# Kalas: A Verified, End-To-End Compiler for a Choreographic Language


**Johannes Åman Pohjola**
University of New South Wales, Sydney, Australia

**Alejandro Gómez-Londoño**
Chalmers University of Technology, Gothenburg, Sweden

**James Shaker**
Australian National University, Canberra, Australia

**Michael Norrish**
Australian National University, Canberra, Australia



## Abstract

Choreographies are an abstraction for globally describing deadlock-free communicating systems. A choreography can be compiled into multiple endpoints preserving the global behavior, providing a path for concrete system implementations. Of course, the soundness of this approach hinges on the correctness of the compilation function. In this paper, we present a verified compiler for Kalas, a choreographic language. Its machine-checked end-to-end proof of correctness ensures all generated endpoints adhere to the system description, preserving the top-level communication guarantees. This work uses the verified CakeML compiler and HOL4 proof assistant, allowing for concrete executable implementations and statements of correctness at the machine code level for multiple architectures.



**2012 ACM Subject Classification** Theory of computation → Concurrency; Software and its engineering → Software verification; Software and its engineering → Compilers

**Keywords and phrases** Choreographies, Interactive Theorem Proving, Compiler Verification

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2022.27

**Supplementary Material** *Software (Source Code)*: `https://github.com/CakeML/choreo/`

**Acknowledgements** We are grateful to Marco Carbone, Rob van Glabbeek and Magnus Myreen for insightful discussion on this work.


## 1 Introduction

In recent years, advances in the fields of concurrency theory and systems verification have taken us closer to the idea of a truly correct communicating system. The former abounds with beautiful high-level specification formalisms and reasoning techniques for communicating systems. At the same time, the latter provides detailed correctness proofs of the low-level computing infrastructure (e.g., compilers, language runtimes, and operating systems) needed to implement them. There is then much to be gained by joining these worlds. In particular, high-level descriptions of communicating systems – along with their guarantees – could be propagated down to low-level implementations to create an end-to-end result. One promising approach is choreographic programming, which at a high level describes communicating systems while providing by-construction guarantees.

A choreography is a global description of a communicating system, written in a style reminiscent of the `Alice` → `Bob` notation for protocol descriptions. Compared to the traditional approach of writing separate programs for every `Alice` and `Bob`, the choreographic approach has the advantage that it is impossible to write a program with a communication mismatch. In particular, deadlock freedom holds by construction. Furthermore, through a procedure called *endpoint projection* choreographies can be compiled into separate programs for each endpoint, such that their parallel composition implements the global behaviour.

© Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish;
licensed under Creative Commons License CC-BY 4.0

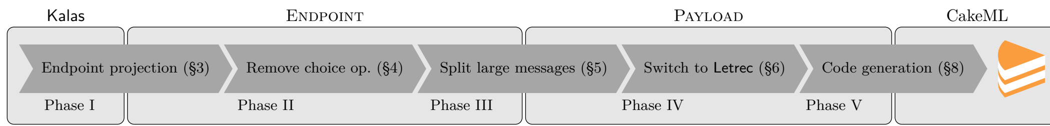



**Figure 1** Compilation Steps and Intermediate Languages.

In this paper, we present a compiler for our choreographic language, Kalas, with a machine-checked, end-to-end proof of correctness. That is, we create an environment based on the HOL4 interactive theorem prover [24] where programmers can write choreographies, and then have the system automatically generate executable code for each endpoint, along with a proof of its correct compilation into machine-code.

Our compiler is structured into five phases, illustrated in Figure 1, with associated correctness result for each. The first step is endpoint projection, where the global choreography is projected into a parallel composition of sequential programs implementing each endpoint, expressed in a process algebra we call ENDPOINT. Second, the ENDPOINT operators for branch selection are encoded with more primitive operators. Third, ENDPOINT is compiled to a second process algebra, PAYLOAD. While messages in ENDPOINT can be arbitrarily large, messages in PAYLOAD have a fixed size. This step introduces a protocol that divides long messages into chunks, thus accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details. A fourth step compiles Kalas's fixpoint operator (with substitution semantics) into recursive function definitions (with environment semantics), to align better with functional programming idioms. The final compilation phase compiles PAYLOAD's endpoints to CakeML [18], a sequential, functional programming language with a verified compiler, giving us semantics preservation down to the machine code.

Composing the compiler correctness results for each phase, we show that the deadlock freedom of Kalas carries over to the compiler output: the generated CakeML code never aborts with a runtime error, and – by the CakeML compiler correctness theorem – neither does the machine code (unless it runs out of memory).

As a convenient byproduct of CakeML's FFI modelcode is parameterised on primitives for sending and receiving messages, making it communication-backend agnostic. Thus, the same CakeML code can be used irrespective of whether the communication happens via (say) TCP/IP, MPI, or IPC, as long as these actions have the same semantics as the corresponding PAYLOAD primitives. Like other choreographic languages, our deadlock freedom guarantee depends on the rather strong assumptions implicit in the operational semantics: the backend stays live, and messages will never be lost in transit. In practice, our theorems are only as good as the backend's ability to abide by this.

As a proof-of-concept of our approach, we implemented a filter choreography and executed the generated code using an IPC communication backend on seL4 [17], a formally verified operating systems microkernel. Hence there is strong evidence, in the form of machine-checked proofs of functional correctness of the kernel [17] and the delivery guarantees of the component platform [9], that this backend is up to the task, even though we do not connect our proofs with the seL4 proofs.

This paper's main contributions are:

- the definition and verification of an end-to-end choreographic compiler, including:
- the proof of *endpoint projection*'s correctness w.r.t. Kalas's asynchronous semantics; and
- the implementation of a proof-of-concept choreography on top of seL4/CAmkES.

All definitions and proofs in this paper are mechanised in HOL4 [24] and available online.[1]

[1] https://github.com/CakeML/choreo/.

## 2 Kalas: A Choreographic Language

In this section we introduce our choreographic language, Kalas. To build an intuition for how choreographies operate, consider a common situation in component-based systems: a producer wishes to send a stream of messages to a consumer, but the consumer can only receive messages of a certain form. A filter that discards malformed messages is inserted.

▶ **Example 1** (Message filter - Choreography).

```
1.  while(true) do
2.    let v@producer = next_msg() in
3.    producer.v -> filter.temp;
4.    let test@filter = test(temp) in
5.    if test@filter then
6.      filter -> consumer[T];
7.      filter.temp -> consumer.v
8.    else filter -> consumer[F]
```

We assume a function `next_msg` to obtain the next message, which is then stored in the `producer`'s local variable $v$ (line 2). The `producer` then communicates the contents of $v$ to the `filter` which stores it locally in *temp* (line 3). The `filter` computes `test`(*temp*) (line 4). If `test`(*temp*) is true (line 5), we inform the consumer that a message is coming (line 6), and forwards the contents of *temp* to the `consumer` (line 7). Otherwise, we inform the `consumer` that a message was dropped (line 8).

This example highlights two important features: a choreography captures both the concrete behaviour of its participants and a global view of the communication occurring between them. That is, interactions between endpoints are presented together with local computation, e.g., `test` above. This allows individual endpoints to be translated into complete sequential programs. Second, communication mismatches are impossible by construction: if no message is forthcoming, the `consumer` will never be stuck waiting for one.

