Mixing Metaphors

Actors as Channels and Channels as Actors

Simon Fowler, Sam Lindley, and Philip Wadler

The University of Edinburgh
simon.fowler@ed.ac.uk, sam.lindley@ed.ac.uk, wadler@inf.ed.ac.uk

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 show translations from λ_{act} into λ_{ch} and λ_{ch} into λ_{act} and prove that both translations are type- and semantics-preserving.

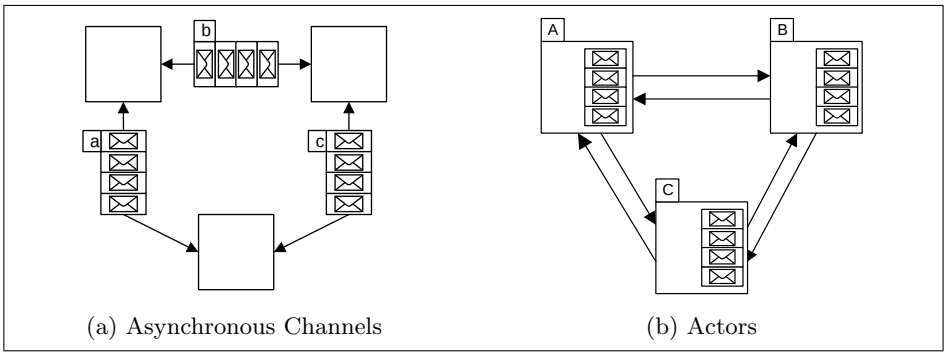
1 Introduction

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.

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 [32], 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.

Channel-based languages such as Go enjoy a firm basis in process calculi such as the π-calculus [31]. It is easy to type channels, either with simple types [30] or more complex systems such as session types [13, 20, 21]. 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 [4, 6].



(a) Asynchronous Channels (b) Actors

Fig. 1: Channels and Actors

Both models are popular with developers, with channel-based languages and frameworks such as Go, Concurrent ML [37], and Hopac [22]; 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/akka framework is similar in nature to coroutines?" [24], 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?" [23]

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 the Go Actor Model framework (GAM) [15] emulate actor-style programming in a channel-based language in an attempt to gain some of the benefits. Hopac [22] is a channel-based library for F#, based on Concurrent ML [37]. 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 GAM, 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 GAM.

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 [37]. By working in the asynchronous setting, we can concentrate on the more fundamental differences between the two models.

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 consider an extension of λ_{act} with the ability to make synchronous calls; an extension of λ_{ch} with input-guarded nondeterministic choice; and discuss how λ_{act} could be extended with behavioural types (§7).

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 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 $\mathsf{Operation}(\mathsf{A}) = \mathsf{Push}(A) \mid \mathsf{Pop}(B)$ (where $B = \mathsf{ChanRef}(A)$ for channels, and $\mathsf{ActorRef}(A)$ for actors) to describe operations on the stack, and $\mathsf{Option}(\mathsf{A}) = \mathsf{Some}(A) \mid \mathsf{None}$ to handle the possibility of popping from an empty stack.

Figure 2 shows the stack implemented using channels (Figure 2a) and using actors (Figure 2b). 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:

```
chanStack(ch) ≜
  rec loop(st).
    let cmd ⇐ take ch in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resCh) ↦
        case st {
          [ ] ↦ give (None) resCh;
                 loop [ ]
          x :: xs ↦ give (Some(x)) resCh;
                     loop xs    }
    }

chanStackClient(stackCh) ≜
  give (Push(5)) stackCh;
  let resCh ⇐ newCh in
  give (Pop(resCh)) stackCh;
  take resCh

chainMain ≜
  let stackCh ⇐ newCh in
  fork (chanStack(stackCh) [ ]);
  chanStackClient(stackCh)
```

(a) Channel-Based Stack

```
actorStack ≜
  rec loop(st).
    let cmd ⇐ receive in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resPid) ↦
        case st {
          [ ] ↦ send (None) resPid;
                 loop [ ]
          x :: xs ↦ send (Some(x)) resPid;
                     loop xs    }
    }

actorStackClient(stackPid) ≜
  send (Push(5)) stackPid;
  let selfPid ⇐ self in
  send (Pop(selfPid)) stackPid;
  receive

actorMain ≜
  let stackPid ⇐ spawn (actorStack [ ]) in
  actorStackClient(stackPid)
```

