\,M)/f, V/x\} \\ \mathsf{let}\,(x, y) = (V, W)\,\mathsf{in}\, M & \longrightarrow_{\mathsf{M}} & M\{V/x, W/y\} \\ \mathsf{case}\,(\mathsf{inl}\, V)\,\{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} & \longrightarrow_{\mathsf{M}} & M\{V/x\} \\ \mathsf{unroll}\,(\mathsf{roll}\, V) & \longrightarrow_{\mathsf{M}} & \mathsf{return}\, V \end{array}$$

Encoding of lists

$$\mathsf{List}(A) \triangleq \mu X.1 + (A \times X) \qquad [\ ] \triangleq \mathsf{roll}\,(\mathsf{inl}\,()) \qquad V :: W \triangleq \mathsf{roll}\,(\mathsf{inr}\,(V, W))$$

$$\mathsf{case}\, V\, \{[\ ] \mapsto M; x :: y \mapsto N\} \triangleq \mathsf{let}\, z \Leftarrow \mathsf{unroll}\, V\ \mathsf{in}\ \mathsf{case}\, z\, \{\mathsf{inl}\,() \mapsto M; \mathsf{inr}\,(x, y) \mapsto N\}$$

Figure 12 Extensions to core languages to allow translation from λ_{ch} into λ_{act}

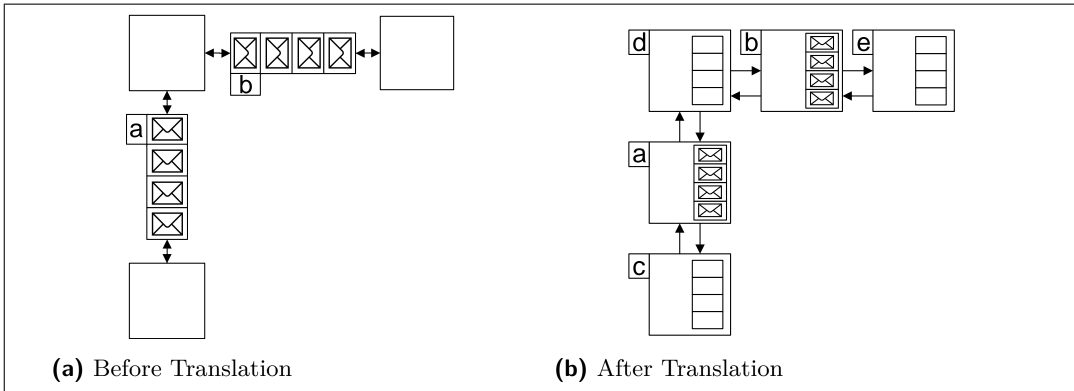
processes, where processes are unaware of the identity of other communicating parties, and the types of messages that another party may receive. Unfortunately, the same does not hold for mailboxes. Consequently, we require that before translating into actors, *every channel has the same type*. Although this may seem restrictive, it is both possible and safe to transform a λ_{ch} program with multiple channel types into a λ_{ch} program with a single channel type.

As an example, suppose we have a program which contains channels carrying values of types Int, String, and ChanRef(String). It is possible to construct a recursive variant type $\mu X.\langle \ell_1 : \mathsf{Int}, \ell_2 : \mathsf{String}, \ell_3 : \mathsf{ChanRef}(X)\rangle$ which can be assigned to all channels in the system. Then, supposing we wanted to send a 5 along a channel which previously had type ChanRef(Int), we would instead send a value $\mathsf{roll}\ \langle \ell_1 = 5\rangle$ (where $\mathsf{roll}\ V$ is the introduction rule for an iso-recursive type). Appendix A provides more details, and proves that the transformation is safe.

6.3 Translation

We write λ_{ch} judgements of the form $\{B\}\ \Gamma \vdash M : A$ for a term where all channels have type B, and similarly for value and configuration typing judgements. Under such a judgement, we can write Chan instead of ChanRef(B).

Metalevel Definitions. The majority of the translation lies within the translation of newCh, which makes use of the meta-level definitions **body** and **drain**. The **body** function emulates a channel. Firstly, the actor receives a message *recvVal* from its mailbox, which is



(a) Before Translation **(b)** After Translation

Figure 13 Translation Strategy: λ_{ch} into λ_{act}

either of the form $\mathsf{inl}\, V$ to store a message V, or $\mathsf{inr}\, W$ to request that a value is dequeued and sent to the actor with ID W. We assume a standard implementation of the list concatenation function $(+\!\!+)$. If the message is $\mathsf{inl}\, V$, then V is appended to the tail of the list of values stored in the channel, and the new state is passed as an argument to **drain**. If the message is $\mathsf{inr}\, W$, then the process ID W is appended to the end of the list of processes waiting for a value. The **drain** function satisfies all requests that can be satisfied, returning an updated channel state.

Translation on Types. Figure 14 shows the translation from λ_{ch} into λ_{act}. The translation function on types $(\!|-|\!)$ is defined with respect to the type of all channels C and is used to annotate function arrows and to assign a parameter to ActorRef types. The (omitted) translations on sums, products, and lists are homomorphic. The translation of Chan is $\mathsf{ActorRef}((\!|C|\!) + \mathsf{ActorRef}((\!|C|\!)))$, meaning an actor which can receive a request to either store a value of type $(\!|C|\!)$, or to dequeue a value and send it to a process ID of type $\mathsf{ActorRef}((\!|C|\!))$.

Translation on Communication and Concurrency Primitives. We omit the translation on values and functional terms, which are homomorphisms. Processes in λ_{ch} are anonymous, whereas all actors in λ_{act} are addressable; to emulate fork, we therefore discard the reference returned by spawn. The translation of give wraps the translated value to be sent in the left injection of a sum type, and sends to the translated channel name $(\!|W|\!)$.

To emulate take, self is firstly used to retrieve the process ID of the actor. Next, the process ID is wrapped in the right injection and sent to the actor emulating the channel, and the actor waits for the response message.

Finally, the translation of newCh spawns a new actor to execute **body**.

Translation on Configurations. The translation function $(\!|-|\!)$ is homomorphic on parallel composition and name restriction. Unlike λ_{ch}, a term cannot exist outwith an enclosing actor context in λ_{act}, so the translation of a process evaluating term M is an actor evaluating $(\!|M|\!)$ with some fresh name a and an empty mailbox, enclosed in a name restriction.

The translation of a λ_{ch} buffer requires a *term-level* list to be constructed from a *meta-level* sequence; the mailbox is required for requests to queue and dequeue values. Moreover, the translation of a buffer is an actor with an empty mailbox which evaluates **body** with a state containing the (term-level) list of values, and an empty request queue.

In contrast to the global transformation in the previous section, although the translation from λ_{ch} into λ_{act}, is much more verbose, it is (once all channels have the same type) a *local transformation* [12].

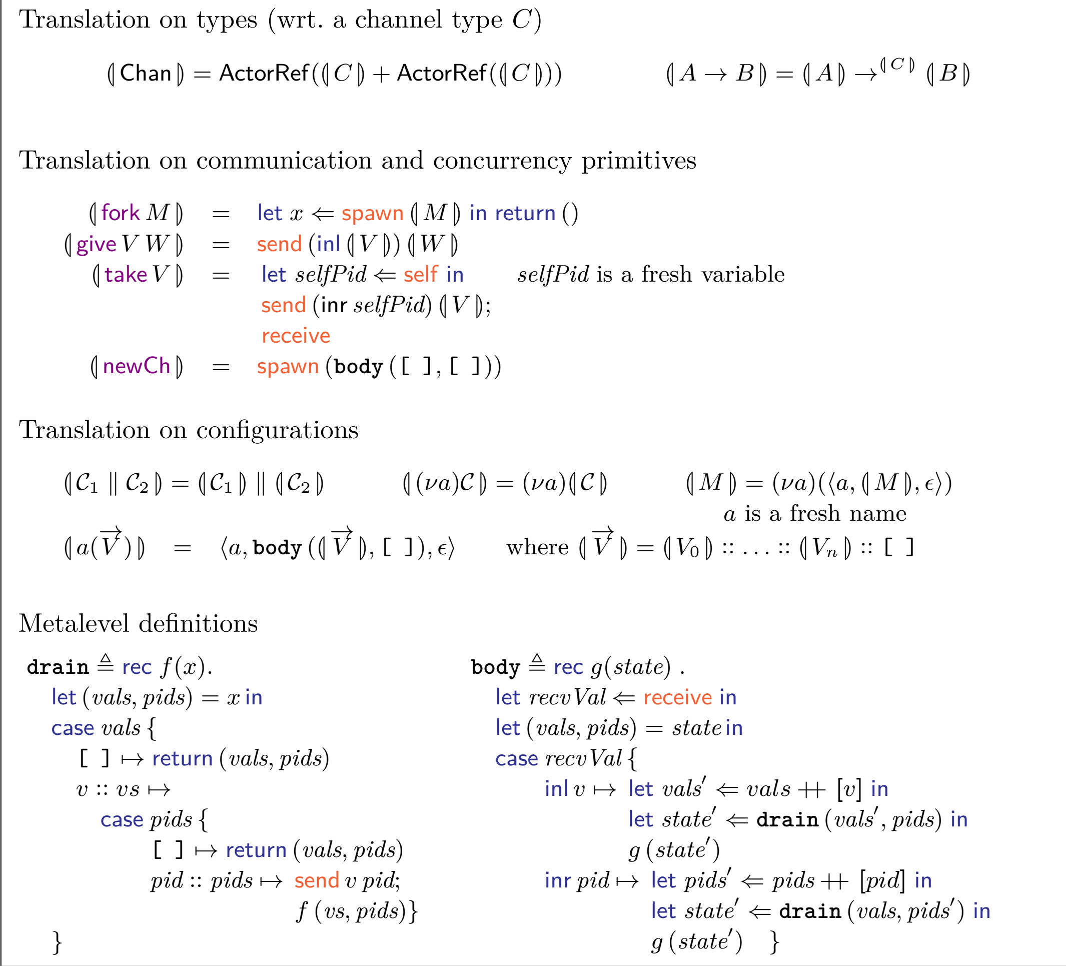


Figure 14 Translation from λ_{ch} into λ_{act}

6.4 Properties of the Translation

We firstly define translations on typing environments. Since all channels in the source language of the translation have the same type, we can assume that each entry in the codomain of Δ is the same type B. Importantly, each entry in the translated environment refers to the name of a *channel*, and thus has the same type as the translation of Chan.

▶ **Definition 22** (Translation of typing environments wrt. a channel type B).

1. If $\Gamma = \alpha_1{:}A_1, \ldots, \alpha_n : A_n$, define $(\!|\,\Gamma\,|\!) = \alpha_1 : (\!|\,A_1\,|\!), \ldots, \alpha_n : (\!|\,A_n\,|\!)$.
2. Given a $\Delta = a_1 : B, \ldots, a_n : B$, define $(\!|\,\Delta\,|\!) =$
   $a_1 : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!))), \ldots, a_n : ((\!|\,B\,|\!) + \mathsf{ActorRef}((\!|\,B\,|\!)))$.

The translation on terms preserves typing.

▶ **Lemma 23** ($(\!|\,-\,|\!)$ preserves typing (terms and values)).

1. *If* $\{B\}\;\Gamma \vdash V{:}A$*, then* $(\!|\,\Gamma\,|\!) \vdash (\!|\,V\,|\!){:}(\!|\,A\,|\!)$.
2. *If* $\{B\}\;\Gamma \vdash M{:}A$*, then* $(\!|\,\Gamma\,|\!) \mid (\!|\,B\,|\!) \vdash (\!|\,M\,|\!){:}(\!|\,A\,|\!)$.

The translation on configurations also preserves typeability. We write $\Gamma \asymp \Delta$ if for each $a : A \in \Delta$, we have that $a : \mathsf{ChanRef}(A) \in \Gamma$; for closed configurations this is ensured by

Chan. This is necessary since the typing rules for λ_{act} require that the local actor name is present in the term environment to ensure preservation in the presence of self, but there is no such restriction in λ_{ch}.

▶ **Theorem 24** ($(\!|-|\!)$ preserves typeability (configurations)). *If* $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$*, then* $(\!|\Gamma|\!)A;(\!|\Delta|\!)A \vdash (\!|\mathcal{C}|\!)$.

It is clear that reduction on translated λ_{ch} terms can simulate reduction in λ_{act}; in fact, we obtain a tighter (lockstep) simulation result than the translation from λ_{act} into λ_{ch} since β-reduction only requires one reduction instead of two.

▶ **Lemma 25.** *If* $\{B\}\ \Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$*, then* $(\!|M_1|\!) \longrightarrow_M (\!|M_2|\!)$.

Finally, we show that λ_{act} can simulate λ_{ch}.

▶ **Lemma 26.** *If* $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $(\!|\mathcal{C}|\!) \equiv (\!|\mathcal{D}|\!)$.

▶ **Theorem 27** (Simulation (λ_{act} configurations in λ_{ch})). *If* $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}_1$*, and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $(\!|\mathcal{C}_1|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\mathcal{C}_2|\!)$.

7 Extensions

In this section, we discuss common extensions to channel- and actor-based languages. Firstly, we discuss synchronisation, which is ubiquitous in practical implementations of actor-inspired languages. Adding synchronisation simplifies the translation from channels to actors, and relaxes the restriction that all channels must have the same type. Secondly, we consider an extension with Erlang-style selective receive, and show how to encode it in λ_{act}. Thirdly, we discuss how to nondeterministically choose a message from a collection of possible sources, and finally, we discuss what the translations tell us about the nature of behavioural typing disciplines for actors. Establishing exactly how the latter two extensions fit into our framework is the subject of ongoing and future work.