### 2.1 Syntax and Semantics

Kalas is similar to Core Choreographies (CC) [5], but features arbitrary local computation and asynchronous communication. The main datatype under consideration in our choreography language is strings or, to be precise, finite sequences of bytes. Strings are used as endpoint names ($p_i$), variable names ($v_i$), process variables ($X$), and as the concrete data that gets bound to variables and transmitted between endpoints ($d$). The use of strings as the value represents a separation of concerns: after marshalling and unmarshalling, local computations have full access to HOL's strongly typed language, but the choreography language is only concerned with data as it is really transmitted, namely as strings. The booleans (ranged over by $b$) are written $\mathsf{T}$ and $\mathsf{F}$. When we use booleans where strings are expected, we tacitly identify $\mathsf{T}$ with [0x01], and $\mathsf{F}$ with [0x00]. We use $a$ to range over the union of strings and booleans, and $f$ to range over functions of type $\mathsf{string}^* \rightarrow \mathsf{string}$.

▶ **Definition 2** (Kalas syntax). *Choreographies in Kalas, ranged over by $C$, are inductively defined by the grammar*

$$
\begin{array}{llllll}
C & ::= & p_1.v_1 \twoheadrightarrow p_2.v_2; C & \textit{(com)} & p_1 \twoheadrightarrow p_2[b]; C & \textit{(sel)} \\
 & & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} & \textbf{let } v@p = f(\tilde{v}) \textbf{ in } C & \textit{(let)} \\
 & & \mu X.\ C & \textit{(fix)} & X & \textit{(var)} \\
 & & \mathbf{0} & \textit{(nil)} & &
\end{array}
$$

**Table 1** Kalas semantics: communication rules. The function $\mathrm{wv}(\alpha)$ returns the variable (if any) that is modified by $\alpha$.

$$\textsc{Com}\ \frac{s\,(v_1, p_1) = d \quad p_1 \neq p_2}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[\epsilon]{p_1.v_1 \twoheadrightarrow p_2.v_2} s[(v_2, p_2) := d] \rhd C}$$

$$\textsc{Com-S}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \notin \mathrm{fp}(\alpha) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C'}$$

$$\textsc{Com-A}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \in \mathrm{fp}(\alpha) \quad \mathrm{wv}(\alpha) \neq (v_1, p_1) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C \xrightarrow[(p_1.v_1 \twoheadrightarrow p_2.v_2)::l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2;\, C'}$$

The prefix *(com)* sends the data bound to variable $v_1$ at endpoint $p_1$ to endpoint $p_2$ which stores it in variable $v_2$, *(sel)* communicates the selection of a branch from $p_1$ to $p_2$, *(if)* branches over the value in variable $v$ at process $p$, and *(nil)* is the empty choreography. *(let)* performs local computation, taking all values bound to the variables $\widetilde{v}$ at endpoint $p$ and applies them as arguments to the function $f$. The result is then stored in $v$. Note that we do not commit to any particular syntax for functions; rather, $f$ is a function in the meta-language in which Kalas is defined. In our case, the meta-language is higher-order logic (Hol). Hence our syntax is only concerned with interaction and branching of endpoints, offloading computation to Hol. This flexibility is useful for specifying open systems, or systems with legacy components: the internal behaviour of an endpoint that we have no control over can be modelled by functions that are non-computable, underspecified, or even completely uninterpreted, and the compiler can ignore such endpoints for code generation. For endpoints that we do intend to project, we require that the $f$'s used in their let-bindings be "sufficiently code-like" – otherwise, code generation will fail. This excludes, for example, functions that use Hilbert choice, sets or quantifiers.

Finally, *(fix)* supports choreographies with infinite behaviour. These can be unfolded, taking e.g. $\mu X.\ p_1 \twoheadrightarrow p_2[b]; X$ to $p_1 \twoheadrightarrow p_2[b]; \mu X.\ p_1 \twoheadrightarrow p_2[b]; X$. The above example also illustrates the only use of *(var)*: as a placeholder for fixpoint unfolding. The **while** loop used in Example 1 is syntactic sugar for *(fix)*.

We will use $\mathrm{fv}(C)$ to refer to the free variables of a choreography $C$, where each variable is paired with the name of the process that owns the variable. The binding operators are **let** and $p_1.v_1 \twoheadrightarrow p_2.v_2; C$, where $(v1, p1)$ is considered free and $(v2, p2)$ is considered bound.

The operational semantics is inductively defined, with some sample rules given in Table 1. Transitions are labelled to indicate both the action being performed (upper $\alpha$), and the trace (lower $l$) of deferred asynchronous actions. We explain the latter mechanism below. We refer to both labels and prefixes as actions, since they directly correspond to all operations that can be performed in the language. A *store* $s$ is a partial function $\mathsf{string} \times \mathsf{string} \hookrightarrow \mathsf{string}$ representing a global view of the endpoints' variable binding environment: $s(v, p)$, if defined, denotes the value bound to $v$ in $p$'s binding environment. In Hol, we use option types in the range to encode this partiality; much of the following presentation elides the logic's special handling of this (e.g., the Some and None constructors).

Kalas uses non-blocking, asynchronous communication. Hence, a sender process should be able to perform further actions before the message has arrived at the receiver. The semantics captures this by allowing an action $\alpha$ to occur before other interactions, provided only the

sender process is present in $\alpha$. A trace of every action that was skipped over is kept, to ensure the consistency between asynchrony and concurrency rules. This trace is used in the rule for **if**, which requires that both branches defer the same actions, though not necessarily in the same order. This constraint guarantees that regardless of the choice of branch, the asynchronous actions that need to be deferred in order to perform $\alpha$ are the same for each of the processes involved, implying that $\alpha$ is independent of the branching caused by the guard.

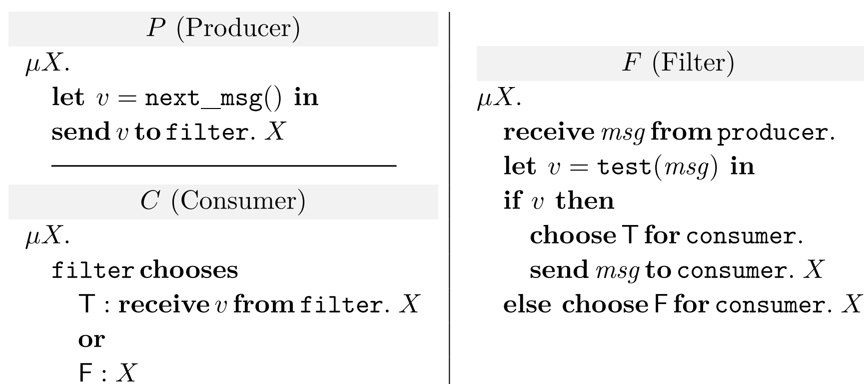
We prove that the resulting semantics is locally confluent, which will turn out to be immensely important for taming the proofs. As a sanity check of our rather involved labels, we also show completeness with respect to a similar semantics (not shown here) with structural congruence instead of swapping rules.

## 3 Endpoint Projection

The first phase of our compiler is *endpoint projection*, where we translate Kalas into Endpoint, our first intermediate language.