(b) Actor-Based Stack

Fig. 2: Concurrent Stacks using Channels and Actors

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 `chanStack-Client` 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 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

```
chanClient2(intStackCh,
            stringStackCh) ≜
  let intResCh ⇐ newCh in
  let strResCh ⇐ newCh in
  give (Pop(intResCh)) intStackCh;
  let res1 ⇐ take intResCh in
  give (Pop(strResCh)) stringStackCh;
  let res2 ⇐ take strResCh in
  (res1, res2)
```

```
actorClient2(intStackPid,
             stringStackPid) ≜
  let selfPid ⇐ self in
  send (Pop(selfPid)) intStackPid;
  let res1 ⇐ receive in
  send (Pop(selfPid)) stringStackPid;
  let res2 ⇐ receive in
  (res1, res2)
```

Fig. 3: Clients Interacting with Multiple Stacks

shown in Figure 3. Here, `chanStackClient2` pushes 5 onto the integer stack as before. Next, the client 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; some implementations sidestep this through the use of subtyping [17], and support synchronisation through either selectively receiving from a mailbox [5], or through synchronisation abstractions such as futures [9].

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

In this section we introduce λ_{ch}, a concurrent λ-calculus extended with asynchronous channels.

3.1 Syntax and Typing of Terms

Figure 4 gives the syntax and typing rules of λ_{ch}, a lambda calculus based on fine-grain call-by-value [27]: terms are partitioned into values and computations. 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.

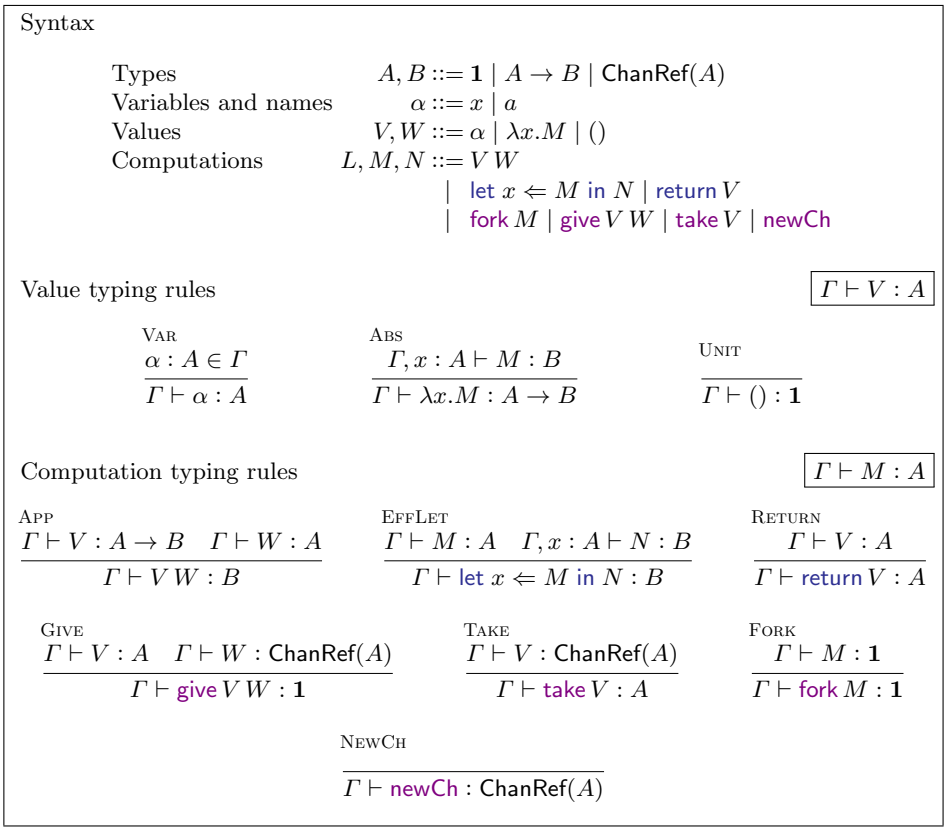


Fig. 4: Syntax and Typing Rules for λ_{ch} Terms and Values

Assuming an extension of the language with integers and arithmetic operators, we can defined a function $\mathtt{neg}(c)$ which receives a number n along channel c and replies with the negation of n as follows:

$$\begin{aligned}\mathtt{neg}(c) \triangleq\ & \mathsf{let}\ n \Leftarrow \mathsf{take}\ c\ \mathsf{in} \\ & \mathsf{let}\ \mathit{negN} \Leftarrow (-n)\ \mathsf{in}\ \mathsf{give}\ \mathit{negN}\ c\end{aligned}$$