7.1 Synchronisation

While communicating with an actor via asynchronous message passing suffices for many purposes, the approach can become cumbersome when implementing "call-response" style interactions. Practical implementations such as Erlang and Akka implement some way of synchronising on a result: Erlang achieves this by generating a unique reference to send along with a request, *selectively receiving* from the mailbox to await a response tagged with the same unique reference. Another method of synchronisation embraced by the Active Object community [10, 27, 28] as well as the Akka framework is to generate a *future variable* which is populated with the result of the call.

Figure 15 details an extension of λ_{act} with a synchronisation primitive, wait. In this extension, we replace the unary type constructor for process IDs with a binary type constructor ActorRef(A, B), where A is the type of messages that the process can receive from its mailbox, and B is the type of value to which the process will eventually evaluate. We assume that the remainder of the primitives are modified to take the additional effect type into account. A variation of the wait primitive is implemented as part of the Links [9] concurrency model to address the type pollution problem.

We now adapt the previous translation from λ_{ch} to λ_{act}, making use of wait to avoid the need for the coalescing transformation. In the modified translation, channel references are

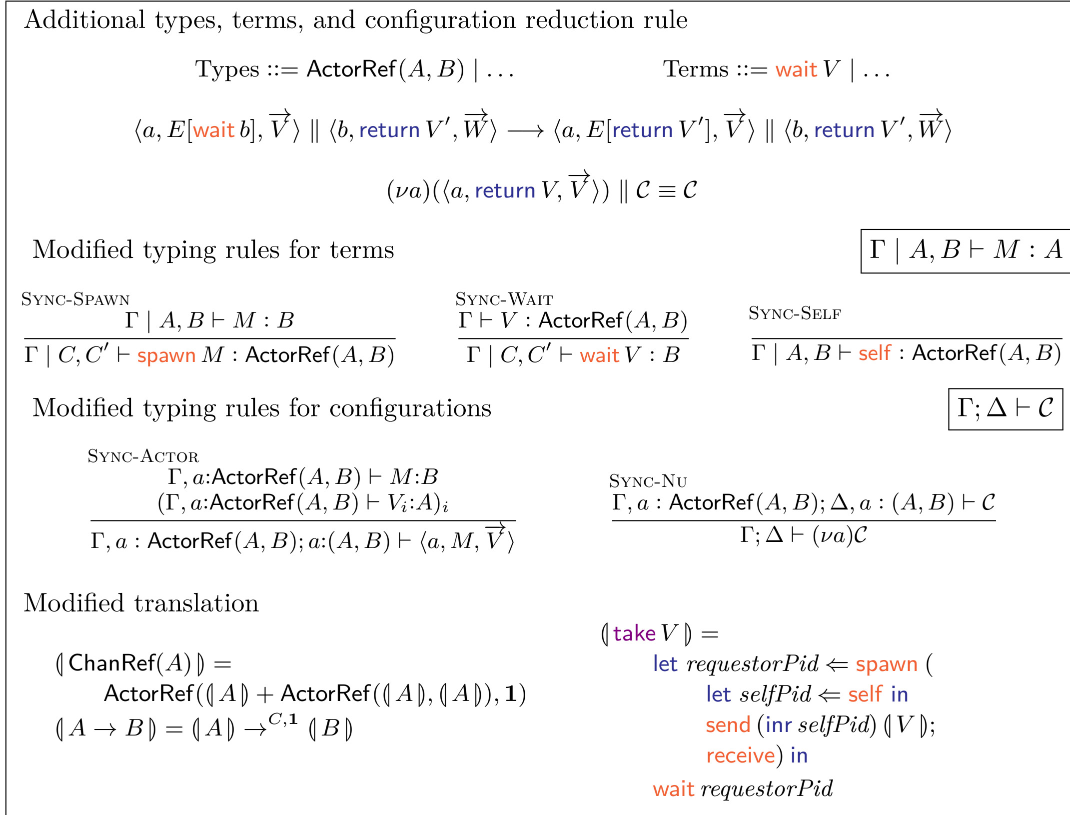


Figure 15 Extensions to add synchronisation to λ_{act}

translated into actor references which can either receive a value of type A, or a process which can receive a value of type A and will eventually evaluate to a value of type A. Note that the unbound annotation $C, 1$ on function arrows reflects that the mailboxes can be of *any* type, since the mailboxes are unused in the actors emulating threads.

The key idea behind the modified translation is to spawn a fresh actor which makes the request to the channel and blocks waiting for the response. Once the spawned actor has received the result, the result can be retrieved synchronously using wait *without* reading from the mailbox. The previous soundness theorems adapt to the new setting.

▶ **Theorem 28.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$*, then* $(\!|\Gamma|\!); (\!|\Delta|\!) \vdash (\!|\mathcal{C}|\!)$.

▶ **Theorem 29.** *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then there exists some* $\mathcal{D}$ *such that* $(\!|\mathcal{C}|\!) \longrightarrow^* \mathcal{D}$ *with* $\mathcal{D} \equiv (\!|\mathcal{C}_2|\!)$.

The translation in the other direction requires named threads and a join construct in λ_{ch}.

7.2 Selective Receive

The receive construct in λ_{act} can only read the first message in the queue, which is cumbersome as it often only makes sense for an actor to handle a subset of possible messages at a given time. In practice, Erlang provides a *selective receive* construct, matching messages in the mailbox against multiple pattern clauses. Assume we have a mailbox containing values $V_1, \ldots V_n$ and perform a selective receive receive $\{c_1, \ldots, c_m\}$. The selective receive first tries to match value V_1 against clause c_1—if the V_1 matches the pattern of c_1, then the body of c_1

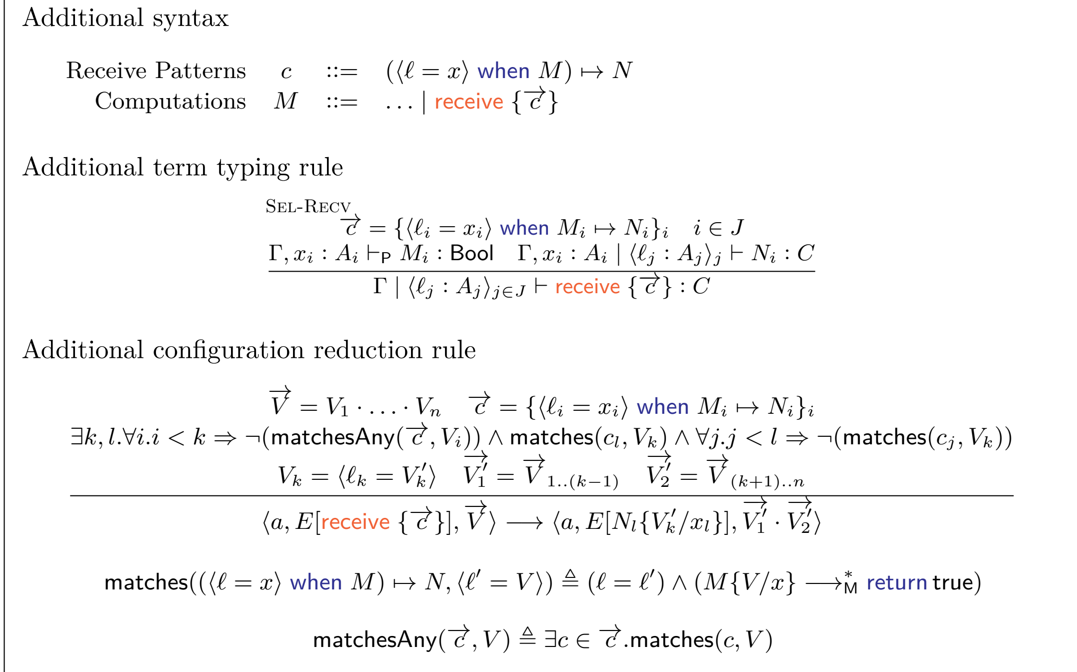


Figure 16 Additional syntax, typing rules, and reduction rules for λ_{act} with selective receive

is evaluated, whereas if it fails, V_1 is tested against c_2 and so on. Should V_1 not match any of the patterns, then the process is repeated with V_2; and subsequently each V_i until V_n has been tested against c_m. At this point, execution blocks until a matching message arrives.

More concretely, consider an actor with mailbox type $C = \langle \mathsf{PriorityMessage} : \mathsf{Message}, \mathsf{StandardMessage} : \mathsf{Message}, \mathsf{Timeout} : \mathbf{1} \rangle$ which can receive both high- and low-priority messages. Let *getPriority* be a function which extracts a priority from a message.
Now consider the following actor:

```
receive {
    ⟨PriorityMessage = msg⟩ when (getPriority msg) > 5 ↦ handleMessage msg
    ⟨Timeout = msg⟩         when true                  ↦ ()
};
receive {
    ⟨PriorityMessage = msg⟩  when true ↦ handleMessage msg
    ⟨StandardMessage = msg⟩  when true ↦ handleMessage msg
    ⟨Timeout = msg⟩          when true ↦ ()
}
```

This actor begins by handling a message only if it has a priority greater than 5. After the timeout message is received, however, it will handle any message—including lower-priority messages that were received beforehand.

Figure 16 shows the additional syntax, typing rule, and configuration reduction rule required to encode selective receive; the type Bool and logical operators are encoded using sums in the standard way. We write $\Gamma \vdash_{\mathsf{P}} M : A$ to mean that under context Γ, a term M which does not perform any communication or concurrency actions has type A. Intuitively, this means that no subterm of M is a communication or concurrency construct.

The receive $\{\overrightarrow{c}\}$ construct models an ordered sequence of receive pattern clauses c of the form $(\langle \ell = x \rangle$ when $M) \mapsto N$, which can be read as "If a message with body x has label ℓ and satisfies predicate M, then evaluate N". The typing rule for receive $\{\overrightarrow{c}\}$ ensures that for

Translation on types

$$\lfloor \mathsf{ActorRef}(\langle \ell_i : A_i \rangle_i) \rfloor = \mathsf{ActorRef}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i) \qquad \lfloor A \times B \rfloor = \lfloor A \rfloor \times \lfloor B \rfloor \qquad \lfloor A + B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor$$

$$\lfloor \mu X.A \rfloor = \mu X.\lfloor A \rfloor \qquad \lfloor A \to^C B \rfloor = \lfloor A \rfloor \to^{\lfloor C \rfloor} \mathsf{List}(\lfloor C \rfloor) \to^{\lfloor C \rfloor} (\lfloor B \rfloor \times \mathsf{List}(\lfloor C \rfloor))$$

where $C = \langle \ell_i : A'_i \rangle_i$, and $\lfloor C \rfloor = \langle \ell_i : \lfloor A'_i \rfloor \rangle_i$

Translation on values

$$\lfloor \lambda x.M \rfloor = \lambda x.\lambda mb.(\lfloor M \rfloor\, mb) \qquad \lfloor \mathsf{rec}\, f(x)\,.\,M \rfloor = \mathsf{rec}\, f(x)\,.\,\lambda mb.(\lfloor M \rfloor\, mb)$$

Translation on computation terms (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$)

$$\begin{aligned}
\lfloor V\, W \rfloor\, mb &= \mathsf{let}\, f \Leftarrow (\lfloor V \rfloor\, \lfloor W \rfloor)\ \mathsf{in}\ f\, mb \\
\lfloor \mathsf{return}\, V \rfloor\, mb &= \mathsf{return}\,(\lfloor V \rfloor, mb) \\
\lfloor \mathsf{let}\, x \Leftarrow M\ \mathsf{in}\ N \rfloor\, mb &= \mathsf{let}\ resPair \Leftarrow \lfloor M \rfloor\, mb\ \mathsf{in}\ \mathsf{let}\,(x, mb') = resPair\ \ \mathsf{in}\ \lfloor N \rfloor\, mb' \\
\lfloor \mathsf{self} \rfloor\, mb &= \mathsf{let}\ selfPid \Leftarrow \mathsf{self}\ \mathsf{in}\ \mathsf{return}\,(selfPid, mb) \\
\lfloor \mathsf{send}\, V\, W \rfloor\, mb &= \mathsf{let}\, x \Leftarrow \mathsf{send}\,(\lfloor V \rfloor)\,(\lfloor W \rfloor)\ \mathsf{in}\ \mathsf{return}\,(x, mb) \\
\lfloor \mathsf{spawn}\, M \rfloor\, mb &= \mathsf{let}\ spawnRes \Leftarrow \mathsf{spawn}(\lfloor M \rfloor\, [\ ])\ \mathsf{in}\ \mathsf{return}\,(spawnRes, mb) \\
\lfloor \mathsf{receive}\, \{\overrightarrow{c}\} \rfloor\, mb &= \mathtt{find}(\overrightarrow{c}, mb)
\end{aligned}$$

Translation on configurations

$$\begin{aligned}
\lfloor (\nu a)\mathcal{C} \rfloor &= \{(\nu a)\mathcal{D} \mid \mathcal{D} \in \lfloor \mathcal{C} \rfloor\} \\
\lfloor \mathcal{C}_1 \parallel \mathcal{C}_2 \rfloor &= \{\mathcal{D}_1 \parallel \mathcal{D}_2 \mid \mathcal{D}_1 \in \lfloor \mathcal{C}_1 \rfloor \wedge \mathcal{D}_2 \in \lfloor \mathcal{C}_2 \rfloor\} \\
\lfloor \langle a, M, \overrightarrow{V} \rangle \rfloor &= \{\langle a, \lfloor M \rfloor\, [\ ], \lfloor \overrightarrow{V} \rfloor \rangle\} \cup \\
&\quad\ \{\langle a, (\lfloor M \rfloor\, \overrightarrow{W_i^1}), \overrightarrow{W_i^2} \rangle \mid i \in 1..n\}
\end{aligned}$$

where

$$\overrightarrow{W_i^1} = \lfloor V_1 \rfloor :: \ldots :: \lfloor V_i \rfloor :: [\ ] \qquad \overrightarrow{W_i^2} = \lfloor V_{i+1} \rfloor \cdot \ldots \cdot \lfloor V_n \rfloor$$

Figure 17 Translation from λ_{act} with selective receive into λ_{act}

each pattern $\langle \ell_i = x_i \rangle$ when $M_i \mapsto N_i$ in $\overrightarrow{c}$, we have that there exists some $\ell_i : A_i$ contained in the mailbox variant type; and when Γ is extended with $x_i : A_i$, that the guard M_i has type Bool and the body N_i has the same type C for each branch.