Continuing with Example 1, when we apply endpoint projection to the producer-consumer-filter choreography ($PCF$), we obtain $[\![PCF]\!]_{\mathsf{E}} = P \mid C \mid F$, comprised of the following endpoints running in parallel:

▶ **Example 3** (Message filter – Endpoint).



```
P (Producer)
μX.
  let v = next_msg() in
  send v to filter. X

C (Consumer)
μX.
  filter chooses
    T : receive v from filter. X
    or
    F : X

F (Filter)
μX.
  receive msg from producer.
  let v = test(msg) in
  if v then
    choose T for consumer.
    send msg to consumer. X
  else choose F for consumer. X
```

Here the (Filter) endpoint receives a message from the producer and, depending on the output of `test`, communicates its choice of branch to the consumer. Conversely, the (Consumer) decides based on the filter's choice whether it should await a message or whether the message was dropped. Finally, the (producer) obtains a message and sends it to the filter. Note that no branching or choice is required in the producer, since it behaves the same whether `test` succeeds or not.

### 3.1 Endpoint: Syntax and Semantics

Endpoint inherits many design decisions from Kalas, but splits unitary Kalas systems into two layers: the endpoint layer is purely sequential, and the *network* layer is a parallel composition of endpoints, each with its own name, queue and binding environment.

A *queue* $q$ is a function $\mathsf{string} \to \mathsf{string}^*$. The value $q(p)$ is the sequence of messages, from first to last, received from *process* $p$ but not yet read. Let $q + (p, a)$ be $q$ with $a$ appended to the end of $q(p)$, and $q - p$ be $q$ with the first element of $q(p)$ removed; if $q(p)$ is empty, $q - p$ is undefined. An *environment* $e$ is a partial function from variable names to values.

**Table 2** Endpoint semantics: communication rules.

$$\textsc{Send}\ \frac{e\ v\ =\ d \quad p_1 \neq p_2}{(p_1, e, q) \rhd \mathbf{send}\, v\, \mathbf{to}\, p_2.P \xrightarrow{p_1 \to p_2 : d} (p_1, e, q) \rhd P}$$

$$\textsc{Enqueue}\ \frac{p_1 \neq p_2}{(p_2, e, q) \rhd P \xrightarrow{p_2 \leftarrow p_1 : d} (p_2, e, q\ +\ (p_1, d)) \rhd P}$$

$$\textsc{Dequeue}\ \frac{q(p_2)\ =\ d :: \widetilde{a} \quad p_1 \neq p_2}{(p_1, e, q) \rhd \mathbf{receive}\, v\, \mathbf{from}\, p_2.P \xrightarrow{\tau} (p_1, e[v\ :=\ d], q\ -\ p_2) \rhd P}$$

▶ **Definition 4** (Endpoint syntax).

| | | | | | |
|---|---|---|---|---|---|
| $P, Q$ | := | $\mathbf{send}\, v\, \mathbf{to}\, p.P$ | *(output)* | $\mathbf{let}\ v = f(\widetilde{v})\ \mathbf{in}\ P$ | *(let)* |
| | | $\mathbf{receive}\, v\, \mathbf{from}\, p.P$ | *(input)* | $\mu X.P$ | *(fix)* |
| | | $\mathbf{choose}\, b\, \mathbf{for}\, p.P$ | *(internal choice)* | $X$ | *(var)* |
| | | $p\, \mathbf{chooses}\, \mathsf{T} : P\, \mathbf{or}\, \mathsf{F} : Q$ | *(external choice)* | $\mathbf{0}$ | *(nil)* |
| | | $\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q$ | *(if)* | | |
| $N$ | := | $N_1 \mid N_2$ | *(parallel)* | $\mathbf{0}$ | *(nil)* |
| | | $(p, e, q) \rhd P$ | *(endpoint)* | | |

Table 2 shows three representative rules from ENDPOINT's operational semantics. $\mathbf{send}\, v\, \mathbf{to}\, p.P$ represents an endpoint ready to send the contents of variable $v$ to $p$, using the SEND rule; the ENQUEUE rule allows a message thus sent to arrive in $p$'s queue. $\mathbf{receive}\, v\, \mathbf{from}\, p.P$ denotes a process ready to dequeue a message from its queue originating from $p$, and bind the contents of the message to the variable $v$ (DEQUEUE); if there is no message from $p$, the endpoint is blocked until one arrives. Similarly, $\mathbf{choose}\, b\, \mathbf{for}\, p.P$ represents an endpoint ready to tell process $p$ that it has chosen the $b$-branch. The corresponding INTCHOICE rule (elided) interacts with ENQUEUE to add the choice to $b$'s message queue. $p\, \mathbf{chooses}\, \mathsf{T} : P\, \mathbf{or}\, \mathsf{F} : Q$ represents a process waiting for $p$ to communicate its choice of branch. If it finds a $\mathsf{T}$ from $p$ in the queue, it proceeds as $P$; if it finds something else from $p$, it proceeds as $Q$.

## 3.2 Endpoint projection

The main complication when defining endpoint projection is how to handle **if** statements, which are not always projectable. For an example, consider the choreography

$$\mathbf{if}\ \text{Alice@}v\ \mathbf{then}\ \text{Bob}.v \Rightarrow \text{Alice}.v\ \mathbf{else}\ \text{Alice}.v \Rightarrow \text{Bob}.v$$

where Alice makes an internal choice, and depending on the result, either Alice sends a message to Bob, or vice versa. How does Bob know whether to send or receive?

We need a projectability criterion that rules out such degenerate cases. Our criterion is, intuitively: whenever Alice chooses an **if** branch, every other endpoint whose projection depends on the choice must immediately be told which branch was chosen. Hence, the example above can be made projectable by adding selections as follows:

$$\mathbf{if}\ \text{Alice@}v\ \mathbf{then}\ \text{Alice} \Rightarrow \text{Bob}[\mathsf{T}]; \text{Bob}.v \Rightarrow \text{Alice}.v$$
$$\mathbf{else}\ \text{Alice} \Rightarrow \text{Bob}[\mathsf{F}]; \text{Alice}.v \Rightarrow \text{Bob}.v$$

**Table 3** Projection and projectability, with the *(sel)* case, which is similar to *(com)*, elided. Partiality is indicated with a result of $\bot$. Recursive calls (e.g., in the let case) that fail propagate that undefinedness to the top-level.