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 5). Configu-

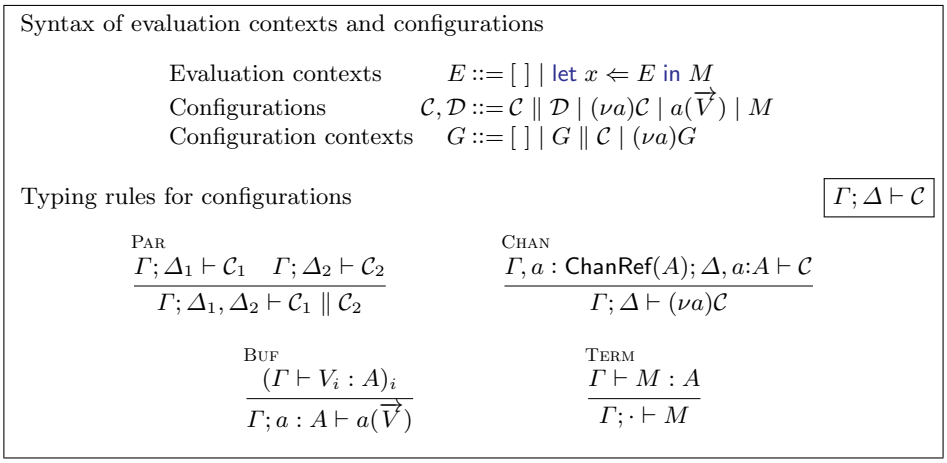


Fig. 5: λ_{ch} Configurations and Evaluation Contexts

rations consist of parallel composition ($\mathcal{C} \parallel \mathcal{D}$), restrictions ($(\nu a)\mathcal{C}$), computations ($M$), and buffers ($a(\overrightarrow{V})$).

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 6 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. Figure 5 includes typing rules on configurations, which ensure that buffers are well-scoped and contain values of the correct type. 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. 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; a term is typeable as a process if it has type $\mathbf{1}$ under an empty linear environment and a buffer $a(\overrightarrow{V})$ is typeable under a singleton linear environment $a : A$ if all $V_i \in \overrightarrow{V}$ have type A. Linearity ensures that under a name restriction $(\nu a)\mathcal{C}$ a configuration $\mathcal{C}$ will contain exactly one buffer with name a.

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

Reduction on terms

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

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

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

Fig. 6: Reduction on λ_{ch} Terms and Configurations

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 the online appendix.

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 entirely 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.

We now state some properties of λ_{ch}. Firstly, typeability of configurations is preserved by structural congruence.

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

Next, we can show that reduction preserves the typeability of configurations.

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

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 [39], and Lindley and Morris [28]) or those using channel priorities (for example, the calculus of Padovani and Novara [35]), 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 ($\lambda_{\mathbf{ch}}$)).
A configuration $\mathcal{C}$ is in canonical form *if $\mathcal{C}$ 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.

Theorem 8 (Weak progress ($\lambda_{\mathbf{ch}}$ 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 fully-reduced term of the form* $\mathsf{return}\, V$;
2. *A buffer $a_i(\overrightarrow{V_i})$, or;*
3. *A term of the form $E[\mathsf{take}\, a_i]$, where $\overrightarrow{V_i} = \epsilon$.*

4 $\lambda_{\mathbf{act}}$: 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. [18] 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. [10] provides a detailed taxonomy), 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 challenging, in particular due to type pollution. Type-parameterised actors were introduced by He [16] and He et al. [17], and more recently implemented in libraries such as Akka Typed [3] and Typed Actors [38].

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 [14]) to type terms with respect to the mailbox type of the enclosing actor.