The reduction rule for selective receive is inspired by that of Fredlund [14]. Assume that the mailbox is of the form $V_1 \cdot \ldots \cdot V_k \cdot \ldots V_n$, with $\overrightarrow{V'_1} = V_1 \cdot \ldots \cdot V_{k-1}$ and $\overrightarrow{V'_2} = V_{k+1} \cdot \ldots \cdot V_n$. The $\mathsf{matches}(c, V)$ predicate is true if the label matches, and the branch guard evaluates to true. The $\mathsf{matchesAny}(\overrightarrow{c}, V)$ predicate returns true if V matches any pattern in $\overrightarrow{c}$. The key idea is that V_k is the first value to satisfy a pattern. The construct evaluates to the body of the matched pattern, with the message payload V'_k substituted for the pattern variable x_k; the final mailbox is the concatenation of $\overrightarrow{V'_1}$ and $\overrightarrow{V'_2}$ (that is, the original mailbox without V_k). Reduction in the presence of selective receive preserves typing.

▶ **Theorem 30** (Preservation (λ_{act} configurations with selective receive)). *If* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then* $\Gamma; \Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_2$*.*

Translation to λ_{act}. Although λ_{act} is a basic calculus, given the additional constructs we used for the translation from λ_{ch} into λ_{act}, it is possible to translate λ_{act} with selective receive into λ_{act} without selective receive. The key to the translation is reasoning about values in the mailbox at the term level; similar to the translation from λ_{ch} into λ_{act}, we maintain a term-level 'save queue' of values that have been received but not yet matched, and can loop through the list to find the first matching term.

Figure 17 shows the translation from λ_{act} with selective receive into plain λ_{act}. The translation on types is largely homomorphic, except for the translation on functions: here, similar to the translation from λ_{act} into λ_{ch}, we add an additional parameter referring to the save queue. The translation on values is similar, adding an extra parameter to functions to capture the save queue.

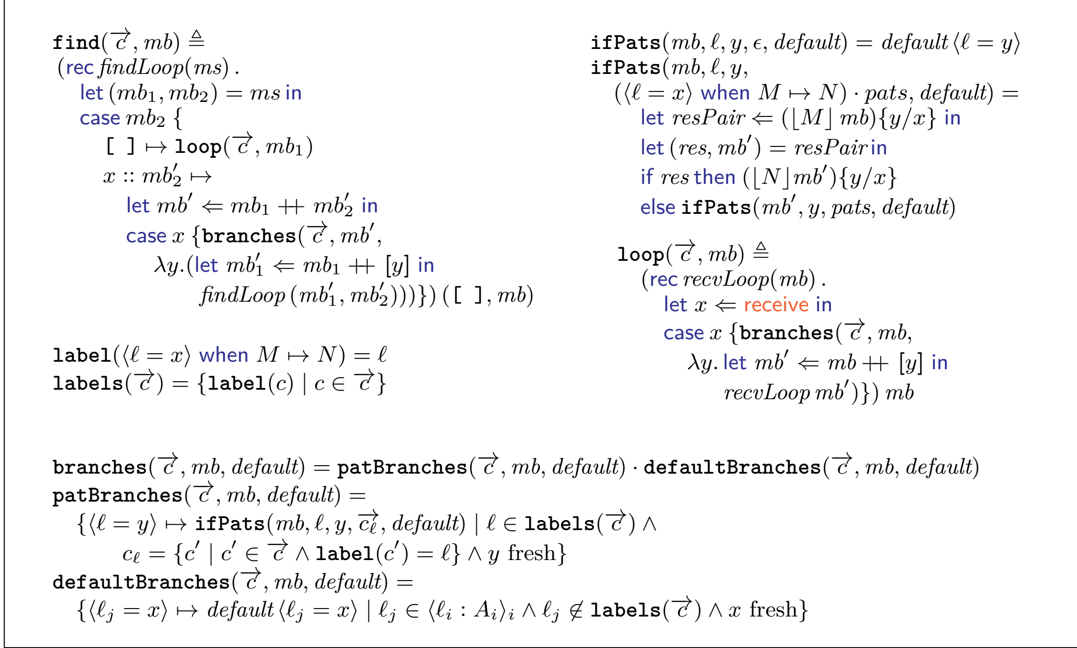


Figure 18 Metalevel definitions for translation from λ_{act} with selective receive to λ_{act}

The translation on terms $\lfloor M \rfloor\, mb$ takes a variable mb representing the save queue as its parameter, returning a pair of the resulting term and the updated save queue. The majority of cases are standard, except for receive $\{\overrightarrow{c}\}$, which relies on the meta-level definition $\texttt{find}(\overrightarrow{c}, mb)$, which takes as its arguments a sequence of clauses $\overrightarrow{c}$ and a save queue mb, and consists of a loop *findLoop*. The loop takes a pair of lists (ms_1, ms_2), where ms_1 is the list of values that has been processed already and found not to match, and ms_2 is a list of values still to be processed. The loop iterates through the list of values until one either matches, or the end of the list is reached. Should no values in the term-level representation of the mailbox match, then the **loop** function repeatedly receives from the mailbox, testing each new message to see whether it matches any of the patterns.

Note that the case construct in the core λ_{act} calculus is more restrictive than that of the selective receive construct: given a variant $\langle \ell_i : A_i \rangle_i$ used as a mailbox type, the core calculus requires a single branch for each possible value, whereas selective receive allows multiple different branches for each label (each containing a possibly-different predicate), and does not require pattern matching to be exhaustive. In order to translate the liberal selective receive into the more basic case construct, we perform pattern matching compilation through the use of the **branches** construct: **patBranches** creates a branch for each label present in the selective receive, creating (via **ifPats**) a sequence of if-then-else statements to check whether a value satisfies each predicate in turn; **defaultBranches** creates a branch for each label that is present in the mailbox type but not in any of the selective receive clauses.

Properties of the translation. The translation preserves typing of terms and values.

▶ **Lemma 31** (Translation preserves typing (values and terms)).

1. *If* $\Gamma \vdash V : A$, *then* $\lfloor \Gamma \rfloor \vdash \lfloor V \rfloor : \lfloor A \rfloor$.
2. *If* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : B$, *then* $\lfloor \Gamma \rfloor, mb : \mathsf{List}(\langle \ell_i : \lfloor A_i \rfloor \rangle)_i \mid \langle \ell_i : \lfloor A_i \rfloor \rangle_i \vdash \lfloor M \rfloor mb : (\lfloor B \rfloor \times \mathsf{List}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i))$.

$$\frac{\Gamma \vdash V : \mathsf{ChanRef}(A) \quad \Gamma \vdash W : \mathsf{ChanRef}(B)}{\Gamma \vdash \mathsf{choose}\, V\, W : A + B}$$

$$E[\mathsf{choose}\, a\, b] \parallel a(W_1 \cdot \overrightarrow{V_1}) \parallel b(\overrightarrow{V_2}) \longrightarrow E[\mathsf{return}\, (\mathsf{inl}\, W_1)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2})$$
$$E[\mathsf{choose}\, a\, b] \parallel a(\overrightarrow{V_1}) \parallel b(W_2 \cdot \overrightarrow{V_2}) \longrightarrow E[\mathsf{return}\, (\mathsf{inr}\, W_2)] \parallel a(\overrightarrow{V_1}) \parallel b(\overrightarrow{V_2})$$

Figure 19 Additional typing and evaluation rules for λ_{ch} with choice

To obtain a semantics preservation result, again we provide a translation on configurations. Alas, it is not possible to do a direct one-to-one translation, since a message in a mailbox in the source language could be either in the mailbox or the save queue in the target language. Consequently, we translate a configuration into a set of possible configurations, depending on how many messages have been processed. We can show that all configurations in the resulting set are type-correct, and can simulate the original reduction—for a set of configurations $\mathcal{S}$, we write $\mathcal{S}^{\equiv}$ for the set of configurations such that $\mathcal{C} \in \mathcal{S}^{\equiv} \Leftrightarrow \mathcal{C}' \in \mathcal{S} \wedge \mathcal{C} \equiv \mathcal{C}'$.

▶ **Theorem 32** (Translation preserves typing). *If* $\Gamma; \Delta \vdash \mathcal{C}$*, then* $\forall \mathcal{D} \in \lfloor \mathcal{C} \rfloor$*, it is the case that* $\lfloor \Gamma \rfloor; \lfloor \Delta \rfloor \vdash \mathcal{D}$*.*

▶ **Theorem 33** (Simulation (λ_{act} with selective receive in λ_{act})). *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$*, then* $\forall \mathcal{D} \in \lfloor \mathcal{C} \rfloor$*, there exists a* $\mathcal{D}'$ *such that* $\mathcal{D} \longrightarrow^{+} \mathcal{D}'$ *and* $\mathcal{D}' \in \lfloor \mathcal{C}' \rfloor^{\equiv}$*.*

7.3 Choice

The calculus λ_{ch} supports only blocking receive on a *single* channel. A more powerful mechanism is *selective communication*, where a value is taken nondeterministically from *two* channels. An important use case is receiving a value when either channel could be empty.

Here we have considered only the most basic form of selective choice over two channels. More generally, it may be extended to arbitrary regular data types [37]. As Concurrent ML [39] embraces rendezvous-based synchronous communication, it provides *generalised selective communication* where a process can synchronise on a mixture of input or output communication events. Similarly, the join patterns of the join calculus [13] provide a general abstraction for selective communication over multiple channels.

As we are working in the asynchronous setting where a give operation can reduce immediately, we consider only input-guarded choice. Input-guarded choice can be added straightforwardly to λ_{ch}, as shown in Figure 19. Emulating such a construct satisfactorily in λ_{act} is nontrivial, because messages must be multiplexed through a local queue. One approach could be to use the work of Chaudhuri [8] which shows how to implement generalised choice using synchronous message passing, but implementing this in λ_{ch} may be difficult due to the asynchrony of give. We leave a more thorough investigation to future work.

7.4 Behavioural Types

Behavioural types allow the type of an object (e.g. a channel) to evolve as a program executes. A widely studied behavioural typing discipline is that of *session types* [22, 23] which supports channel types that are sufficiently rich to describe *communication protocols* between participants. As an example, the session type for a channel which sends two integers and receives their sum could be defined as !Int.!Int.?Int.end, where $!A.S$ is the type of a channel which sends a value of type A before continuing with behaviour S. Session types are

particularly suited to channels, whereas current work on session-typed actors concentrates on runtime monitoring [34].

A natural question to ask is whether one can combine the benefits of actors and of session types—indeed, this was one of our original motivations for wanting to better understand the relationship between actors and channels in the first place! A session-typed channel may support both sending and receiving (at different points in the protocol it encodes). But communication with another processes mailbox is one-way. We have studied several variants of λ_{act} with *polarised* session types which capture such one-way communication, but they seem too weak to simulate session-typed channels. In future, we would like to find a natural extension of λ_{act} with behavioural types that admits a similar simulation result to the ones in this paper.

8 Related Work

Our formulation of concurrent λ-calculi is inspired by $\lambda(\mathsf{fut})$ [35], a concurrent λ-calculus with threads and future variables, as well as reference cells and an atomic exchange construct. In the presence of a list construct, futures are sufficient to encode asynchronous channels. In λ_{ch}, we concentrate on asynchronous channels as primitive entities to better understand the correspondence with actors. Channel-based concurrent λ-calculi have been used as a formalism for the design of channel-based languages with session types, richer type systems which are expressive enough to encode protocols such as SMTP [15, 30].

Channel-based programming languages are inspired by CSP [21] and the π-calculus [32]; the name restriction and parallel composition operators in λ_{ch} and λ_{act} are directly inspired by analogous constructs in the π-calculus. Concurrent ML [39] extends Standard ML with a rich set of concurrency constructs centred around synchronous channels, which again, can emulate asynchronous channels. A core notion in Concurrent ML is nondeterministically synchronising on multiple synchronous events, such as sending or receiving messages; relating such a construct to an actor calculus is nontrivial, and remains an open problem.

The actor model was designed by Hewitt et al. [20] and examined in the context of distributed systems by Agha [2]. Agha describes an operational semantics on systems of actors, with a denotational interpretation of actor behaviours. Agha et al. [3] describe a formalism for a functional actor language, based on the λ-calculus. Their formalism consists of three core constructs: send sends a message; letactor creates a new actor; and become changes an actor's behaviour. While this system is closer to the actor model as originally envisaged as a model of concurrency, by using an explicit receive construct instead of using behaviours, our calculus—especially when extended with selective receive—more closely resembles Erlang (which the authors refer to as "essentially an actor language"). The semantics of this language are defined in terms of an actor configuration, consisting of a global actor mapping, a global multiset of messages, as well as a set of *receptionists* (names of actors which are externally visible to other configurations) and a set of external actor names. In contrast, our semantics for λ_{act} takes a form closer to the π-calculus, with visibility encoded via name restrictions and structural congruences. A strength of the work is that the authors consider behavioural theory in terms of operational equivalence and testing equivalence—something we have not investigated. Another difference is that we (following the work of He et al. [19] and libraries such as Akka Typed) consider a *typed* actor calculus, whereas this calculus is untyped.

Erlang [6] is a functional programming language supporting an actor-based style of concurrency, and has a strong reputation for supporting the development of highly-scalable

and fault-tolerant distributed systems [5]. In addition to the Akka framework, Scala has native support for actor-style concurrency, implemented efficiently without explicit virtual machine support [17]. Scala's native support employs Erlang-style selective receive. In the object-oriented setting, the actor model inspires *active objects* [28]: objects supporting asynchronous method calls which return responses using futures. De Boer et al. [10] describe a language and proof system for active objects with cooperatively scheduled threads within each object. Core ABS [27] is a specification language based on active objects. Using futures for synchronisation sidesteps the type pollution problem inherent in call-response patterns with actors, although our translations work in the absence of synchronisation. By working in the functional setting, we obtain more compact calculi.