$$
\begin{aligned}
\mathsf{pr}_p(\gamma, \mathbf{0}) &= \mathbf{0} \\
\mathsf{pr}_p(\gamma, p_1.v_1 \twoheadrightarrow p_2.v_2; C) &= \begin{cases} \bot & \text{if } p_1 = p_2 = p \\ \textbf{send}\, v_1\, \textbf{to}\, p_2.\mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \neq p_2 \\ \textbf{receive}\, v_2\, \textbf{from}\, p_1.\mathsf{pr}_p(\gamma, C) & \text{if } p \neq p_1 = p_2 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases} \\
\mathsf{pr}_p(\gamma, \textbf{let } v@p_1 = f(\widetilde{v}) \textbf{ in } C) &= \begin{cases} \textbf{let } v = f(\widetilde{v}) \textbf{ in } \mathsf{pr}_p(\gamma, C) & \text{if } p = p_1 \\ \mathsf{pr}_p(\gamma, C) & \text{otherwise} \end{cases} \\
\mathsf{pr}_p(\gamma, \mu X.C) &= \begin{cases} \mu X.\mathsf{pr}_p(\gamma[X := \mathsf{procs}(C)], C) & \text{if } p \in \mathsf{procs}(C) \\ \mathbf{0} & \text{otherwise} \end{cases} \\
\mathsf{pr}_p(\gamma, X) &= \begin{cases} \bot & \text{if } X \notin \mathrm{dom}(\gamma) \\ X & \text{if } p \in \gamma(X) \\ \mathbf{0} & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\mathsf{pr}_p(\gamma, \textbf{if } v@p_1 \textbf{ then } C_1 \textbf{ else } C_2)
$$

$$
= \begin{cases} \textbf{if } v \textbf{ then } \mathsf{pr}_p(\gamma, C_1) \textbf{ else } \mathsf{pr}_p(\gamma, C_2) & \text{if } p = p_1 \\ p_1\, \textbf{chooses}\, \mathsf{T} : \mathsf{pr}_p(\gamma, C_1')\, \textbf{or}\, \mathsf{F} : \mathsf{pr}_p(\gamma, C_2') & \text{if } p \neq p_1 \text{ and } \mathsf{sp}_{p_1,p}(C_1) = (\mathsf{T}, C_1') \\ & \text{and } \mathsf{sp}_{p_1,p}(C_2) = (\mathsf{F}, C_2') \\ \mathsf{pr}_p(\gamma, C_1) & \text{if } p \neq p_1 \text{ and } \mathsf{pr}_p(\gamma, C_1) = \mathsf{pr}_p(\gamma, C_2) \\ \bot & \text{otherwise} \end{cases}
$$

To formalise this criterion, we use the auxiliary function sp to split off initial selections pertaining to a pair of endpoints and check which branch was chosen.

▶ **Definition 5** (Split selections). *The partial function* sp *is inductively defined as follows (in all other cases,* sp *is undefined)*

$$
\mathsf{sp}_{p_1,p_2}(p_3 \twoheadrightarrow p_4[b]; C) = \begin{cases} (b, C) & \textit{if } p_1 = p_3 \textit{ and } p_2 = p_4 \\ \mathsf{sp}_{p_1,p_2}(C) & \textit{if } p_1 = p_3 \textit{ and } p_2 \neq p_4 \\ \text{undefined} & \textit{otherwise} \end{cases}
$$

Fixpoints motivate some additional projectability criteria: (i) orphan *(var)* statements are not allowed, and (ii) the projection of a *(fix)* statement for endpoints that do not appear in its body should be $\mathbf{0}$ (otherwise, the compiler introduces divergence). To enforce these requirements, we use a *fixpoint context* $\gamma$, a partial function from process names to sets of endpoint names that keeps track of which endpoints occur in the body of each *(fix)* statement.

We define a single partial function pr that given an endpoint name, a fixpoint context, and a choreography, returns an endpoint (its projection), if it exists.

▶ **Definition 6** (Projection and projectability). *A choreography $C$ is* projectable *if for all $p \in \mathsf{procs}(C)$, $\mathsf{pr}_p(\epsilon, C)$ is defined. The* projection *of the endpoints $\widetilde{p}$ from a choreography $C$ with state $s$ is defined as $[\![s \triangleright C]\!]_{\mathsf{E}}^{\widetilde{p}} = \Pi_{p_i \in \widetilde{p}}.\ (p_i, s\!\downarrow_{p_i}, \epsilon) \triangleright \mathsf{pr}_{p_i}(\epsilon, C)$ where $\Pi$ denotes iterated parallel composition, $s\!\downarrow_p$ denotes $\lambda v.s(p, v)$, $\epsilon$ is an empty fixpoint context, and* pr *is defined inductively by the equations in Table 3. $[\![s \triangleright C]\!]_{\mathsf{E}}$ abbreviates $[\![s \triangleright C]\!]_{\mathsf{E}}^{\mathsf{procs}(C)}$.*

# 4 Refining Choice

In Phase II, we implement ENDPOINT's choice primitives using send and receive actions. This simplifies reasoning about later compilation phases and the implementation of communication backends, which only need to consider two message-passing primitives instead of four.

After refining choice from the parallel composition $P \mid C \mid F$ in Example 3, we obtain $[\![P \mid C \mid F]\!]_{\mathsf{C}} = P \mid C' \mid F'$, where the producer $P$ is unchanged because it uses no choice constructs. The filter and consumer are compiled as follows:

▶ **Example 7** (Message filter – Refining choice).

| $F'$ (Filter) | $C'$ (Consumer) |
|---|---|
| $\mu X.$ | $\mu X.$ |
| **receive** $msg$ **from** `producer`. | **receive** $v$ **from** `filter`. |
| **let** $test = \mathtt{test}(msg)$ **in** | **if** $v$ **then** |
| **if** $test$ **then let** $v = \mathsf{T}$ **in** | **receive** $v$ **from** `filter`. $X$ |
| **send** $v$ **to** `consumer`. | **else** $X$ |
| **send** $msg$ **to** `consumer`. $X$ | |
| **else let** $v = \mathsf{F}$ **in** | |
| **send** $v$ **to** `consumer`. $X$ | |

The phase is mostly straightforward: internal choice is encoded as sending a boolean value, and external choice is encoded as receiving a value, storing it in a temporary variable $v$, then branching on it using **if**.

▶ **Definition 8** (Phase II). *The compilation function is homomorphic on all operators except internal and external choice, where it is defined as follows for any $v$ not free in $P, Q$:*

$$[\![\textbf{choose}\, b\, \textbf{for}\, p.P]\!]_{\mathsf{C}} = \textbf{let}\ v = (\lambda x.b)\epsilon\ \textbf{in send}\, v\, \textbf{to}\, p.[\![P]\!]_{\mathsf{C}}$$

$$[\![p\, \textbf{chooses}\, \mathsf{T} : P\, \textbf{or}\, \mathsf{F} : Q]\!]_{\mathsf{C}} = \textbf{receive}\, v\, \textbf{from}\, p.(\textbf{if}\ v\ \textbf{then}\ [\![P]\!]_{\mathsf{C}}\ \textbf{else}\ [\![Q]\!]_{\mathsf{C}})$$

Since $v$ is not used further in the continuation, it can be reused for subsequent choice encodings, meaning that in practice, a single fresh name suffices.

The design of the ENDPOINT semantics anticipates this compilation phase, by allowing type confusion between boolean values and string values. This feature, which may seem otherwise undesirable, makes branch selection messages indistinguishable from other messages. This makes the difference between $N_{\mathsf{E}}$ and $[\![N_{\mathsf{E}}]\!]_{\mathsf{C}}$ unobservable by other processes.

# 5 Splitting Large Messages

In both ENDPOINT and our source language, transmitting a message of arbitrary size is a single atomic operation, whether it carries one bit or one terabyte of information. This is convenient for the programmer, but doesn't reflect how real communication protocols work.

Our second compiler phase introduces a protocol that divides long messages into chunks (see Figure 2), accounting for the fact that real communication protocols have bounds on message size, without burdening the application programmer with the details.