4.1 Syntax and Typing of Terms

Figure 7 shows the syntax and typing rules for λ_{act}. As with λ_{ch}, α ranges over variables and names. $\mathsf{ActorRef}(A)$ 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 $\mathit{Int} \to^{\mathit{Int}} \mathit{Int}$, 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

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

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

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

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

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

Fig. 7: λ_{act} Typing Rules

4.2 Operational Semantics

Figure 8 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 9 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 $\Gamma \vdash M' : A$.

Syntax of evaluation contexts and configurations

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

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

$$\textsc{Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C}_1 \quad \Gamma; \Delta_2 \vdash \mathcal{C}_2}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C}_1 \parallel \mathcal{C}_2} \qquad \textsc{Pid}\ \frac{\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

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

Fig. 8: λ_{act} Evaluation Contexts and Configurations

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 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

Reduction on terms

$$(\lambda x.M)\, V \longrightarrow_{\mathsf{M}} M\{V/x\}$$
$$\mathsf{let}\ x \Leftarrow \mathsf{return}\ V\ \mathsf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\}$$
$$E[M] \longrightarrow_{\mathsf{M}} E[M'] \quad \text{if } M \longrightarrow_{\mathsf{M}} M'$$

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

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

Fig. 9: Reduction Rules and Equivalences for Terms and Configurations

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 ($\lambda_{\mathbf{act}}$)). *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 ($\lambda_{\mathbf{act}}$ 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. *$\langle a, E[\mathsf{receive}], \epsilon\rangle$.*

5 From $\lambda_{\mathbf{act}}$ to $\lambda_{\mathbf{ch}}$

With both calculi in place, we are now ready to look at the translation from λ_{act} into λ_{ch}. The translation strategy is straightforward, but not previously formalised. 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 ($\lambda_{\mathbf{act}}$ to $\lambda_{\mathbf{ch}}$)

Figure 10 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 to emulate a mailbox.

An actor reference is simply 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 are translated to themselves, with the exception of λ abstractions, which are translated to take an additional parameter denoting the channel used to emulate operations on a mailbox. Given a parameter ch, the translation function for terms emulates receive by taking a value from ch, and emulates self by returning ch.

Although the translation is straightforwardly defined, it is a *global* translation [11], since all functions must be modified in order to take the channel emulating the mailbox as an additional parameter.

Translation on types

$$\begin{aligned} [\![\,\mathsf{ActorRef}(A)\,]\!] &= \mathsf{ChanRef}([\![\,A\,]\!]) \\ [\![\,\mathbf{1}\,]\!] &= \mathbf{1} \\ [\![\,A \to^{C} B\,]\!] &= [\![\,A\,]\!] \to \mathsf{ChanRef}([\![\,C\,]\!]) \to [\![\,B\,]\!] \end{aligned}$$

Translation on values

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

Translation on computation terms

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

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

Translation on configurations

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

Fig. 10: Translation from λ_{act} into λ_{ch}

5.2 Properties of the Translation

The translation on terms and values preserves typing. We extend the translation function pointwise to typing environments:

$$[\![\,\alpha_1 : A_1, \ldots, \alpha_n : A_n\,]\!] = \alpha_1 : [\![\,A_1\,]\!], \ldots, \alpha_n : [\![\,A_n\,]\!]$$

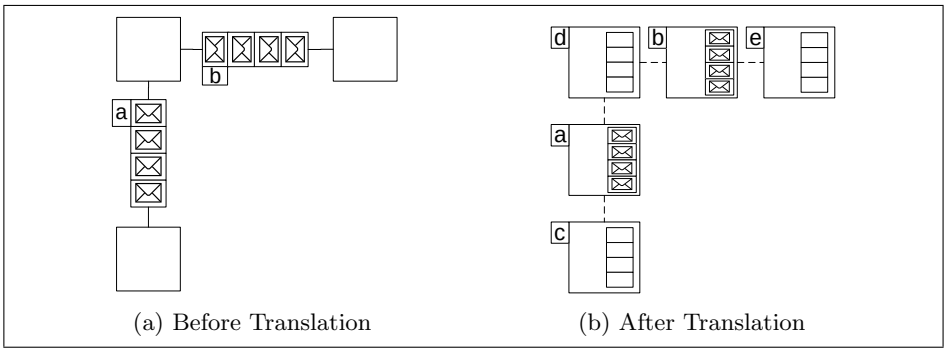
Lemma 17 ($[\![-]\!]$ preserves typing (terms and values)).

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}.