Links [9] is a programming language designed for developing web applications which includes an implementation of typed message-passing concurrency built on an effect type system. The design of λ_{act} was inspired by Links.

Hopac [24] is a channel-based concurrency library for the F# programming language, based on Concurrent ML. The Hopac documentation includes a discussion of CML-style synchronous channels and actors [1], providing an implementation of actor-style concurrency primitives using channels, and an implementation of channel-style concurrency primitives using actors. The implementation of channels using actors uses shared-memory concurrency in the form of ML-style references in order to implement the `take` function, whereas our translation achieves this using message passing. Additionally, our translation is formalised and we prove that the translations are type- and semantics-preserving.

9 Conclusion

Inspired by languages such as Go which take channels as core constructs for communication, and languages such as Erlang which are based on the actor model of concurrency, we have presented translations back and forth between a concurrent λ-calculus λ_{ch} with channel-based communication constructs and a concurrent λ-calculus λ_{act} with actor-based communication constructs. We have proved that λ_{act} can simulate λ_{ch} and vice-versa.

The translation from λ_{act} to λ_{ch} is straightforward, whereas the translation from λ_{ch} to λ_{act} requires considerably more effort. Returning to Figure 2, this is unsurprising!

We have also shown how to extend λ_{act} with synchronisation, greatly simplifying the translation from λ_{ch} into λ_{act}, and have shown how Erlang-style selective receive can be emulated in λ_{act}. Additionally, we have discussed input-guarded choice in λ_{ch}, and how behavioural types may fit in with λ_{act}.

With the base calculi and metatheory in place, we look forward to the following areas of future work. First, we plan to strengthen our operational correspondence results by considering the completeness direction for both translations. Second, we plan to investigate how λ_{ch} with input-guarded nondeterministic choice can be emulated using λ_{act}. Finally, we intend to use the lessons learnt from studying theses calculi to inform the design of an actor-style language with behavioural types which will support static checking of conformance to communication patterns.

References

1 Actors and Hopac. https://www.github.com/Hopac/Hopac/blob/master/Docs/Actors.md, 2016.

2 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

3 Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.
4 Akka Typed. http://doc.akka.io/docs/akka/current/scala/typed.html, 2016.
5 Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.
6 Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.
7 Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP*. " O'Reilly Media, Inc.", 2016.
8 Avik Chaudhuri. A Concurrent ML Library in Concurrent Haskell. In *ICFP*, pages 269–280, New York, NY, USA, 2009. ACM.
9 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO*, volume 4709, pages 266–296. Springer Berlin Heidelberg, 2007.
10 Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330. Springer, 2007.
11 Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *AGERE*. ACM, 2016.
12 Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.
13 Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL*, pages 372–385. ACM Press, 1996.
14 Lars-Åke Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2001.
15 Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50, January 2010.
16 David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38. ACM, 1986.
17 Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.
18 Jiansen He. Type-parameterized actors and their supervision. MPhil thesis, The University of Edinburgh, 2014.
19 Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA*, pages 23–33. ACM, 2014.
20 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
21 C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8): 666–677, August 1978. ISSN 0001-0782.
22 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993.
23 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP*, chapter 9, pages 122–138. Springer Berlin Heidelberg, Berlin/Heidelberg, 1998.
24 Hopac. http://www.github.com/Hopac/hopac, 2016.
25 How are Akka actors different from Go channels? https://www.quora.com/How-are-Akka-actors-different-from-Go-channels, 2013.
26 Is Scala's actors similar to Go's coroutines? http://stackoverflow.com/questions/22621514/is-scalas-actors-similar-to-gos-coroutines, 2014.
27 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, pages 142–164. Springer, 2010.
28 R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

29 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.
30 Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In *ESOP*, pages 560–584. Springer, 2015.
31 Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
32 Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999.
33 Rüdiger Möller. Comparision of different concurrency models: Actors, CSP, Disruptor and Threads. `http://java-is-the-new-c.blogspot.com/2014/01/comparision-of-different-concurrency.html`, 2014.
34 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146. Springer, 2014.
35 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.
36 Luca Padovani and Luca Novara. Types for Deadlock-Free Higher-Order Programs. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE*, pages 3–18. Springer International Publishing, 2015.
37 Jennifer Paykin, Antal Spector-Zabusky, and Kenneth Foner. `choose` your own derivative. In *TyDe*, pages 58–59. ACM, 2016.
38 Proto Actor. `http://www.proto.actor`, 2016.
39 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.
40 Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.
41 Typed Actors. `https://github.com/knutwalker/typed-actors`, 2016.
42 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, 2014.

A Coalescing Transformation

Our translation from λ_{ch} into λ_{act} relies on the assumption that all channels have the same type, which is rarely the case in practice. Here, we sketch a sample type-directed transformation which we call *coalescing*, which transforms an arbitrary λ_{ch} program into an λ_{ch} program which has only one type of channel. To encode such a translation, we use an extension of the language with equirecursive types.

The transformation works by encapsulating each type of message in a variant type, and ensuring that give and take use the correct variant injections. Although the translation necessarily loses type information, thus introducing partiality, we can show that terms and configurations that are the result of the coalescing transformation never reduce to an error.

We say that a type A is a *base carried type* in a configuration $\Gamma; \Delta \vdash \mathcal{C}$ if there exists some subterm $\Gamma \vdash V : \mathsf{ChanRef}(A)$, where A is *not* of the form $\mathsf{ChanRef}(B)$.

In order to perform the coalescing transformation, we require an environment σ which maps each base carried type A to a unique token ℓ, which we use as an injection into a variant type.

We write $\sigma \smile \Gamma; \Delta \vdash \mathcal{C}$ if σ contains a bijective mapping $A \mapsto \ell$ for each base carried type in $\Gamma; \Delta \vdash \mathcal{C}$. We extend the relation analogously to judgements on values and computation terms.

Next, we define the notion of a *coalesced channel type*, which can be used to ensure that all channels in the system have the same type.

▶ **Definition 34** (Coalesced channel type).
Given a token environment $\sigma = A_0 \mapsto \ell_0, \dots A_n \mapsto \ell_n$, we define the *coalesced channel type* $\mathsf{cct}(\sigma)$ as

$$\mathsf{cct}(\sigma) = \mu X.\langle \ell_0 : \underline{A_0}, \dots, \ell_n : \underline{A_n}, \ell_c : \mathsf{ChanRef}(X) \rangle$$

where

$$\begin{aligned} \underline{\mathbf{1}} &= \mathbf{1} \\ \underline{A \to B} &= \underline{A} \to \underline{B} \\ \underline{A \times B} &= \underline{A} \times \underline{B} \end{aligned} \qquad \begin{aligned} \underline{A + B} &= \underline{A} + \underline{B} \\ \underline{\mathsf{List}(A)} &= \mathsf{List}(\underline{A}) \\ \underline{\mathsf{ChanRef}(A)} &= \mathsf{ChanRef}(X) \\ \underline{\mu X.A} &= \mu X.\underline{A} \end{aligned}$$

which is the single channel type which can receive values of all possible types sent in the system.

Note that the definition of a base carried type excludes the possibility of a type of the form $\mathsf{ChanRef}(A)$ appearing in σ. To handle the case of sending channels, we require $\mathsf{cct}(\sigma)$ to be a recursive type; a distinguished token ℓ_c denotes the variant case for sending a channel over a channel.

Retrieving a token from the token environment σ is defined by the following inference rules. Note that $\mathsf{ChanRef}(A)$ maps to the distinguished token ℓ_c.

$$\frac{(A \mapsto \ell) \in \sigma}{\sigma(A) = \ell} \qquad \frac{}{\sigma(\mathsf{ChanRef}(A)) = \ell_c}$$

With the coalesced channel type defined, we can define a function mapping types to coalesced types.

▶ **Definition 35** (Type coalescing).

$$
\begin{array}{rcl}
|\mathbf{1}|^\sigma & = & \mathbf{1} \\
|A \to B|^\sigma & = & |A|^\sigma \to |B|^\sigma \\
|A \times B|^\sigma & = & |A|^\sigma \times |B|^\sigma
\end{array}
\qquad
\begin{array}{rcl}
|A + B|^\sigma & = & |A|^\sigma + |B|^\sigma \\
|\mathsf{List}(A)|^\sigma & = & \mathsf{List}(|A|^\sigma) \\
|\mathsf{ChanRef}(A)|^\sigma & = & \mathsf{ChanRef}(\mathsf{cct}(\sigma)) \\
|\mu X.A|^\sigma & = & \mu X.|A|^\sigma
\end{array}
$$

We then extend $|-|^\sigma$ to typing environments Γ, taking into account that we must annotate channel names.

▶ **Definition 36** (Type coalescing on environments). **1.** For Γ:

a. $|\emptyset|^\sigma = \emptyset$
b. $|x : A, \Gamma|^\sigma = x : |A|^\sigma, |\Gamma|^\sigma$.
c. $|a : \mathsf{ChanRef}(A), \Gamma|^\sigma = a_A : \mathsf{ChanRef}(\mathsf{cct}(\sigma)), |\Gamma|^\sigma$.

2. For Δ:

a. $|\emptyset|^\sigma = \emptyset$
b. $|a : A, \Delta|^\sigma = a_A : \mathsf{cct}(\sigma), |\Delta|^\sigma$

Figure 20 describes the coalescing pass from λ_{ch} with multiple channel types into λ_{ch} with a single channel type. Judgements are of the shape $\{\sigma\}\ \Gamma \vdash V : A \rightsquigarrow V'$ for values; $\{\sigma\}\ \Gamma \vdash M : A \rightsquigarrow M'$ for computations; and $\{\sigma\}\ \Gamma \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'$ for configurations, where σ is an bijective mapping from types to tokens, and primed values are the results of the coalescing pass. We omit the rules for values and functional terms, which are homomorphisms.

Of particular note are the rules for give and take. The coalesced version of give ensures that the correct token is used to inject into the variant type. The translation of take retrieves a value from the channel, and pattern matches to retrieve a value of the correct type from the variant. As we have less type information, we have to account for the possibility that pattern matching fails by introducing an error term, which we define at the top-level of the term:

$$\mathsf{let}\,\mathtt{error} = (\mathsf{rec}\, f(x)\,.\, f\ x)\ \mathsf{in}\ \ldots$$

The translation on configurations ensures that all existing values contained within a buffer are wrappen in the appropriate variant injection.

The coalescing step necessarily loses typing information on channel types. To aid us in stating an error-freedom result, we annotate channel names a with their original type; for example, a channel with name a carrying values of type A would be translated as a_A. It is important to note that annotations are irrelevant to reduction, i.e.:

$$E[\mathsf{give}\, a_A\, W] \parallel a_B(\overrightarrow{V}) \longrightarrow E[\mathsf{return}\,()] \parallel a_B(\overrightarrow{V} \cdot W)$$

As previously discussed, the coalescing pass means that channel types are less specific, with the pass introducing partiality in the form of an error term, $\mathtt{error}$. However, since we began with a type-safe program in λ_{ch}, we can show that programs that have been coalesced from well-typed λ_{ch} configurations never reduce to an error term.

▶ **Definition 37** (Error configuration).
A configuration $\mathcal{C}$ is an *error configuration* if $\mathcal{C} \equiv G[\mathtt{error}]$ for some configuration context G.

Coalescing of channel names $\boxed{\{\sigma\}\ \Gamma \vdash V \rightsquigarrow V'}$

Ref
$$\frac{a : \mathsf{ChanRef}(A) \in \Gamma}{\{\sigma\}\ \Gamma \vdash a : \mathsf{ChanRef}(A) \rightsquigarrow a_A}$$

Coalescing of communication and concurrency primitives $\boxed{\{\sigma\}\ \Gamma \vdash M \rightsquigarrow M'}$

Give
$$\frac{\{\sigma\}\ \Gamma \vdash V : A \rightsquigarrow V' \qquad \{\sigma\}\ \Gamma \vdash W : \mathsf{ChanRef}(A) \rightsquigarrow W' \quad \sigma(A) = \ell}{\{\sigma\}\ \Gamma \vdash \mathsf{give}\, V\, W : \mathbf{1} \rightsquigarrow \mathsf{give}\,(\mathsf{roll}\,\langle \ell = V' \rangle)\, W'}$$

Take
$$\frac{\{\sigma\}\ \Gamma \vdash V : \mathsf{ChanRef}(A) \rightsquigarrow V' \quad \sigma(A) = \ell_j}{\{\sigma\}\ \Gamma \vdash \mathsf{take}\, V : A \rightsquigarrow \begin{array}{l} \mathsf{let}\ x \Leftarrow \mathsf{take}\, V'\ \mathsf{in} \\ \mathsf{let}\ y \Leftarrow \mathsf{unroll}\ x\ \mathsf{in} \\ \mathsf{case}\ y\ \{ \\ \quad \langle \ell_0 = y \rangle \ldots \langle \ell_{j-1} = y \rangle \mapsto \mathtt{error} \\ \quad \langle \ell_j = y \rangle \mapsto y \\ \quad \langle \ell_{j+1} = y \rangle \ldots \langle \ell_n = y \rangle \mapsto \mathtt{error} \quad \} \end{array}}$$

NewCh
$$\frac{}{\{\sigma\}\ \Gamma \vdash \mathsf{newCh} : \mathsf{ChanRef}(A) \rightsquigarrow \mathsf{newCh}_A}$$

Fork
$$\frac{\{\sigma\}\ \Gamma \vdash M : \mathbf{1} \rightsquigarrow M'}{\{\sigma\}\ \Gamma \vdash \mathsf{fork}\, M : \mathbf{1} \rightsquigarrow \mathsf{fork}\, M'}$$

Coalescing of configurations $\boxed{\{\sigma\}\ \Gamma \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'}$

Par
$$\frac{\{\sigma\}\ \Gamma; \Delta \vdash \mathcal{C}_1 \rightsquigarrow \mathcal{C}_1' \quad \{\sigma\}\ \Gamma; \Delta \vdash \mathcal{C}_2 \rightsquigarrow \mathcal{C}_2'}{\{\sigma\}\ \Gamma; \Delta \vdash \mathcal{C}_1 \parallel \mathcal{C}_2 \rightsquigarrow \mathcal{C}_1' \parallel \mathcal{C}_2'}$$

Chan
$$\frac{\{\sigma\}\ \Gamma, a : \mathsf{ChanRef}(A); \Delta, a : A \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'}{\{\sigma\}\ \Gamma; \Delta \vdash (\nu a)\mathcal{C} \rightsquigarrow (\nu a_A)\mathcal{C}'}$$

Term
$$\frac{\{\sigma\}\ \Gamma \vdash M : A \rightsquigarrow M'}{\{\sigma\}\ \Gamma; \cdot \vdash M \rightsquigarrow M'}$$

Buf
$$\frac{(\{A, \sigma\}\ \Gamma \vdash V_i : A \rightsquigarrow V_i')_i}{\{\sigma\}\ \Gamma; a : A \vdash a(\overrightarrow{V}) \rightsquigarrow a_A(\overrightarrow{V'})}$$

Coalescing of buffer values $\boxed{\{A, \sigma\}\ \Gamma \vdash V : A \rightsquigarrow V'}$

$$\frac{\{\sigma\}\ \Gamma \vdash V : A \rightsquigarrow V' \quad \sigma(A) = \ell}{\{A, \sigma\}\ \Gamma \vdash V : A \rightsquigarrow \langle \ell = V \rangle}$$

Figure 20 Type-directed coalescing pass

▶ **Definition 38** (Error-free configuration).
A configuration $\mathcal{C}$ is *error-free* if it is not an error configuration.