For these purposes, we introduce another intermediate representation, PAYLOAD, which is similar to ENDPOINT except messages have a fixed size. It turns out that this compiler phase is more proof-relevant than compiler implementation-relevant. Syntactically, the compilation function $[\![\cdot]\!]_{\mathsf{C}}$ from ENDPOINT to PAYLOAD is essentially the identity function. Semantically, send and receive actions are no longer atomic, which leads to a combinatorial explosion in the number of possible interleavings. The associated proof complications are largely mitigated by observing that the target terms are always locally confluent.

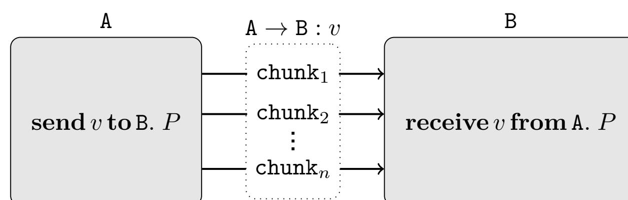



**Figure 2** Messages split into chunks.

Payload is parameterised by a *payload size* $\sigma > 0$. Unlike in previous languages where messages can have arbitrary size, here messages in transit must be exactly $\sigma + 1$ bytes long. Longer messages are transmitted in chunks, and shorter messages are padded; the extra byte encodes the bookkeeping necessary to realise this. In particular, we must track whether a given chunk ends a message, or whether it will be continued in future messages.

▶ **Definition 9** (Payload syntax, I). *The syntax of Payload is obtained by removing endpoint, input, output and choice from Endpoint, and adding:*

$(p, e, \eta, q) \rhd P$ *(endpoint)*

$\textbf{send}\ v_n\ \textbf{to}\ p.P$ *(output)*

$\textbf{receive}\ v\ \textbf{in}\ \langle d \rangle\ \textbf{from}\ p.P$ *(input)*

The message-splitting aspect of Payload's semantics is embodied here: the new input and output prefixes record how far along in a transmission we are. Hence $\textbf{send}\ v_n\ \textbf{to}\ p.P$ will send the value of $v$ to $p$, starting from the $n$-th byte, divided into as many chunks as necessary, one chunk at a time. Similarly $\textbf{receive}\ v\ \textbf{in}\ \langle d \rangle\ \textbf{from}\ p.P$ will receive chunks from $p$, recording every intermediate chunk in the temporary buffer $d$. When a final chunk arrives, all received chunks are concatenated and bound to the variable $v$.

The state component $\eta$ is a closure environment; we will discuss it in Section 6.1, where the operators that need it are introduced. For space reasons, readers interested in the operational semantics are referred to the formalisation.

## 6 Introducing Closures

Finally, before we transition from Payload to CakeML, we introduce closures. That is, we translate all instances of the fixpoint operator $\mu$ into a **letrec** primitive, to better match CakeML's representation of recursive functions. Though CakeML supports global, updateable variables (SML's `ref` types), reasoning is much simpler if one remains "purely functional", and uses parameters with closures and local, immutable bindings. Thus: variables written to in the fixpoint body (ultimately from the Kalas source) must be made function parameters.

Why not just put **letrec** in the source language, if we have to add it later anyway? Briefly, it becomes technically complicated to maintain a consistent view of the global environment in the presence of out-of-order execution. Fixpoint semantics do not need environments. See Section 9 for a comparison with related approaches.

### 6.1 Closures: syntax and semantics

Recall from Section 5 that endpoint states contain a closure environment $\eta$. It is a mapping from function names to *closures*. Closures are triples $(e, \eta, \lambda\widetilde{x}.P)$, where: $e, \eta$ are the local variable environments and closure environments, respectively; $\widetilde{x}$ is the function's parameters; and $P$ is the function body. Note that closures and environments are mutually recursive.

▶ **Definition 10** (Payload syntax, II). *These augment the operators from Definition 9.*

$\textbf{letrec } F(\widetilde{d}) = P$ *(letrec)* $F(\widetilde{v})$ *(call)*

*(letrec)* and *(call)* are function definitions and function calls, respectively. They are similar to *(fix)*, but use environment semantics instead of substitution semantics. Note that **letrec** has no continuation; instead, the defined function will be called immediately. This suffices for our purposes, which is to use **letrec** to encode $\mu$.

## 6.2 Compilation

To compile the fixpoint operator into the letrec operator, the basic idea is the obvious one: a fixpoint binder $\mu X$ becomes a recursive function definition, and a process variable $X$ becomes a function call. But what arguments should we give the function?

To prepare for compilation to CakeML, the main goal is to make sure we use the constructs that mutate variables (let and input) consistently with functional programming idioms. But we have a second, conflicting goal. To simplify proofs, we want the local variable environment of target terms to be identical to the global environment of their source terms. The following table illustrates the options we considered:

| Source |
| --- |
| $\textbf{let } x, y = \ldots \textbf{ in } \mu X.\, \textbf{let } x = \mathtt{f}(x, y) \textbf{ in } \textbf{send } x \textbf{ to } \mathtt{p}.\textbf{receive } z \textbf{ from } \mathtt{p}.X$ |

| Target I | Target II | Target III |
| --- | --- | --- |
| $\textbf{let } x, y = \ldots \textbf{ in}$ | $\textbf{let } x, y = \ldots \textbf{ in}$ | $\textbf{let } x, y = \ldots \textbf{ in}$ |
| $\textbf{letrec } X(x, y, z) =$ | $\textbf{letrec } X(x, z) =$ | $\textbf{letrec } X(x) =$ |
| $\textbf{let } x = \mathtt{f}(x, y) \textbf{ in}$ | $\textbf{let } x = \mathtt{f}(x, y) \textbf{ in}$ | $\textbf{let } x = \mathtt{f}(x, y) \textbf{ in}$ |
| $\textbf{send } x \textbf{ to } \mathtt{p}.$ | $\textbf{send } x \textbf{ to } \mathtt{p}.$ | $\textbf{send } x \textbf{ to } \mathtt{p}.$ |
| $\textbf{receive } z \textbf{ from } \mathtt{p}.$ | $\textbf{receive } z \textbf{ from } \mathtt{p}.$ | $\textbf{receive } z \textbf{ from } \mathtt{p}.$ |
| $X(x, y, z)$ | $X(x, z)$ | $X(x)$ |

Target I represents the simplest compilation strategy that could possibly work: every program variable becomes a parameter of every function. This, however, is rather wasteful: $y$ is never modified within the function, so there's no need to pass it around; $z$ is modified in the body, but not subsequently read, so there is no need to remember its value between calls.

Taking all this into account would yield Target III, which is to take as function parameters only those variables that may be read before they are written to within a fixpoint's body. Unfortunately, this is not compatible with our second goal above: while the source term retains the value of $z$ between subsequent fixpoint unfoldings, Target III will restore $z$ to its value at the point of $X$'s definition at each recursive call.

As a compromise, we opt for Target II: the function parameters are the variables that may be written to within the body of the fixpoint expression. To this end, we let the function $\mathrm{wv}(e)$ return all variables that are modified (by **let** or receive) in $e$.