(a) Before Translation (b) After Translation

Fig. 11: Translation Strategy: λ_{ch} into λ_{act}

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 $M_1 \longrightarrow_M M_2$ in λ_{act}, then given some α, $[\![\, M_1 \,]\!]\ \alpha \longrightarrow_M^* [\![\, M_2 \,]\!]\ \alpha$ 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 a actor mailbox using a channel, using it to implement operations which normally rely on the context of the actor. Although a global translation, 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 Translation Strategy (λ_{ch} into λ_{act})

To translate typed actors into typed channels (shown in Figure 11), 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 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 actors. Consequently, we require that before translating into actors, that *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 Bool. It is possible to construct a variant type $\langle \ell_1 : \mathsf{Int}, \ell_2 : \mathsf{String}, \ell_3 : \mathsf{Bool} \rangle$ 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 $\mathsf{ChanRef}(Int)$, we would instead send a value $\langle \ell_1 = 5 \rangle$.

The online appendix provides more details, and proves that the transformation is safe.

6.2 Extensions to Core Language

To emulate channels using actors, we require several more term-level language constructs: sums, products, recursive functions, and lists. Recursive functions are used to implement an event loop, and lists 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 used to encode monomorphic variant types.

Figure 12 shows the extensions required to the core term language and reduction rules; we omit the reduction rules for case analysis on the right injection of a sum and the empty list. 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.

6.3 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 $\mathsf{ChanRef}(B)$.

Metalevel Definitions. The majority of the translation lies within the translation of newCh, which makes use of the meta-level definitions shown in Figure 13. The `body` function emulates a channel. Firstly, the actor receives a message *recvVal* from its mailbox, which is 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

Syntax

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

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

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

$$\textsc{Inr}\ \frac{\Gamma \vdash V : B}{\Gamma \vdash \mathsf{inr}\, V : A + B} \qquad \textsc{EmptyList}\ \frac{}{\Gamma \vdash [\,] : \mathsf{List}(A)} \qquad \textsc{Cons}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{List}(A)}{\Gamma \vdash V :: W : \mathsf{List}(A)}$$

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

$$\textsc{Let}\ \frac{\Gamma \vdash V : A \times A' \quad \Gamma, x : A, y : A' \vdash M : B}{\Gamma \vdash \mathsf{let}\, (x, y) = V\, \mathsf{in}\, M : B} \qquad \textsc{ListCase}\ \frac{\Gamma \vdash V : \mathsf{List}(A) \quad \Gamma \vdash M : B \quad \Gamma, x : A, y : \mathsf{List}(A) \vdash N : B}{\Gamma \vdash \mathsf{case}\, V\, \{[\,] \mapsto M; x :: y \mapsto N\} : B}$$

$$\textsc{Case}\ \frac{\Gamma \vdash V : A + A' \quad \Gamma, x : A \vdash M : B \quad \Gamma, y : A' \vdash N : B}{\Gamma \vdash \mathsf{case}\, V\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} : B}$$

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

$$\begin{array}{rl} (\mathsf{rec}\, f(x)\,.\, M\, V) & \longrightarrow_{\mathsf{M}} M\{(\mathsf{rec}\, f(x)\,.\, M)/f, V/x\} \\ \mathsf{let}\, (x, y) = (V, W)\, \mathsf{in}\, M & \longrightarrow_{\mathsf{M}} M\{V/x, W/y\} \\ \mathsf{case}\, (\mathsf{inl}\, V)\, \{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} & \longrightarrow_{\mathsf{M}} M\{V/x\} \\ \mathsf{case}\, V :: W\, \{[\,] \mapsto M; x :: y \mapsto N\} & \longrightarrow_{\mathsf{M}} N\{V/x, W/y\} \end{array}$$

Fig. 12: Extensions to Core Languages to Allow Translation from λ_{ch} into λ_{act}

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 $(\!|-|\!)\, C$ takes the type of all channels C as its argument, and is used to annotate all 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 $\mathsf{ActorRef}((\!|\, C\,|\!)\, C + \mathsf{ActorRef}((\!|\, C\,|\!)\, C))$, meaning an actor which can receive a request to either store a value of type $(\!|\, C\,|\!)\, C$, or to dequeue a value and send it to a process ID of type $\mathsf{ActorRef}((\!|\, C\,|\!)\, C)$.