We can straightforwardly see that the initial result of a coalescing pass is error-free:

▶ **Lemma 39.** *If $\sigma \smile \Gamma \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'$, then $\mathcal{C}'$ is error-free.*

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'$. ◀

Next, we show that error-freedom is preserved under reduction. To do so, we make essential use of the fact that the coalescing pass annotates each channel with its original type.

▶ **Lemma 40** (Error-freedom (coalesced λ_{ch})).
If $\Gamma; \Delta \vdash \mathcal{C} \rightsquigarrow \mathcal{C}'_1$, and $\mathcal{C}'_1 \longrightarrow^* \mathcal{C}'_2$, then $\mathcal{C}'_2$ is error free.

Proof. By preservation in λ_{ch}, we have that $\Gamma; \Delta \vdash \mathcal{C}'_2$, and by Lemma 39, we can assume that $\mathcal{C}'_1$ is error-free.

We show that an error term can never arise. Suppose that $\mathcal{C}'_2$ was an error configuration, meaning that $\mathcal{C}'_2 \equiv G[\texttt{error}]$ for some configuration context G. As we have decreed that the `error` term does not appear in user programs, we know that `error` must have arisen from the refinement pass. By observation of the refinement rules, we see that `error` is introduced only in the refinement rule for Take.

Stepping backwards through the reduction sequence introduced by the Take rule, we have that:

$$
\begin{array}{l}
\mathsf{case}\ \langle \ell_k = W \rangle\ \{ \\
\quad \langle \ell_0 = y \rangle \ldots \langle \ell_{j-1} = y \rangle \mapsto \texttt{error} \\
\quad \langle \ell_j = y \rangle \mapsto y \\
\quad \langle \ell_{j+1} = y \rangle \ldots \langle \ell_n = y \rangle \mapsto \texttt{error} \\
\}
\end{array}
$$

for some $k \neq j$.
Stepping back further, we have that:

$$
\begin{array}{l}
\mathsf{let}\ x \Leftarrow \mathsf{take}\ a_B\ \mathsf{in} \\
\mathsf{let}\ y \Leftarrow \mathsf{unroll}\ (\mathsf{roll}\ x)\ \mathsf{in} \\
\mathsf{case}\ y\ \{ \\
\quad \langle \ell_0 = y \rangle \ldots \langle \ell_{j-1} = y \rangle \mapsto \texttt{error} \\
\quad \langle \ell_j = y \rangle \mapsto y \\
\quad \langle \ell_{j+1} = y \rangle \ldots \langle \ell_n = y \rangle \mapsto \texttt{error} \\
\}
\end{array}
$$

Now, inspecting the premises of the refinement rule for Take, we have that $\Gamma \vdash a_B : \mathsf{ChanRef}(A) \rightsquigarrow V'$ and $\sigma(A) = \ell_j$. Examining the refinement rule for Name, we have that $\Gamma; \Psi \vdash a : \mathsf{ChanRef}(A) \rightsquigarrow a_A$, thus we have that $B = A$.

However, we have that $\sigma(A) = \ell_j$ and $\sigma(A) = \ell_k$ but we know that $k \neq j$, thus leading to a contradiction since σ is bijective. ◀

Since annotations are irrelevant to reduction, it follows that $\mathcal{C}'$ has identical reduction behaviour with all annotations erased.

B Selected full proofs

B.1 λ_{ch} Preservation

▶ **Lemma 41** (Replacement). *If* $\Gamma \vdash E[M] : A$, $\Gamma \vdash M : B$, *and* $\Gamma \vdash N : B$, *then* $\Gamma \vdash E[N] : A$.

Proof. By induction on the structure of E. ◀

Theorem 4 (Preservation (λ_{ch} configurations)
If $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\Gamma; \Delta \vdash \mathcal{C}_2$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{C}'$. We use Lemma 41 implicitly throughout.
Case Give
From the assumption $\Gamma; \Delta \vdash E[\mathsf{give}\, W\, a] \parallel a(\overrightarrow{V})$, we have that $\Gamma; \cdot \vdash E[\mathsf{give}\, W\, a]$ and $\Gamma; a : A \vdash a(\overrightarrow{V})$. Consequently, we know that $\Delta = a : A$.

From this, we know that $\Gamma \vdash \mathsf{give}\, W\, a : \mathbf{1}$ and thus $\Gamma \vdash W : A$ and $\Gamma \vdash a : \mathsf{ChanRef}(A)$. We also have that $\Gamma; a : A \vdash a(\overrightarrow{V})$, thus $\Gamma \vdash V_i : A$ for all $V_i \in \overrightarrow{V}$. By Unit we can show $\Gamma; \cdot \vdash E[\mathsf{return}\, ()]$ and by Buf we can show $\Gamma; a : A \vdash a(\overrightarrow{V} \cdot W)$; recomposing, we arrive at $\Gamma; \Delta \vdash E[\mathsf{return}\, ()] \parallel a(\overrightarrow{V} W)$ as required.
Case Take
From the assumption $\Gamma; \Delta \vdash E[\mathsf{take}\, a] \parallel a(W \cdot \overrightarrow{V})$, we have that $\Gamma; \cdot \vdash E[\mathsf{take}\, a]$ and that $\Gamma; a : A \vdash a(W \cdot \overrightarrow{V})$. Consequently, we know that $\Delta = a : A$.

From this, we know that $\Gamma \vdash \mathsf{take}\, a : A$, and thus $\Gamma \vdash a : \mathsf{ChanRef}(A)$. Similarly, we have that $\Gamma; a : \mathsf{ChanRef}(A) \vdash a(W \cdot \overrightarrow{V})$, and thus $\Gamma \vdash W : A$.

Consequently, we can show that $\Gamma; \cdot \vdash E[\mathsf{return}\, W]$ and $\Gamma; a : A \vdash a(\overrightarrow{V})$; recomposing, we arrive at $\Gamma; \Delta \vdash E[\mathsf{return}\, W] \parallel a(\overrightarrow{V})$ as required.
Case NewCh
By Buf we can type $\Gamma; a : A \vdash a(\epsilon)$, and since $\Gamma \vdash \mathsf{newCh} : \mathsf{ChanRef}(A)$, it is also possible to show $\Gamma, a : \mathsf{ChanRef}(A) \vdash a : \mathsf{ChanRef}(A)$, thus $\Gamma, a : \mathsf{ChanRef}(A) \vdash E[\mathsf{return}\, a]$.

Recomposing by Par we have $\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash E[\mathsf{return}\, a] \parallel a(\epsilon)$, and by Chan we have $\Gamma; \cdot \vdash (\nu a)(E[\mathsf{return}\, a] \parallel a(\epsilon))$ as required.
Case Fork
From the assumption $\Gamma; \Delta \vdash E[\mathsf{fork}\, M]$, we have that $\Delta = \emptyset$ and $\Gamma \vdash M : \mathbf{1}$. By Unit we can show $\Gamma; \cdot \vdash E[\mathsf{return}\, ()]$, and by Term we can show $\Gamma; \cdot \vdash M$. Recomposing, we arrive at $\Gamma; \Delta \vdash E[\mathsf{return}\, ()] \parallel M$ as required.
Case Lift Immediate by the inductive hypothesis.
Case LiftV Immediate by Lemma 1. ◀

B.2 λ_{act} Preservation

▶ **Lemma 42** (Replacement). *If* $\Gamma \mid C \vdash E[M] : A$*,* $\Gamma \mid C \vdash M : B$*, and* $\Gamma \mid C \vdash N : B$*, then* $\Gamma \mid C \vdash E[N] : B$*.*

Proof. By induction on the structure of E. ◀

Theorem 12 (Preservation (λ_{act} configurations))
If $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then* $\Gamma; \Delta \vdash \mathcal{C}_2$*.*

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, making implicit use of Lemma 42.
Case Spawn
From the assumption that $\Gamma; \Delta \vdash \langle a, E[\mathsf{spawn}\, M], \vec{V} \rangle$, we have that $\Delta = a : C$, that $\Gamma \mid C \vdash \mathsf{spawn}\, M : \mathsf{ActorRef}(A)$ and $\Gamma \mid A \vdash M : \mathbf{1}$.

We can show $\Gamma, b : \mathsf{ActorRef}(A) \vdash b : \mathsf{ActorRef}(A)$; and therefore that $\Gamma, b : \mathsf{ActorRef}(A) \vdash E[\mathsf{return}\, b] : \mathbf{1}$. By Actor, it follows that $\Gamma, b : \mathsf{ActorRef}(A); a : C \vdash \langle a, E[\mathsf{return}\, b], \vec{V} \rangle$.

By Actor, we can show $\Gamma, b : \mathsf{ActorRef}(A); b : A \vdash \langle b, M, \epsilon \rangle$.

Finally, by Pid and Par, we have $\Gamma; \Delta \vdash (\nu b)(\langle a, E[\mathsf{return}\, b], \vec{V} \rangle \parallel \langle b, M, \epsilon \rangle)$ as required.
Case Send
From the assumption that $\Gamma; \Delta \vdash \langle a, E[\mathsf{send}\, V'\, b], \vec{V} \rangle \parallel \langle b, M, \vec{W} \rangle$, we have that $\Gamma; a : A \vdash \langle a, E[\mathsf{send}\, V'\, b], \vec{V} \rangle$ and $\Gamma; b : C \vdash \langle b, M, \vec{W} \rangle$. Consequently, we can write $\Delta = a : A, b : C$. From this, we know that $\Gamma \mid A \vdash \mathsf{send}\, V'\, b : \mathbf{1}$, so we can write $\Gamma = \Gamma', b : \mathsf{ActorRef}(C)$, and $\Gamma \vdash V' : C$. Additionally, we know that $\Gamma; b : C \vdash \langle b, M, \vec{W} \rangle$ and thus that $(\Gamma \vdash W_i : C)$ for each entry $W_i \in \vec{W}$.

As $\Gamma \vdash V' : C$, it follows that $\Gamma \vdash \vec{W} \cdot V'$ and therefore that $\Gamma; b : C \vdash \langle b, M, \vec{W} \cdot V \rangle$. We can also show that $\Gamma \mid C \vdash \mathsf{return}\, () : \mathbf{1}$, and therefore it follows that $\Gamma; a : A \vdash \langle a, E[\mathsf{return}\, ()], \vec{V} \rangle$.

Recomposing, we have that $\Gamma; \Delta \vdash \langle a, E[\mathsf{return}\, ()], \vec{W} \rangle \parallel \langle b, M, \vec{W} \cdot V' \rangle$ as required.
Case Receive
By Actor, we have that $\Gamma; \Delta \vdash \langle a, E[\mathsf{receive}], W \cdot \vec{V} \rangle$. From this, we know that $\Gamma \mid A \vdash E[\mathsf{receive}] : \mathbf{1}$ (and thus $\Gamma \mid A \vdash \mathsf{receive} : A$) and $\Gamma \vdash W : A$.

Consequently, we can show $\Gamma \vdash E[\mathsf{return}\, W] : \mathbf{1}$. By Actor, we arrive at $\Gamma; \Delta \vdash \langle a, E[\mathsf{return}\, W], \vec{V} \rangle$ as required.
Case Self
By Actor, we have that $\Gamma; \Delta \vdash \langle a, E[\mathsf{self}], \vec{V} \rangle$, and thus that $\Gamma \mid A \vdash E[\mathsf{self}] : \mathbf{1}$ and $\Gamma \mid A \vdash \mathsf{self} : \mathsf{ActorRef}(A)$. We also know that $\Gamma = \Gamma', a : \mathsf{ActorRef}(A)$, and that $\Delta = a : A$.