▶ **Definition 11** (Phase IV). *The compilation function* $[\![\cdot]\!]^{\gamma}_{\mathsf{F}}$ *is homomorphic on all operators except: letrec and call, where it is undefined; and fix and val, where it is*

$$[\![\mu X.P]\!]^{\gamma}_{\mathsf{F}} = \textbf{letrec } X(\mathrm{wv}(P)) = [\![P]\!]^{\gamma[X:=\mathrm{wv}(P)]}_{\mathsf{F}} [\![X]\!]^{\gamma}_{\mathsf{F}} = X(\gamma(X))$$

*In the above,* $\gamma$ *is a partial function from process variables to lists of local variables.*

A further minor complication is that a variable can be used as a function argument before its definition. We could add support for optional arguments, but since CakeML has no such feature we would eventually have to compile them away. Our fix is that before we apply $[\![X]\!]_{\mathsf{F}}$, we add a prelude to each endpoint that initialises all variables to a default value.

# 7 Compiler Correctness

In this section, we discuss the compiler correctness theorem connecting Kalas to Payload with Letrec, and its proof.

## 7.1 Theorem Statement

Let $[\![\cdot]\!]^{\widetilde{p}}$ denote the composition $[\![\cdot]\!]^{\widetilde{p}}_{\mathsf{E}} \circ [\![\cdot]\!]_{\mathsf{C}} \circ [\![\cdot]\!]_{\mathsf{P}} \circ [\![\cdot]\!]^{\epsilon}_{\mathsf{F}}$ and let $[\![C]\!] = [\![C]\!]^{\mathsf{procs}(C)}$. We prove weak operational correspondence up-to strong bisimilarity (denoted $\dot{\sim}$) for $[\![\cdot]\!]^{\widetilde{p}}$:

▶ **Theorem 12.** *If $c$ is a projectable choreography and $\mathrm{fv}(c) \subseteq \mathrm{dom}(s)$, then*

1. *(Operational completeness) If $s \rhd C \implies s' \rhd C'$ then there exist $s'', C'', N_{\mathsf{F}}$ such that $s' \rhd C' \implies s'' \rhd C''$ and $[\![s \rhd C]\!] \implies N_{\mathsf{F}}$ and $N_{\mathsf{F}} \dot{\sim} [\![s'' \rhd C'']\!]^{\mathsf{procs}(C)}$*
2. *(Operational soundness) If $[\![s \rhd C]\!] \implies N_{\mathsf{F}}$ then there exist $s', C', N'_{\mathsf{F}}$ such that $N_{\mathsf{F}} \implies N'_{\mathsf{F}}$ and $s \rhd C \implies s' \rhd C'$ and $N'_{\mathsf{F}} \dot{\sim} [\![s' \rhd C']\!]^{\mathsf{procs}(C)}$*

Here $\implies$ over networks denotes $\xrightarrow{\tau}^{\star}$, and $\implies$ over choreographies denotes $(\bigcup_{a,l} \xrightarrow[l]{a})^{\star}$. Our presentation of *operational completeness* requires a catch-up transition because projectability is not, in general, preserved by reduction. However, any non-projectable choreography reachable from a projectable choreography can always reduce to a projectable choreography.

One important consequence of Theorem 12 is that the compiler output is deadlock-free:

▶ **Theorem 13** (Network-level deadlock-freedom). *If $C$ is a projectable choreography, and $\mathrm{fv}(C) \subseteq \mathrm{dom}(s)$, and $[\![s \rhd C]\!] \implies N_{\mathsf{F}}$, then either all endpoints in $N_{\mathsf{F}}$ are* Nil*, or there exists $N'_{\mathsf{F}}$ such that $N_{\mathsf{F}} \xrightarrow{\tau} N'_{\mathsf{F}}$*

## 7.2 On the proofs

As the reader may expect, we prove soundness and completeness separately for each compilation phase before composing the theorems. A common theme is strategic use of confluence to reduce the number of interleavings we must consider.

The proof of *operational completeness* for Phase I leverages local confluence to simplify reasoning in a major way. The asynchrony and swapping rules in Kalas's semantics, which are otherwise a pain point, play no role in these proofs. This is because any reduction involving them has a common successor with a reduction that only uses the syntax-directed rules (e.g., rule Com from Table 1). This yields a simpler proof than, for example, Montesi [20, Appendix C]; his language is also confluent, yet his proof makes no use of this, and includes cases for the swapping and asynchrony rules.

To prove *operational soundness* we use a technique based on *inert reduction*, first conceived by van Glabbeek to study encodings from the synchronous to the asynchronous $\pi$-calculus [28]. Intuitively, an inert reduction is one that performs a bookkeeping step without committing to a branch. We say that $N_{\mathsf{E}} \longrightarrow N'_{\mathsf{E}}$ is *inert* if for every $N''_{\mathsf{E}} \neq N'_{\mathsf{E}}$ such that $N_{\mathsf{E}} \longrightarrow N''_{\mathsf{E}}$, there is an $N'''_{\mathsf{E}}$ such that $N'_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$ and there is an inert transition $N''_{\mathsf{E}} \longrightarrow N'''_{\mathsf{E}}$. The key insight is that for encodings that only use inert catch-up transitions, operational soundness can be proven by induction on the length of the reduction sequence. Moreover, since inertness is a form of confluence, it suffices to consider just one interleaving of the intermediate steps, namely the one that directly mimics one source-language step at a time. All our catch-up

transitions are inert, which makes the proof of *operational soundness* much more tractable, with roughly half the effort going into proving confluence. The same technique is also used to great effect to tame the interleaving explosion of Phase III.

Phase II uses a traditional invariant-based technique, which we found intractable for the other phases with more complicated interleavings. The main headache here is alpha-equivalence considerations arising from the need to invent fresh names.

The proofs for Phase IV are different. In the other phases, the bulk of the effort is chasing transitions. Here, that part is trivial since we have one-to-one transition correspondence (up-to strong bisimilarity). The difficulty is in wrangling the candidate relation used to prove that the continuations of fixpoints and **letrec** unfoldings are bisimilar. The relation, which describes the precise relationship between closure environments and (possibly unfolded) fixpoints, is surprisingly complicated at almost 50 lines of HOL4 script. It is worth pointing out that this complicated relation entails no trust issues; its only use in the overall proof story is to witness an existential quantifier.

## 8 Compilation into CakeML

CakeML [18] is an impure, sequential, functional programming language similar to Standard ML. Its most notable feature is a compiler correctness proof in HOL4 that extends down to the machine code level for mainstream architectures such as x86-64 and ARM [25]. Interaction with the outside world is supported by a foreign function interface (FFI). We assume two foreign functions, `send` and `receive`, that support communication with the other endpoints. Compilation to CakeML consists of two parts: the static part, which is verified once and for all, and the dynamic part, which is proof-producing.

### 8.1 Static compiler

The static compilation is performed by the function $[\![\cdot]\!]_{\mathsf{ML}}$, which maps PAYLOAD endpoints to CakeML expressions. Its full definition would not fit here, but to show its flavour, $[\![\textbf{receive}\, v\, \textbf{in}\, \langle\epsilon\rangle\, \textbf{from}\, p.P]\!]_{\mathsf{ML}}$ produces the code

```
let val v =
  let val buff = Word8Array.array (σ + 1) 0
      fun receiveloop d =
        (#(receive) p buff;
         let val m = unpad buff
         in if final buff then concat(reverse(m::d))
            else let fun zerobuf(i) =
                           if i < 0 then ()
                           else (Word8Array.update(buff,i,0); zerobuf(i-1))
                 in zerobuf(Word8Array.length(buff)-1);
                    receiveloop(m::d)
                 end
         end)
  in receiveloop [] end
in [[P]]ML end
```

First, a receive buffer of size $\sigma+1$ is allocated. Then, the function `receiveloop` repeatedly calls the foreign function `#(receive)` until a final chunk from $p$ is received, zeroing the receive buffer between every message. All chunks of the message are unpadded, concatenated and finally bound to the variable $v$ before proceeding.