Translation on Communication and Concurrency Primitives. We omit the translation on values and functional terms, which are homomorphisms. Pro-

```
drain ≜ rec f(x).                          body ≜ rec g(state) .
  let (vals, pids) = x in                    let recvVal ⇐ receive in
  case vals {                                let (vals, pids) = state in
    [ ] ↦ return (vals, pids)                case recvVal {
    v :: vs ↦                                    inl v ↦ let vals′ ⇐ vals ++ [v] in
      case pids {                                          let state′ ⇐ drain (vals′, pids) in
          [ ] ↦ return (vals, pids)                        g (state′)
          pid :: pids ↦ send v pid;                inr pid ↦ let pids′ ⇐ pids ++ [pid] in
                        f (vs, pids)}                        let state′ ⇐ drain (vals, pids′) in
  }                                                          g (state′)   }
```

Fig. 13: Metalevel Definitions Required for the Translation from λ_{ch} into λ_{act}

cesses 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}. Consequently, the translation of a process evaluating term M is an actor with some fresh name a and an empty mailbox evaluating $(\!|M|\!)$, 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* [11].

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 A. 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).

1. *If* $\Gamma = \alpha_1{:}A_1, \ldots, \alpha_n : A_n$, *define* $(\!|\Gamma|\!)\, B) = \alpha_1 : (\!|A_1|\!)\, B, \ldots, \alpha_n : (\!|A_n|\!)\, B$.

Translation on types

$$\begin{aligned}(\!|\,\mathsf{Chan}\,|\!)\,C &= \mathsf{ActorRef}((\!|\,C\,|\!)\,C + \mathsf{ActorRef}((\!|\,C\,|\!)\,C)) \\ (\!|\,A \to B\,|\!)\,C &= ((\!|\,A\,|\!)\,C) \to^{(\!|\,C\,|\!)\,C} ((\!|\,B\,|\!)\,C)\end{aligned}$$

Translation on communication and concurrency primitives

$$\begin{aligned}(\!|\,\mathsf{fork}\,M\,|\!) &= \mathsf{let}\ x \Leftarrow \mathsf{spawn}\,(\!|\,M\,|\!)\ \mathsf{in}\ \mathsf{return}\,() \\ (\!|\,\mathsf{give}\,V\,W\,|\!) &= \mathsf{send}\,(\mathsf{inl}\,(\!|\,V\,|\!))\,(\!|\,W\,|\!) \\ (\!|\,\mathsf{take}\,V\,|\!) &= \mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in} \qquad \mathit{selfPid}\ \text{is a fresh variable} \\ &\qquad\quad \mathsf{send}\,(\mathsf{inr}\ \mathit{selfPid})\,(\!|\,V\,|\!); \\ &\qquad\quad \mathsf{receive} \\ (\!|\,\mathsf{newCh}\,|\!) &= \mathsf{spawn}\,(\mathsf{body}\,([\,],[\,]))\end{aligned}$$

Translation on configurations

$$\begin{aligned}(\!|\,\mathcal{C}_1 \parallel \mathcal{C}_2\,|\!) &= (\!|\,\mathcal{C}_1\,|\!) \parallel (\!|\,\mathcal{C}_2\,|\!) \\ (\!|\,(\nu a)\mathcal{C}\,|\!) &= (\nu a)(\!|\,\mathcal{C}\,|\!) \\ (\!|\,M\,|\!) &= (\nu a)(\langle a, (\!|\,M\,|\!), \epsilon\rangle) \qquad a\ \text{is a fresh name} \\ (\!|\,a(\overrightarrow{V})\,|\!) &= \langle a, \mathsf{body}\,(\!|\,\overrightarrow{V}\,|\!), [\,]), \epsilon\rangle \\ &\qquad \text{where } (\!|\,\overrightarrow{V}\,|\!) = (\!|\,V_0\,|\!) :: \ldots :: (\!|\,V_n\,|\!) :: [\,]\end{aligned}$$

Fig. 14: Translation from λ_{ch} into λ_{act}

2. *Given* $a\ \Delta = a_1 : A, \ldots, a_n : A$, *define* $(\!|\,\Delta\,|\!)\,A =$
$a_1 : ((\!|\,A\,|\!)\,A + \mathsf{ActorRef}((\!|\,A\,|\!)\,A)), \ldots, a_n : ((\!|\,A\,|\!)\,A + \mathsf{ActorRef}((\!|\,A\,|\!)\,A))$.