Trivially, we can show $\Gamma', a : \mathsf{ActorRef}(A) \vdash a : \mathsf{ActorRef}(A)$. Thus it follows that $\Gamma', a : \mathsf{ActorRef}(A) \vdash E[\mathsf{return}\, a] : \mathbf{1}$ and thus it follows that $\Gamma; \Delta \vdash \langle a, E[\mathsf{return}\, a], \vec{V} \rangle$ as required.
Case Lift Immediate by the inductive hypothesis.
Case LiftV Immediate by Lemma 9. ◀

B.3 Translation: λ_{act} into λ_{ch}

Lemma 23

1. *If* $\Gamma \vdash V : A$ *in* λ_{act}, *then* $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!]$ *in* λ_{ch}.
2. *If* $\Gamma \mid B \vdash M : A$ *in* λ_{act}, *then* $[\![\Gamma]\!], \alpha : \mathsf{ChanRef}([\![B]\!]) \vdash [\![M]\!]\ \alpha : [\![A]\!]$ *in* λ_{ch}.

Proof. By simultaneous induction on the derivations of $\Gamma \vdash V : A$ and $\Gamma \mid B \vdash M : A$.

Premise 1

Case $\Gamma \vdash x : A$
By the definition of $[\![-]\!]$, we have that $[\![\alpha]\!] = \alpha$. By the definition of $[\![\Gamma]\!]$, we have that $\alpha : [\![A]\!] \in [\![\Gamma]\!]$. Consequently, it follows that $[\![\Gamma]\!] \vdash \alpha : [\![A]\!]$.
Case $\Gamma \vdash \lambda x.M : A \to B$
From the assumption that $\Gamma \vdash \lambda x.M : A \to^C B$, we have that $\Gamma, x : A \mid C \vdash M : B$. By the inductive hypothesis (premise 2), we have that $[\![\Gamma]\!], x : [\![A]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![M]\!]\ ch : [\![B]\!]$.

By two applications of Abs, we have $[\![\Gamma]\!] \vdash \lambda x.\lambda ch.[\![M]\!]\ ch : [\![A]\!] \to \mathsf{ChanRef}([\![C]\!]) \to [\![B]\!]$ as required.
Case $\Gamma \vdash () : \mathbf{1}$ Immediate.
Case $\Gamma \vdash (V, W) : (A \times B)$
From the assumption that $\Gamma \vdash (V, W) : (A, B)$ we have that $\Gamma \vdash V : A$ and $\Gamma \vdash W : B$. By the inductive hypothesis (premise 1) and Pair, we can show $[\![\Gamma]\!] \vdash ([\![V]\!], [\![W]\!]) : ([\![A]\!] \times [\![B]\!])$ as required.

Premise 2

Case $\Gamma \mid C \vdash V\,W : B$
From the assumption that $\Gamma \mid C \vdash V\,W : B$, we have that $\Gamma \vdash V : A \to^C B$ and $\Gamma \vdash W : B$. By the inductive hypothesis (premise 1), we have that $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!] \to \mathsf{ChanRef}([\![C]\!]) \to [\![B]\!]$, and $[\![\Gamma]\!] \vdash [\![W]\!] : [\![B]\!]$.

By extending the context Γ with a $ch : \mathsf{ChanRef}([\![C]\!])$, we can show that $[\![\Gamma]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![V]\!]\,[\![W]\!]\ ch : [\![B]\!]$ as required.
Case $\Gamma \mid C \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B$
From the assumption that $\Gamma \mid C \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B$, we have that $\Gamma \mid C \vdash M : A$ and that $\Gamma, x : A \mid C \vdash N : B$.

By the inductive hypothesis (premise 2), we have that $[\![\Gamma]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![M]\!]\ ch : [\![A]\!]$ and $[\![\Gamma]\!], x : [\![A]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![N]\!]\ ch : [\![B]\!]$.

By EffLet, it follows that $[\![\Gamma]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash \mathsf{let}\ x \Leftarrow [\![M]\!]\ ch\ \mathsf{in}\ [\![N]\!]\ ch : [\![B]\!]$ as required.
Case $\Gamma \mid C \vdash \mathsf{return}\,V : A$
From the assumption that $\Gamma \mid C \vdash \mathsf{return}\,V : A$, we have that $\Gamma \vdash V : A$.

By the inductive hypothesis (premise 1), we have that $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!]$.

By weakening (as we do not use the mailbox channel), we can show that $[\![\Gamma]\!], y : \mathsf{ChanRef}([\![C]\!]) \vdash \mathsf{return}\,[\![V]\!] : [\![A]\!]$ as required.
Case $\Gamma \mid C \vdash \mathsf{send}\,V\,W : \mathbf{1}$
From the assumption that $\Gamma \mid C \vdash \mathsf{send}\,V\,W : \mathbf{1}$, we have that $\Gamma \vdash V : A$ and $\Gamma \vdash W : \mathsf{ChanRef}(A)$.

By the inductive hypothesis (premise 1) we have that $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!]$ and $[\![\Gamma]\!] \vdash [\![W]\!] : \mathsf{ChanRef}([\![A]\!])$.

By Give, we can show that $[\![\Gamma]\!] \vdash \mathsf{give}\,[\![V]\!]\,[\![W]\!] : \mathbf{1}$, and by weakening we have that $[\![\Gamma]\!], y : \mathsf{ChanRef}([\![C]\!]) \vdash \mathsf{give}\,[\![V]\!]\,[\![W]\!] : \mathbf{1}$ as required.

Case $\Gamma \mid C \vdash \mathsf{receive} : C$
Given a $ch : \mathsf{ChanRef}(\llbracket C \rrbracket)$, we can show that $\llbracket \Gamma \rrbracket, ch : \mathsf{ChanRef}(\llbracket C \rrbracket) \vdash ch : \mathsf{ChanRef}(\llbracket C \rrbracket)$ and therefore that $\llbracket \Gamma \rrbracket, ch : \mathsf{ChanRef}(\llbracket C \rrbracket) \vdash \mathsf{take}\ ch : \llbracket C \rrbracket$ as required.
Case $\Gamma \mid C \vdash \mathsf{spawn}\ M : \mathsf{ActorRef}(A)$
From the assumption that $\Gamma \mid C \vdash \mathsf{spawn}\ M : \mathsf{ActorRef}(A)$, we have that $\Gamma \mid A \vdash M : A$.

By the inductive hypothesis (premise 2), we have that $\llbracket \Gamma \rrbracket, chMb : \mathsf{ChanRef}(\llbracket A \rrbracket) \vdash \llbracket M \rrbracket\ chMb : \llbracket A \rrbracket$. By Fork and Return, we can show that $\llbracket \Gamma \rrbracket, chMb : \mathsf{ChanRef}(\llbracket A \rrbracket) \vdash \mathsf{fork}\ (\llbracket M \rrbracket\ chMb); \mathsf{return}\ chMb : \mathsf{ChanRef}(\llbracket A \rrbracket)$.

By NewCh and EffLet, we can show that $\llbracket \Gamma \rrbracket \vdash \mathsf{let}\ chMb \Leftarrow \mathsf{newCh}\ \mathsf{in}\ \mathsf{fork}\ (\llbracket M \rrbracket\ chMb); \mathsf{return}\ chMb : \mathsf{ChanRef}(\llbracket A \rrbracket)$.

Finally, by weakening, we have that

$$\llbracket \Gamma \rrbracket, ch : \mathsf{ChanRef}(\llbracket C \rrbracket) \vdash \mathsf{let}\ chMb \Leftarrow \mathsf{newCh}\ \mathsf{in}\ (\mathsf{fork}\ \llbracket M \rrbracket\ chMb); \mathsf{return}\ chMb : \mathsf{ChanRef}(\llbracket A \rrbracket)$$

as required.
Case $\Gamma \mid C \vdash \mathsf{self} : \mathsf{ActorRef}(C)$
Given a $ch : \mathsf{ChanRef}(\llbracket C \rrbracket)$, we can show that $\llbracket \Gamma \rrbracket, ch : \mathsf{ChanRef}(\llbracket C \rrbracket) \vdash \mathsf{return}\ ch : \mathsf{ChanRef}(\llbracket C \rrbracket)$ as required.

◀

Theorem 18 *If* $\Gamma; \Delta \vdash \mathcal{C}$, *then* $\llbracket \Gamma \rrbracket; \llbracket \Delta \rrbracket \vdash \llbracket \mathcal{C} \rrbracket$.

Proof. By induction on the derivation of $\Gamma \vdash \mathcal{C}$.
Case Par
From the assumption that $\Gamma; \Delta \vdash \mathcal{C}_1 \parallel \mathcal{C}_2$, we have that Δ splits as Δ_1, Δ_2 such that $\Gamma; \Delta_1 \vdash \mathcal{C}_1$ and $\Gamma; \Delta_2 \vdash \mathcal{C}_2$. By the inductive hypothesis, we have that $\llbracket \Gamma \rrbracket; \llbracket \Delta_1 \rrbracket \vdash \llbracket \mathcal{C}_1 \rrbracket$ and $\llbracket \Gamma \rrbracket; \llbracket \Delta_2 \rrbracket \vdash \llbracket \mathcal{C}_2 \rrbracket$. Recomposing by Par, we have that $\llbracket \Gamma \rrbracket; \llbracket \Delta_1 \rrbracket, \llbracket \Delta_2 \rrbracket \vdash \llbracket \mathcal{C}_1 \rrbracket \parallel \llbracket \mathcal{C}_2 \rrbracket$ as required.
Case Pid
From the assumption that $\Gamma; \Delta \vdash (\nu a)\mathcal{C}$, we have that $\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}$. By the inductive hypothesis, we have that $\llbracket \Gamma \rrbracket, a : \mathsf{ChanRef}(\llbracket A \rrbracket); \llbracket \Delta \rrbracket, a : \llbracket A \rrbracket \vdash \llbracket \mathcal{C} \rrbracket$. Recomposing by Pid, we have that $\llbracket \Gamma \rrbracket \vdash (\nu a)\llbracket \mathcal{C} \rrbracket$ as required.
Case Actor
From the assumption that $\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \vec{V} \rangle$, we have that $\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : \mathbf{1}$. By Lemma 17, we have that $\llbracket \Gamma \rrbracket, a : \mathsf{ChanRef}(\llbracket A \rrbracket) \vdash \llbracket M \rrbracket\ a : \mathbf{1}$. It follows straightforwardly that $\llbracket \Gamma \rrbracket, a : \mathsf{ChanRef}(\llbracket A \rrbracket); \cdot \vdash \llbracket M \rrbracket\ a$.

We can also show that $\llbracket \Gamma \rrbracket, a : \mathsf{ChanRef}(\llbracket A \rrbracket); a : \llbracket A \rrbracket \vdash a(\llbracket \vec{V} \rrbracket)$ (where $\llbracket \vec{V} \rrbracket = \llbracket V_1 \rrbracket \cdot \ldots \cdot \llbracket V_n \rrbracket$), by repeated applications of Lemma 17.

By Term and Par, we have that $\llbracket \Gamma \rrbracket, a : \mathsf{ChanRef}(\llbracket A \rrbracket); a : \llbracket A \rrbracket \vdash a(\llbracket \vec{V} \rrbracket) \parallel \llbracket M \rrbracket\ a$ as required.

◀

Theorem 21
If $\Gamma \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then there exists some* $\mathcal{D}$ *such that* $\llbracket \mathcal{C}_1 \rrbracket \longrightarrow^* \mathcal{D}$, *with* $\mathcal{D} \equiv \llbracket \mathcal{C}_2 \rrbracket$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{C}'$.
Case Spawn

Assumption $\quad \llbracket \langle a, E[\mathsf{spawn}\ M], \vec{V} \rangle \rrbracket \quad (1)$

Definition of $[\![-]\!]$	$a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{let}\ c \Leftarrow \mathsf{newCh}\ \mathsf{in}\ \mathsf{fork}\ ([\![M]\!]\ c); \mathsf{return}\ c]\ a)$	(2)
newCh reduction	$a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathsf{let}\ c \Leftarrow \mathsf{return}\ b\ \mathsf{in}\ \mathsf{fork}\ ([\![M]\!]\ c); \mathsf{return}\ c]\ a) \parallel b(\epsilon))$	(3)
Let reduction	$a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathsf{fork}\ ([\![M]\!]\ b); \mathsf{return}\ b]\ a) \parallel b(\epsilon))$	(4)
Fork reduction	$a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathsf{return}\ (); \mathsf{return}\ b]\ a) \parallel ([\![M]\!]\ b) \parallel b(\epsilon))$	(5)
Let reduction	$a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathsf{return}\ b]\ a) \parallel ([\![M]\!]\ b) \parallel b(\epsilon))$	(6)
$\equiv$	$(\nu b)(a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{return}\ b]\ a) \parallel b(\epsilon) \parallel ([\![M]\!]\ b))$	(7)
$=$	$[\![(\nu b)(\langle a, E[\mathsf{return}\ b], \overrightarrow{V} \rangle \parallel \langle b, M, \epsilon \rangle)]\!]$	(8)

Case Self

Assumption	$[\![\langle a, E[\mathsf{self}], \overrightarrow{V} \rangle]\!]$	(1)
Definition of $[\![-]\!]$	$a(([\![\overrightarrow{V}]\!])) \parallel ([\![E]\!][\mathsf{return}\ a]\ a)$	(2)
$=$	$[\![\langle a, E[\mathsf{return}\ a], \overrightarrow{V} \rangle]\!]$	

Case Send

Assumption	$[\![\langle a, E[\mathsf{send}\ V'\ b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle]\!]$	(1)
Definition of $[\![-]\!]$	$a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{give}\ [\![V']\!]\ b]\ a) \parallel b([\![\overrightarrow{W}]\!]) \parallel ([\![M]\!]\ b)$	(2)
Give reduction	$a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{return}\ ()]\ a) \parallel b([\![\overrightarrow{W}]\!] \cdot [\![V']\!]) \parallel ([\![M]\!]\ b)$	(3)
$=$	$[\![\langle a, E[\mathsf{return}\ ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle]\!]$	(4)

Case Receive

Assumption	$[\![\langle a, E[\mathsf{receive}], W \cdot \overrightarrow{V} \rangle]\!]$	(1)
Definition of $[\![-]\!]$	$a([\![W]\!] \cdot [\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{take}\ a]\ a)$	(2)
Take reduction	$a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathsf{return}\ [\![W]\!]]\ a)$	(3)
$=$	$[\![\langle a, E[\mathsf{return}\ W], \overrightarrow{V} \rangle]\!]$	(4)

Lift is immediate from the inductive hypothesis, and LiftV is immediate from Lemma 19.