We use a small-step, relational (but deterministic) presentation of CakeML's semantics, allowing a natural expression of our eventual simulation theorem. We write $(p_0, cs_0) \rightarrow_c (p, cs)$, with $p_0$ the initial CakeML program, and $cs_0$ its accompanying state, to mean that this pair can evolve in a single step to $(p, cs)$. The states $cs_i$ contain FFI information (see below), the internal program state (variable environment, reference contents), as well as a continuation stack to track what remains to be done. The semantics is parametric on the behaviour of foreign functions: states include a freely chosen model of the outside world, and a freely chosen *oracle function* that describes how this model reacts to FFI calls.

We are interested in how generated CakeML code interacts with the choreography's other endpoints, so our FFI state models the outside world as triple $(p, q, N)$, with $p$ the name of the CakeML endpoint, $q$ its queue, and $N$ a Payload network that $p$ interacts with. There is an unfortunate mismatch here: the FFI model must be a function (CakeML is deterministic), but Payload's semantics is a one-to-many relation: when we receive a message from $N$, there is not in general a unique $N'$ that the network will reach after sending us our message, as actions internal to $N$ may or may not fire before $N$ sends the message. However, as long as all endpoints in $N$ have unique names (a reasonable invariant), Payload reductions and send actions are locally confluent. So whether such internal actions fired or not, the resulting states are observationally equivalent from $p$'s point of view.

Let $N \xRightarrow{p \rightarrow \widetilde{p}:\widetilde{d}} N'$ denote $N \Longrightarrow \xrightarrow{p \rightarrow p_0 : d_0} \Longrightarrow \ldots \xrightarrow{p \rightarrow p_n : d_n} \Longrightarrow N'$. We define the oracle so that when `#(send)` $p\,d$ executes in a state $(p_1, q, N)$, if there is no endpoint named $p$ in $N$, we abort with a run-time error; otherwise we produce a new state $(p_1, q + \widetilde{(p, d)} + \widetilde{(p', d')}, N')$, chosen with Hilbert Choice to satisfy $N \xRightarrow{\widetilde{p} \rightarrow p_1 : \widetilde{d}} \xrightarrow{p \leftarrow p_1 : d} \xRightarrow{\widetilde{p'} \rightarrow p_1 : \widetilde{d'}} N'$. That is, the network component $N'$ records its delivery of some number of messages to us (from $\widetilde{p}$), the delivery of our message $d$ to $p_1$, followed by its sending us possibly yet more messages (from $\widetilde{p'}$).

The semantics of `#(receive)` is similar, with the addition that the FFI call diverges if there is no reduction sequence causing a message to be enqueued. A key sanity check and technical lemma to show that this use of Hilbert choice is innocuous is the following:

▶ **Lemma 14** (FFI irrelevance). *Two CakeML steps starting from equal environments, equal expressions and bisimilar initial states yield bisimilar states and otherwise equal results.*

Let $N_p$ denote the endpoint named $p$ in $N$, and $N - p$ the network with that endpoint removed. Let $\mathsf{FFI}(cs)$ denote the FFI component of the CakeML state $cs$. Write $cs_1 =_{\mathit{ffi}} cs_2$ when $\mathsf{FFI}(cs_1)$ is bisimilar to $\mathsf{FFI}(cs_2)$ and all other components of the two states are equal.

▶ **Theorem 15** (Network Forward Correctness). *Let $N$ be a well-formed Payload network that includes an arbitrary endpoint $p$. Further, assume a CakeML state $cs$ that is appropriately related to $N$ (see below), with $\mathsf{FFI}(cs) = (p, q, N - p)$. Then, if $N$ can reduce to $N'$, there exist $cs'$, $mp$ (the "merge program"), $cs_1$ and $cs_2$ (two "merge states") such that*

- $\mathsf{FFI}(cs') = (p, q', N' - p)$ *and $cs'$ is appropriately related to $N'$ ;*
- $([\![N_p]\!]_{\mathsf{ML}}, cs) \rightarrow_c^* (mp, cs_1)$*;*
- $([\![N'_p]\!]_{\mathsf{ML}}, cs') \rightarrow_c^* (mp, cs_2)$*; and*
- $cs_1 =_{\mathit{ffi}} cs_2$*.*

The "appropriate relation" above between a network and a CakeML state requires that: all bindings of $N_p$ are present in the CakeML state's environment; our library functions (e.g., `List.drop`) are defined and have the expected behaviour; and for every function $f$ used in a let expression in $N_p$, a CakeML function that is a totally correct implementation of $f$ is present in the environment.

As CakeML is deterministic, Theorem 15 gives us that (i) all infinite traces in Payload are necessarily simulated by an infinite trace in CakeML, and (ii) compilation of a terminating choreography produces CakeML endpoints that will all also terminate successfully.

Though Theorem 15 tells us that every step taken by an endpoint will result in corresponding movement at the CakeML level, we have not transferred deadlock freedom to this level if the original choreography has only infinite paths. This is because currently, our theorems are not strong enough to rule out the possibility of *livelocks*: states where global progress is possible, but some nodes may be stuck waiting to receive. This is impossible by construction in Kalas, so while no such livelocks can occur (under weak fairness), operational correspondence by itself is only strong enough to guarantee global progress. One possible solution is to prove, in addition to operational correspondence, that an invariant stating "every receive can eventually be matched by a send" holds throughout the compilation chain.

## 8.2 Dynamic compiler by example

The dynamic compiler creates the initial environment assumed in Theorem 15, and proves that it is appropriate. The environment is built on top of the CakeML basis library by invoking CakeML's proof-producing code synthesis tool [21] on each function used in the endpoints' let expressions.

Kalas and the compiler are all deeply embedded in Hol4. Hence, users program choreographies by writing instances of the Hol4 datatype that encodes the choreography syntax. We define the system in Example 1 as a choreography `filter` where the producer has an infinite message stream, and where `test` is a simple function that checks if the message starts with "`A`" or not. To run the compiler, the invocation is

```
project_to_camkes builddir filename "filter";
```

This automatically performs the following tasks: (i) proves that the current environment is appropriate; (ii) evaluates the compiler in the logic to produce CakeML code for each of the three endpoints; (iii) produces end-to-end theorems for each endpoint by composing Theorems 12 and 15, discharging all assumptions; (iv) finally, generates all the glue code and build instructions necessary to create a complete system image that runs our choreography on top of the verified microkernel seL4 [17]. The system consists of three components in parallel, each running our generated CakeML code. The CakeML code is linked with a thin layer of C glue code that implements `send` and `receive` using the dataport and IPC mechanisms of the CAmkES [19] component platform. Thus: the user writes a choreography, calls `project_to_camkes`, and obtains a correctly compiled choreography running on a verified component platform on a verified microkernel.