We can now begin to state our final set of results. The translation on terms preserves typing.

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

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

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 and Future Work

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 discuss how to nondeterministically choose a message from a collection of possible sources. Thirdly, 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 [9, 25, 26] 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 $\mathsf{ActorRef}(A, B)$, 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 [8] 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. Figure 16 shows the modified translation from λ_{ch} into λ_{act} with wait. Channel references are 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

Additional types, terms, and configuration reduction rule

$$\text{Types} ::= \mathsf{ActorRef}(A, B) \mid \ldots \qquad \text{Terms} ::= \mathsf{wait}\, V \mid \ldots$$

$$\langle a, E[\mathsf{wait}\, b], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathsf{return}\, V'], \overrightarrow{V}\rangle \parallel \langle b, \mathsf{return}\, V', \overrightarrow{W}\rangle$$

$$(\nu a)(\langle a, \mathsf{return}\, V, \overrightarrow{V}\rangle) \parallel \mathcal{C} \equiv \mathcal{C}$$

Modified typing rules for terms $\boxed{\Gamma \mid A, B \vdash M : A}$

$$\textsc{Sync-Spawn}\ \frac{\Gamma \mid A, B \vdash M : B}{\Gamma \mid C, C' \vdash \mathsf{spawn}\, M : \mathsf{ActorRef}(A, B)} \qquad \textsc{Sync-Wait}\ \frac{\Gamma \vdash V : \mathsf{ActorRef}(A, B)}{\Gamma \mid C, C' \vdash \mathsf{wait}\, V : B}$$

$$\textsc{Sync-Self}\ \frac{}{\Gamma \mid A, B \vdash \mathsf{self} : \mathsf{ActorRef}(A, B)}$$

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

$$\textsc{Sync-Actor}\ \frac{\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash M{:}B \quad (\Gamma, a{:}\mathsf{ActorRef}(A, B) \vdash V_i{:}A)_i}{\Gamma, a : \mathsf{ActorRef}(A, B); a{:}(A, B) \vdash \langle a, M, \overrightarrow{V}\rangle} \qquad \textsc{Sync-Nu}\ \frac{\Gamma, a : \mathsf{ActorRef}(A, B); \Delta, a : (A, B) \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

Fig. 15: Extensions to Add Synchronisation to λ_{act}

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 actor spawned to make the request 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* $\mathcal{D} \equiv (\!|\, \mathcal{C}_2 \,|\!)$*.*

Translation in the other direction requires named threads and a join construct in λ_{ch}, but is otherwise unsurprising.

7.2 Choice

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

Modified translation on types

$$(\!|\,\mathsf{ChanRef}(A)\,|\!) = \mathsf{ActorRef}((\!|\,A\,|\!) + \mathsf{ActorRef}((\!|\,A\,|\!), (\!|\,A\,|\!)), \mathbf{1})$$
$$(\!|\,A \to B\,|\!) = (\!|\,A\,|\!) \to^{C,\mathbf{1}} (\!|\,B\,|\!)$$

Modified translation of take

$$\begin{aligned}(\!|\,\mathsf{take}\,V\,|\!) = {} & \mathsf{let}\ \mathit{requestorPid} \Leftarrow \mathsf{spawn}\ (\\ & \quad \mathsf{let}\ \mathit{selfPid} \Leftarrow \mathsf{self}\ \mathsf{in}\\ & \quad \mathsf{send}\ (\mathsf{inr}\ \mathit{selfPid})\ (\!|\,V\,|\!);\\ & \quad \mathsf{receive})\ \mathsf{in}\\ & \mathsf{wait}\ \mathit{requestorPid}\end{aligned}$$

Fig. 16: Translation from λ_{ch} into λ_{act} Using Synchrony

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

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

Fig. 17: Additional Typing and Evaluation Rules for λ_{ch} with Choice

Here we have only considered the most basic case of selective choice over two channels. More generally, it can be extended to arbitrary regular data types [36]. Since Concurrent ML [37] 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 patters of join calculus [12] and the selective receive operation on mailboxes of Erlang also provide general abstractions for selective communication.