B.4 Translation: λ_{ch} into λ_{act}

Lemma 23

1. *If* $\{B\}\ \Gamma \vdash V : A$*, then* $(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!)$.
2. *If* $\{B\}\ \Gamma \vdash M : A$*, then* $(\!|\Gamma|\!) \mid (\!|B|\!) \vdash (\!|M|\!) : (\!|A|\!)$.

Proof. By simultaneous induction on the derivations of $\{B\}\ \Gamma \vdash V : A$ and $\{B\}\ \Gamma \vdash M : A$.
Premise 1

Case Var
From the assumption that $\{B\}\ \Gamma \vdash \alpha : A$, we know that $\alpha : A \in \Gamma$. By the definition of $(\!|\Gamma|\!)$, we have that $\alpha : (\!|A|\!) \in (\!|\Gamma|\!)$. Since $(\!|\alpha|\!) = \alpha$, it follows that $(\!|\Gamma|\!) \vdash \alpha : (\!|A|\!)$ as required.
Case Abs
From the assumption that $\{C\}\ \Gamma \vdash \lambda x.M : A \to B$, we have that $\{C\}\ \Gamma, x : A \vdash M : B$. By the inductive hypothesis (Premise 2), we have that $(\!|\Gamma|\!), x : (\!|A|\!) \mid (\!|C|\!) \vdash (\!|M|\!) : (\!|A|\!)$. By Abs, we can show that $(\!|\Gamma|\!) \mid (\!|C|\!) \vdash \lambda x.(\!|M|\!) : (\!|A|\!) \to^{(\!|C|\!)} (\!|B|\!)$ as required.
Case Rec Similar to Abs.
Case Unit Immediate.
Case Pair
From the assumption that $\{C\}\ \Gamma \vdash (V, W) : A \times B$, we have that $\{C\}\ \Gamma \vdash V : A$ and that $\{C\}\ \Gamma \vdash W : B$.

By the inductive hypothesis (premise 1), we have that $(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!)$ and that $(\!|\Gamma|\!) \vdash (\!|W|\!) : (\!|B|\!)$.

It follows by Pair that $(\!|\Gamma|\!) \vdash ((\!|V|\!), (\!|W|\!)) : ((\!|A|\!) \times (\!|B|\!))$ as required.
Cases Inl, Inr Similar to Pair.
Case Roll Immediate.
Premise 2

Case App
From the assumption that $\{C\}\ \Gamma \vdash V\,W : B$, we have that $\{C\}\ \Gamma \vdash V : A \to B$ and $\{C\}\ \Gamma \vdash W : A$. By the inductive hypothesis (premise 1), we have that $(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!) \to^{(\!|C|\!)} (\!|B|\!)$ and that $(\!|\Gamma|\!) \vdash (\!|W|\!) : (\!|A|\!)$.

By App, it follows that $(\!|\Gamma|\!) \mid (\!|C|\!) \vdash (\!|V|\!)\,(\!|W|\!) : (\!|B|\!)$ as required.
Case Return
From the assumption that $\{C\}\ \Gamma \vdash \mathsf{return}\, V : A$, we have that $\{C\}\ \Gamma \vdash V : A$. By the inductive hypothesis we have that $(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!)$ and thus by Return we can show that $(\!|\Gamma|\!) \mid (\!|C|\!) \vdash \mathsf{return}\,(\!|V|\!) : (\!|A|\!)$ as required.

Case EffLet
From the assumption that $\{C\}\ \Gamma \vdash \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N : B$, we have that $\{C\}\ \Gamma \vdash M : A$ and $\{C\}\ \Gamma, x : A \vdash N : B$.

By the inductive hypothesis (premise 2), we have that $(\!|\Gamma|\!) \mid (\!|C|\!) \vdash (\!|M|\!) : (\!|A|\!)$ and $(\!|\Gamma|\!), x : (\!|A|\!) \mid (\!|C|\!) \vdash (\!|N|\!) : (\!|B|\!)$. Thus by EffLet it follows that $(\!|\Gamma|\!) \mid (\!|C|\!) \vdash \mathsf{let}\ x \Leftarrow (\!|M|\!)\ \mathsf{in}\ (\!|N|\!) : (\!|B|\!)$.
Case LetPair Similar to EffLet.
Case Case
From the assumption that $\{C\}\ \Gamma \vdash \mathsf{case}\ V\ \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} : B$, we have that $\{C\}\ \Gamma \vdash V : A + A'$, that $\{C\}\ \Gamma, x : A \vdash M : B$, and that $\{C\}\ \Gamma, y : A' \vdash N : B$.

By the inductive hypothesis (premise 1) we have that $(\!| \Gamma |\!) \vdash (\!| V |\!) : (\!| A |\!) + (\!| A' |\!)$, and by premise 2 we have that $(\!| \Gamma |\!), x : (\!| A |\!) \mid (\!| C |\!) \vdash (\!| M |\!) : (\!| B |\!)$ and $(\!| \Gamma |\!), y : (\!| A' |\!) \mid (\!| C |\!) \vdash (\!| M |\!) : (\!| B |\!)$.

By Case, it follows that $(\!| \Gamma |\!) \mid (\!| C |\!) \vdash \mathsf{case}\ (\!| V |\!)\ \{\mathsf{inl}\ x \mapsto (\!| M |\!); \mathsf{inr}\ y \mapsto (\!| N |\!)\} : (\!| B |\!)$.

Case Unroll Immediate.

Case Fork

From the assumption that $\{A\}\ \Gamma \vdash \mathsf{fork}\ M : \mathbf{1}$, we have that $\{A\}\ \Gamma \vdash M : \mathbf{1}$. By the inductive hypothesis, we have that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash (\!| M |\!) : \mathbf{1}$.

We can show that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{spawn}\ (\!| M |\!) : \mathsf{ActorRef}(\mathbf{1})$ and also show that $(\!| \Gamma |\!), x : \mathsf{ActorRef}(\mathbf{1}) \vdash \mathsf{return}\ () : \mathbf{1}$.

Thus by EffLet, it follows that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{let}\ x \Leftarrow \mathsf{spawn}\ (\!| M |\!)\ \mathsf{in}\ \mathsf{return}\ () : \mathbf{1}$ as required.

Case Give

From the assumption that $\{A\}\ \Gamma \vdash \mathsf{give}\ V\ W : \mathbf{1}$, we have that $\{A\}\ \Gamma \vdash V : A$ and $\{A\}\ \Gamma \vdash W : \mathsf{Chan}$. By the inductive hypothesis, we have that $(\!| \Gamma |\!) \vdash (\!| V |\!) : (\!| A |\!)$ and $(\!| \Gamma |\!) \vdash (\!| W |\!) : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)))$. We can show that $(\!| \Gamma |\!) \vdash \mathsf{inl}\ (\!| V |\!) : (\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))$, and thus it follows that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{send}\ \mathsf{inl}\ V\ (\!| W |\!) : \mathbf{1}$ as required.

Case Take

From the assumption that $\{A\}\ \Gamma \vdash \mathsf{take}\ V : A$, we have that $\{A\}\ \Gamma \vdash V : \mathsf{Chan}$.

By the inductive hypothesis (premise 1), we have that $(\!| \Gamma |\!) \vdash (\!| V |\!) : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)))$.

We can show that:

- $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{self} : \mathsf{ActorRef}((\!| A |\!))$
- $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{inr}\ \mathsf{self} : (\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))$
- $(\!| \Gamma |\!), \mathit{selfPid} : \mathsf{ActorRef}((\!| A |\!)) \mid (\!| A |\!) \vdash \mathsf{send}\ \mathsf{inr}\ \mathit{selfPid}\ (\!| V |\!) : \mathbf{1}$
- $(\!| \Gamma |\!), \mathit{selfPid} : \mathsf{ActorRef}((\!| A |\!)), z : \mathbf{1} \mid (\!| A |\!) \vdash \mathsf{receive} : (\!| A |\!)$

Thus by two applications of EffLet (noting that we desugar $M; N$ into $\mathsf{let}\ z \Leftarrow M\ \mathsf{in}\ N$, where z is fresh), we arrive at:

$$(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}\ \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ (\!| V |\!); \mathsf{receive} : (\!| A |\!)$$

as required.

Case NewCh

We have that $\{A\}\ \Gamma; \Delta \vdash \mathsf{newCh} : \mathsf{Chan}$.

Our goal is to show that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash \mathsf{spawn}\ \mathbf{body}\ (\texttt{[ ]}, \texttt{[ ]}) : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)))$. To do so amounts to showing that $(\!| \Gamma |\!) \mid (\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)) \vdash \mathbf{body}\ (\texttt{[ ]}, \texttt{[ ]}) : \mathbf{1}$.

We sketch the proof as follows. Firstly, by the typing of receive, *recvVal* must have type $(\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))$. By inspection of both case branches and LetPair, we have that *state* must have type $\mathsf{List}((\!| A |\!)) \times \mathsf{List}(\mathsf{ActorRef}((\!| A |\!)))$.

We expect **`drain`** to have type

$$\mathsf{List}((\!| A |\!)) \times \mathsf{List}(\mathsf{ActorRef}((\!| A |\!))) \to^{(\!| A |\!)} \mathsf{List}((\!| A |\!)) \times \mathsf{List}(\mathsf{ActorRef}((\!| A |\!)))$$

since we use the returned value as a recursive call to g, which must have the same type of *state*. Inspecting the case split in **`drain`**, we have that the empty list case for values returns the input state, which of course has the same type. The same can be said for the empty list case of the *readers* case split.

In the case where both the values and readers lists are non-empty, we have that v has type $(\!| A |\!)$ and pid has type $\mathsf{ActorRef}((\!| A |\!))$. Additionally, we have that vs has type $\mathsf{List}((\!| A |\!))$ and $pids$ has type $\mathsf{List}(\mathsf{ActorRef}((\!| A |\!)))$. We can show via send that $\mathsf{send}\, v\, pid$ has type $\mathbf{1}$. After desugaring $M; N$ into an EffLet, we can show that a recursive call to f is well-typed. ◀

Theorem 24 *If* $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}$ *with* $\Gamma \asymp \Delta$, *then* $(\!| \Gamma |\!); (\!| \Delta |\!) \vdash (\!| \mathcal{C} |\!)$.

Proof. By induction on the derivation of $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}$.

Case Par

From the assumption that $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}_1 \parallel \mathcal{C}_2$, we have that Δ splits as Δ_1, Δ_2 such that $\{A\}\ \Gamma; \Delta_1 \vdash \mathcal{C}_1$ and $\{A\}\ \Gamma; \Delta_2 \vdash \mathcal{C}_2$.

By the inductive hypothesis, we have that $(\!| \Gamma |\!); (\!| \Delta_1 |\!) \vdash (\!| \mathcal{C}_1 |\!)$ and $(\!| \Gamma |\!); (\!| \Delta_2 |\!) \vdash (\!| \mathcal{C}_2 |\!)$. By the definition of $(\!| - |\!)$ on linear configuration environments, it follows that $(\!| \Delta_1 |\!), (\!| \Delta_2 |\!) = (\!| \Delta |\!)$. Consequently, by Par, we can show that $(\!| \Gamma |\!); (\!| \Delta |\!) \vdash (\!| C |\!)_1 \parallel (\!| C |\!)_2$ as required.

Case Chan

From the assumption that $\{A\}\ \Gamma; \Delta \vdash (\nu a)\mathcal{C}$, we have that $\{A\}\ \Gamma, a : \mathsf{ChanRef}(A); \Delta, a : A \vdash \mathcal{C}$. By the inductive hypothesis, we have that $(\!| \Gamma |\!), a : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))); (\!| \Delta |\!), a : (\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)) \vdash (\!| C |\!)$.

By Pid, it follows that $(\!| \Gamma |\!); (\!| \Delta |\!) \vdash (\nu a)(\!| \mathcal{C} |\!)$ as required.

Case Term

From the assumption that $\{A\}\ \Gamma; \cdot \vdash M$, we have that $\{A\}\ \Gamma \vdash M : \mathbf{1}$.

By Lemma 23, we have that $(\!| \Gamma |\!) \mid (\!| A |\!) \vdash (\!| M |\!) : \mathbf{1}$. By weakening, we can show that $(\!| \Gamma |\!), a : \mathsf{ActorRef}((\!| A |\!)) \mid (\!| A |\!) \vdash (\!| M |\!) : \mathbf{1}$.

It follows that, by Actor, we can construct a configuration of the form $(\!| \Gamma |\!), a : \mathsf{ActorRef}((\!| A |\!)); a : (\!| A |\!) \vdash \langle a, (\!| M |\!), \epsilon \rangle$, and by Pid, we arrive at

$$(\!| \Gamma |\!); \cdot \vdash (\nu a)(\langle a, (\!| M |\!), \epsilon \rangle)$$

as required.

Case Buf

We assume that $\{A\}\ \Gamma; a : A \vdash a(\overrightarrow{V})$, and since $\Gamma \asymp \Delta$, we have that we can write $\Gamma = \Gamma', a : \mathsf{ChanRef}(A)$.

From the assumption that $\{A\}\ \Gamma', a : \mathsf{ChanRef}(A); a : A \vdash a(\overrightarrow{V})$, we have that $\{A\}\ \Gamma', a : \mathsf{ChanRef}(A) \vdash V_i : A$ for each $V_i \in \overrightarrow{V}$.