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 17. Emulating such a construct satisfactorily in λ_{act} is nontrivial, because data must be received from a *local* message queue as opposed to a separate entity. One approach could be to use the work of Chaudhuri [7] 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.

7.3 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* [20, 21] 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 follows:

$$!\mathsf{Int}.!\mathsf{Int}.?\mathsf{Int}.\mathsf{end}$$

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 [33].

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 $\lambda(\mathsf{fut})$ [34], 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 [13, 28].

Channel-based programming languages are inspired by CSP [19] and the π-calculus [31]; 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 [37] 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. [18] 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. In the object-oriented setting, the actor model inspires *active objects* [26]: objects supporting asynchronous method calls which return responses using futures. De Boer et al. [9] describe a language and proof system for active objects with cooperatively scheduled threads within each object. Core ABS [25] 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 have smaller calculi.

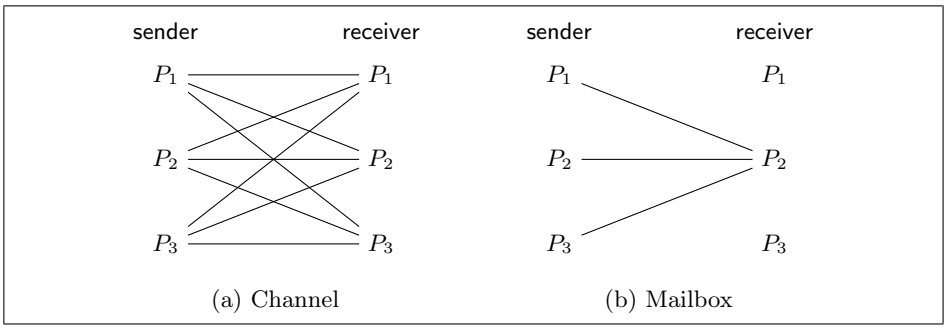
Links [8] is a programming language designed for developing web applications which includes an implementation of typed message-passing concurrency built on an effect type system. The design of λ_{act} was inspired by Links.

Hopac [22] 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. The discrepancy is illustrated in Figure 18. Any process can send and receive messages along a channel (Figure 18a), whereas, although any process can send a message to a mailbox, only one process can receive from that mailbox (Figure 18b). 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 [29].



(a) Channel (b) Mailbox

Fig. 18: Mailboxes as Pinned Channels

Bibliography

[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] Akka Typed. http://doc.akka.io/docs/akka/current/scala/typed.html, 2016.

[4] Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[5] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[6] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP*. " O'Reilly Media, Inc.", 2016.

[7] Avik Chaudhuri. A Concurrent ML Library in Concurrent Haskell. In *ICFP*, pages 269–280, New York, NY, USA, 2009. ACM.

[8] 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.

[9] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330. Springer, 2007.

[10] 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.

[11] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

[12] 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.

[13] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20:19–50, January 2010.

[14] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38. ACM, 1986.

[15] Go Actor Model. https://github.com/AsynkronIT/gam, 2016.

[16] Jiansen He. Type-parameterized actors and their supervision. MPhil thesis, The University of Edinburgh, 2014.

[17] Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: From Akka to TAkka. In *SCALA*, pages 23–33. ACM, 2014.

[18] 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.

[19] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978. ISSN 0001-0782.

[20] 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.

[21] 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.

[22] Hopac. http://www.github.com/Hopac/hopac, 2016.

[23] How are Akka actors different from Go channels? https://www.quora.com/How-are-Akka-actors-different-from-Go-channels, 2013.

[24] Is Scala's actors similar to Go's coroutines? http://stackoverflow.com/questions/22621514/is-scalas-actors-similar-to-gos-coroutines, 2014.

[25] 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.

[26] 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.

[27] Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

[28] Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In *ESOP*, pages 560–584. Springer, 2015.

[29] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[30] Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and algebra of specification*, pages 203–246. Springer, 1993.

[31] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999.

[32] 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.

[33] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *COORDINATION*, pages 131–146. Springer, 2014.

[34] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

[35] 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.

[36] Jennifer Paykin, Antal Spector-Zabusky, and Kenneth Foner. `choose` your own derivative. In *TyDe*, pages 58–59. ACM, 2016.

[37] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[38] Typed Actors. https://github.com/knutwalker/typed-actors, 2016.

[39] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24 (2-3):384–418, 2014.