By repeated application of Lemma 23, we have that $\{A\}\ (\!| \Gamma |\!), a : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))) \vdash (\!| V_i |\!) : (\!| A |\!)$ for each $V_i \in \overrightarrow{V}$, and by ListCons and EmptyList we can construct a list $(\!| V_0 |\!) :: \ldots :: (\!| V_n |\!) :: [\ ]$ with type $\mathsf{List}((\!| A |\!))$.

Relying on our previous analysis of the typing of `body`, we have that $(\!|$`body`$|\!)$ has type:

$$(\mathsf{List}((\!| A |\!)) \times \mathsf{List}(\mathsf{ActorRef}((\!| A |\!)))) \rightarrow^{(\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))} \mathbf{1}$$

Thus it follows that

$$(\!| \Gamma' |\!), a : \mathsf{ActorRef}((\!| A |\!) + \mathsf{ActorRef}((\!| A |\!))) \mid (\!| A |\!) + \mathsf{ActorRef}((\!| A |\!)) \vdash \\ \texttt{body}\ ((\!| \overrightarrow{V} |\!), [\ ]) : \mathbf{1}$$

By ACTOR, we can see that

$$(\!|\Gamma'|\!), a : \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!))); a : (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)) \vdash \langle a, \mathbf{body}\,((\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle$$

as required. ◀

Theorem 27
If $\{A\}\ \Gamma; \Delta \vdash \mathcal{C}_1$, and $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, then there exists some $\mathcal{D}$ such that $(\!|\mathcal{C}_1|\!) \longrightarrow^* \mathcal{D}$ with $\mathcal{D} \equiv (\!|\mathcal{C}_2|\!)$.

Proof.
By induction on the derivation of $\Gamma; \Delta \vdash \mathcal{C}_1$.
Case GIVE

$$\begin{array}{rl}
\text{Assumption} & E[\mathsf{give}\, W\, a] \parallel a(\overrightarrow{V}) \\
\text{Definition of } (\!|-|\!) & (\nu b)(\langle b, (\!|E|\!)[\mathsf{send}\,(\mathsf{inl}\,(\!|W|\!))\, a], \epsilon\rangle) \parallel \langle a, \mathbf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \epsilon\rangle \\
\equiv & (\nu b)(\langle b, (\!|E|\!)[\mathsf{send}\,(\mathsf{inl}\,(\!|W|\!))\, a], \epsilon\rangle \parallel \langle a, \mathbf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \epsilon\rangle) \\
\longrightarrow\ (\text{Send}) & (\nu b)(\langle b, (\!|E|\!)[\mathsf{return}\,()], \epsilon\rangle \parallel \langle a, \mathbf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \mathsf{inl}\,(\!|W|\!)\rangle)
\end{array}$$

Now, let $G[-] = (\nu b)\langle b, (\!|E|\!)[\mathsf{return}\,()], \epsilon\rangle \parallel [-])$.
We now have

$$G[\langle a, \mathbf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), (\mathsf{inl}\,(\!|W|\!))\rangle]$$

which we can expand to

$$\begin{array}{l}
G[\langle a, \quad (\mathsf{rec}\ g(state)\ . \qquad\qquad , \mathsf{inl}\,(\!|W|\!)\rangle] \\
\quad \mathsf{let}\ recvVal \Leftarrow \mathsf{receive}\ \mathsf{in} \\
\quad \mathsf{let}\ (vals, readers) = state\ \mathsf{in} \\
\quad\quad \mathsf{case}\ recvVal\ \{ \\
\quad\quad\quad \mathsf{inl}\ v \mapsto \mathsf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\,(newVals, readers)\ \mathsf{in} \\
\quad\quad\quad\quad \mathbf{body}\,(state') \\
\quad\quad\quad \mathsf{inr}\ pid \mapsto \mathsf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\,(vals, newReaders)\ \mathsf{in} \\
\quad\quad\quad\quad \mathbf{body}\,(state')\})\ ((\!|\overrightarrow{V}|\!), [\,])
\end{array}$$

Applying the arguments to the recursive function f; performing the receive, and the let and case reductions, we have:

$$\begin{array}{l}
G[\langle a, \mathsf{let}\ newVals \Leftarrow (\!|\overrightarrow{V}|\!) +\!\!+ [(\!|W|\!)]\ \mathsf{in} \quad , \epsilon\rangle] \\
\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\,(newVals, [\,])\ \mathsf{in} \\
\quad \mathbf{body}\,(state')
\end{array}$$

Next, we reduce the append operation, and note that since we pass a state without pending readers into **drain**, that the argument is returned unchanged:

$$G[\langle a, \mathsf{let}\ \mathit{state}' \Leftarrow \mathsf{return}\ (\llparenthesis \overrightarrow{V} \rrparenthesis :: \llparenthesis W \rrparenthesis :: \texttt{[ ]})\ \mathsf{in}, \epsilon\rangle]$$
$$\quad \mathbf{body}\ \mathit{state}'$$

Next, we apply the let-reduction and expand the evaluation context:

$$(\nu b)(\langle b, \llparenthesis E \rrparenthesis[\mathsf{return}\ ()], \epsilon\rangle \parallel \langle a, \mathbf{body}\ (\llparenthesis \overrightarrow{V} \rrparenthesis :: \llparenthesis W \rrparenthesis :: \texttt{[ ]}), \epsilon\rangle)$$

which is structurally congruent to

$$(\nu b)(\langle b, \llparenthesis E \rrparenthesis[\mathsf{return}\ ()], \epsilon\rangle) \parallel \langle a, \mathbf{body}\ (\llparenthesis \overrightarrow{V} \rrparenthesis :: \llparenthesis W \rrparenthesis :: \texttt{[ ]}), \epsilon\rangle$$

which is equal to

$$\llparenthesis E[\mathsf{return}\ ()] \parallel a(\llparenthesis \overrightarrow{V} \cdot W \rrparenthesis) \rrparenthesis$$

as required.

Case Take

$$
\begin{array}{rl}
\text{Assumption} & E[\mathsf{take}\ a] \parallel a(W \cdot \overrightarrow{V}) \\
\text{Definition of } \llparenthesis - \rrparenthesis & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\,\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}, \epsilon\rangle) \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), \epsilon\rangle \\
& \qquad\qquad\quad \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ a; \\
& \qquad\qquad\quad \mathsf{receive}] \\
\equiv & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\,\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}, \epsilon\rangle \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), \epsilon\rangle) \\
& \qquad\qquad\quad \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ a; \\
& \qquad\qquad\quad \mathsf{receive}] \\
\longrightarrow (\text{Self}) & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\,\mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{return}\ b\ \mathsf{in}, \epsilon\rangle \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), \epsilon\rangle) \\
& \qquad\qquad\quad \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ a; \\
& \qquad\qquad\quad \mathsf{receive}] \\
\longrightarrow_{\mathsf{M}} (\text{Let}) & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\,\mathsf{send}\ (\mathsf{inr}\ b)\ a;, \epsilon\rangle \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), \epsilon\rangle) \\
& \qquad\qquad\quad \mathsf{receive}] \\
\longrightarrow (\text{Send}) & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\mathsf{return}\ (); \mathsf{receive}], \epsilon\rangle \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), (\mathsf{inr}\ b)\rangle) \\
\longrightarrow_{\mathsf{M}} (\text{Let}) & (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\mathsf{receive}], \epsilon\rangle \parallel \langle b, \mathbf{body}\ (\llparenthesis W \rrparenthesis :: \llparenthesis \overrightarrow{V} \rrparenthesis, \texttt{[ ]}), (\mathsf{inr}\ b)\rangle)
\end{array}
$$

Now, let $G[-] = (\nu b)(\langle b, \llparenthesis E \rrparenthesis[\mathsf{receive}], \epsilon\rangle \parallel [-])$.

Expanding, we begin with:

$$
\begin{array}{l}
G[\langle a, \quad (\mathsf{rec}\ g(state)\ . \qquad , (\mathsf{inr}\ b)\rangle] \\
\quad \mathsf{let}\ recvVal \Leftarrow \mathsf{receive}\ \mathsf{in} \\
\quad \mathsf{let}\ (vals, readers) = state\ \mathsf{in} \\
\quad\quad \mathsf{case}\ recvVal\ \{ \\
\quad\quad\quad \mathsf{inl}\ v \mapsto \mathsf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\,(newVals, readers)\ \mathsf{in} \\
\quad\quad\quad\quad g\,(state') \\
\quad\quad\quad \mathsf{inr}\ pid \mapsto \mathsf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\,(vals, newReaders)\ \mathsf{in} \\
\quad\quad\quad\quad g\,(state')\})\ ((\!| W |\!) :: (\!| \overrightarrow{V} |\!), [\ ])
\end{array}
$$

Reducing the recursive function, receiving from the mailbox, splitting the pair, and then taking the second branch on the case statement, we have:

$$
\begin{array}{l}
G[\langle a, \quad \mathsf{let}\ newReaders \Leftarrow [\ ] +\!\!+ [b]\ \mathsf{in} \qquad , \epsilon\rangle] \\
\quad \mathsf{let}\ state' \Leftarrow \mathbf{drain}\ (vals, newReaders)\ \mathsf{in} \\
\quad \mathbf{body}\ state'
\end{array}
$$

Reducing the list append operation, expanding **drain**, and re-expanding G, we have:

$$
\begin{array}{l}
(\nu b)(\langle b, E[\mathsf{receive}], \epsilon\rangle \parallel \langle a, \mathsf{let}\ state' \Leftarrow (\mathsf{rec}\ f(x). \qquad , \epsilon\rangle \\
\quad \mathsf{let}\ (vals, readers) \ = x\ \mathsf{in} \\
\quad \mathsf{case}\ vals\ \{ \\
\quad\quad [\ ] \mapsto \mathsf{return}\,(vals, readers) \\
\quad\quad v :: vs \mapsto \\
\quad\quad\quad \mathsf{case}\ readers\ \{ \\
\quad\quad\quad\quad [\ ] \mapsto \mathsf{return}\,(vals, readers) \\
\quad\quad\quad\quad pid :: pids \mapsto \mathsf{send}\ v\ pid; \\
\quad\quad\quad\quad\quad f\ (vs, pids)\}\})\ ((\!| W |\!) :: (\!| \overrightarrow{V} |\!), [b])\ \mathsf{in} \\
\quad \mathbf{body}\ state'
\end{array}
$$

Next, we reduce the recursive function and the case statements:

$$
\begin{array}{l}
(\nu b)(\langle b, E[\mathsf{receive}], \epsilon\rangle \parallel \langle a, \mathsf{let}\ state' \Leftarrow \mathsf{send}\ (\!| W |\!)\ b; \qquad , \epsilon\rangle \\
\quad\quad \mathbf{drain}\,((\!| \overrightarrow{V} |\!), [\ ])) \\
\quad \mathbf{body}\ state'
\end{array}
$$

We next perform the send operation, and note that the recursive call to **drain** will return the argument unchanged, since $((\!| \overrightarrow{V} |\!), [\ ])$ has no pending requests.

Thus we have:

$$(\nu b)(\langle b, E[\mathsf{receive}], (\!| W |\!)\rangle \parallel \langle a, \mathbf{body}\ (((\!| \overrightarrow{V} |\!), [\ ])), \epsilon\rangle$$

Finally, we perform the receive and apply a structural congruence to arrive at

$$(\nu b)(\langle b, E[(\!| W |\!)], \epsilon\rangle) \parallel \langle a, \mathbf{body}\ (((\!| \overrightarrow{V} |\!), [\ ])), \epsilon\rangle$$

which is equal to

$$(\!| E[\mathsf{return}\, W] \parallel a(\overrightarrow{V}) |\!)$$

as required.

Case NEWCH

$$\begin{array}{rl}
\text{Assumption} & E[\mathsf{newCh}] \\
\text{Definition of } (\!| - |\!) & (\nu a)(\langle a, (\!| E |\!)[\mathsf{spawn}\ (\mathtt{body}\ ([\ ], [\ ]))], \epsilon\rangle) \\
\longrightarrow & (\nu a)(\nu b)(\langle a, (\!| E |\!)[\mathsf{return}\, b], \epsilon\rangle \parallel \langle b, \mathtt{body}\ ([\ ], [\ ]), \epsilon\rangle) \\
\equiv & (\nu b)(\nu a)(\langle a, (\!| E |\!)[\mathsf{return}\, b], \epsilon\rangle) \parallel \langle b, \mathtt{body}\ ([\ ], [\ ]), \epsilon\rangle) \\
= & (\!| (\nu b)(E[\mathsf{return}\, b] \parallel b(\epsilon)) |\!)
\end{array}$$

as required.

Case **Fork**

$$\begin{array}{rl}
\text{Assumption} & E[\mathsf{fork}\, M] \\
\text{Definition of } (\!| - |\!) & (\nu a)(\langle a, (\!| E |\!)[\mathsf{let}\ x \Leftarrow \mathsf{spawn}\ (\!| M |\!)\ \mathsf{in}\ \mathsf{return}\ ()], \epsilon\rangle) \\
\longrightarrow & (\nu a)(\nu b)(\langle a, (\!| E |\!)[\mathsf{let}\ x \Leftarrow \mathsf{return}\, b\ \mathsf{in}\ \mathsf{return}\ ()], \epsilon\rangle \parallel \langle a, (\!| M |\!), \epsilon\rangle) \\
\longrightarrow_{\mathsf{M}} & (\nu a)(\nu b)(\langle a, (\!| E |\!)[\mathsf{return}\ ()], \epsilon\rangle \parallel \langle b, (\!| M |\!), \epsilon\rangle) \\
\equiv & (\nu a)(\langle a, (\!| E |\!)[\mathsf{return}\ ()], \epsilon\rangle) \parallel (\nu b)(\langle b, (\!| M |\!), \epsilon\rangle) \\
= & (\!| E[\mathsf{return}\ ()] \parallel M |\!)
\end{array}$$

as required.

Case **Lift** Immediate by the inductive hypothesis.

Case **LiftV** Immediate by Lemma 25.