# 9 Related Work

Session types [14] have seen extensive use in the $\pi$-calculus [15] and other concurrent languages [22, 29, 8, 16]. In recent years, the field has seen more mechanised proofs, perhaps motivated by past mistakes [31, 23]. In Castro et al. [3] a revised version of the session-typed $\pi$-calculus [31] is formalised in Coq [4]. Furthermore, Thiemann [27] proves type soundness and session fidelity in Agda [1] for an asynchronous functional session type language based on Gay et al. [10]. Tassarotti et al. [26] develop a higher-order concurrent logic, and verify a refinement procedure for a session-typed language as a case study.

Hallal et al. [12] synthesise the distributed components of a communicating system from a global choreography. Their result aims only to capture the communication logic of the system; by way of contrast, we consider local computation also.

Carbone and Montesi [2, 20] present a choreographic language with multi-party asynchronous session types (demonstrating the combination of the two approaches to great effect) along with a projection function into a variant of the calculus for multi-party sessions, with a proof – albeit pen-and-paper – of projection correctness. Kalas began as a simplified version of their language.

The most closely related work is two recent Coq formalisations of endpoint projection in different settings, by Cruz-Filipe *et al.* [6], and by Hirsch and Garg [13]. Cruz-Filipe et al. verify endpoint projection from CC (Core Choreographies) to a distributed process calculus. Hirsch and Garg [13] formalise endpoint projection from Pirouette, a higher-order functional choreographic language, where functions can return, and be parameterised on, choreographies. The most obvious difference between our work and these other papers is one of scope: both of [6, 13] formalise endpoint projection in isolation; for us, this is just the first step towards our goal of integrating endpoint projection into an end-to-end verified compilation toolchain that can be used to build real, runnable code.

Both CC and Pirouette are parameterised on a local language for describing computation, which is assumed to be available also in the target language. We achieve similar generality by representing local computation as shallow embeddings (functions in HOL4's logic). This lets us use a more abstract presentation, with no need to carry around an extra syntax, semantics, and associated well-formedness assumptions. The tradeoff is that we need a proof-producing (as opposed to verified) compiler phase to generate CakeML code.

In terms of semantics, one difference is that Kalas has asynchronous communication, whereas both CC and Pirouette are synchronous languages. Another interesting difference between the three languages is their representation of choreographies with infinite behaviour. Pirouette uses function closures. CC does not support the definition of local procedures, but executes in a context where a number of top-level, parameterless procedures are available. Kalas uses a fixpoint operator, which is parameterless, like CC, but supports arbitrary nesting of local procedures, like Pirouette.

CakeML has functions with closure semantics, but we nonetheless chose fixpoints over functions for Kalas. This is because, in an environment semantics, it is difficult to maintain a consistent view of the global environment in the presence of out-of-order execution: the semantics needs to track which local computations should be executed in the caller's environment (if they're ahead) or in the callee's environment (if they're behind). One solution is Cruz-Filipe et al. [6]'s approach, which breaks the abstraction of global, atomic actions by introducing an operator representing partially-completed procedure entry into the source language. Hirsch and Garg use an interesting approach, where function calls are considered global *both in source and target language*. In particular, executing a function call in a single endpoint has a CSP-like synchronous semantics where, as a single atomic action, the entire network performs the same function call together. While assuredly simplifying endpoint projection, this comes at the expense of complicated synchronisation when realising this in a distributed setting. In contrast, Kalas's unfolding of fixpoints can be implemented locally.

The target language used by Hirsch and Garg is a parallel composition of nodes expressed in the so-called *control language*. It mixes $\lambda$-calculus features with communication-enabling effects like send, receive and choose. This is rather like a functional language, which invites comparisons to our final target language, CakeML; but the role it plays in their development is much more akin to the role ENDPOINT plays in ours. Much like the relationship between Kalas and ENDPOINT, the feature set of Pirouette and the control language are essentially the same, except the latter is a localised representation.

Not all aspects considered by Hirsch and Garg, and by Cruz-Filipe et al., are present in our work. For example, Hirsch and Garg prove progress and preservation for an associated type system, while we do not consider types at all. In a companion paper, Cruz-Filipe *et al.* [7] prove that CC is Turing-complete, by showing that it can implement partial recursive functions. Turing completeness for Kalas is trivial because local computations may use arbitrary HOL functions.

## 10 Conclusion

We have presented what we believe to be the first end-to-end verified compiler for a choreographic language. After passing through five phases and two intermediate languages, our language, Kalas can be compiled to machine-code by reusing existing work from the CakeML project. Further, we have implemented a deployment on top of the micro-kernel seL4, itself also verified software. There, message-passing is implemented by IPC between separate user-processes.

There are a number of interesting directions for future work. Data types other than strings require a framework for verified marshalling and de-marshalling. Our model of the communication backend assumes unboundedly long message queues, which is arguably unrealistic. It would be interesting to investigate if deadlock freedom holds in a model where queues are bounded but not lossy. Alternative ITree-base [30] semantics (i.e., a co-inductive observational semantics) for Kalas and other intermediate languages, could significantly simplify projection proofs and allow for more lax projectability criteria. The CakeML compiler correctness theorem has an "unless the compiler output runs out of memory" side-condition, so liveness properties such as deadlock freedom carry over to the machine code only with this caveat, which could be discharged using CakeML's verified space-cost semantics [11]. We would also like to deploy on other communication backends, perhaps on top of TCP/IP, which would demonstrate verified distributed computation over the Internet.

## References


1. Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *J. Funct. Program.*, 27:e8, 2017. doi:10.1017/S0956796816000319.
2. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
3. David Castro, Francisco Ferreira, and Nobuko Yoshida. EMTST: engineering the meta-theory of session types. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 26th International Conference, TACAS 2020, Dublin, Ireland. Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2020. URL: http://doi.org/10.1007/978-3-030-45237-7_17, doi:10.1007/978-3-030-45237-7_17.
4. Coq Development Team. The Coq proof assistant, version 8.11.0, January 2020. doi:10.5281/zenodo.3744225.
5. Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.
6. Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021 – 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

7 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021. Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. URL: `http://doi.org/10.4230/LIPIcs.ITP.2021.15`, `doi:10.4230/LIPIcs.ITP.2021.15`.

8 Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming, 20th European Conference, Nantes, France. Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. `doi:10.1007/11785477_20`.

9 Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. Automated verification of RPC stub code. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods – 20th International Symposium, Oslo, Norway. Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2015. `doi:10.1007/978-3-319-19249-9_18`.

10 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. `doi:10.1017/S0956796809990268`.

11 Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):204:1–204:29, 2020. `doi:10.1145/3428272`.

12 Rayan Hallal, Mohamad Jaber, and Rasha Abdallah. From global choreography to efficient distributed implementation. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 756–763. IEEE, 2018. URL: `http://doi.org/10.1109/HPCS.2018.00122`, `doi:10.1109/HPCS.2018.00122`.

13 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. `doi:10.1145/3498684`.

14 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. `doi:10.1007/3-540-57208-2_35`.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Lisbon. Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. `doi:10.1007/BFb0053567`.

16 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

17 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. `doi:10.1145/1743546.1743574`.

18 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego*, pages 179–192. ACM, 2014. `doi:10.1145/2535838.2535841`.

19 Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: a component model for secure microkernel-based embedded systems. *J. Syst. Softw.*, 80(5):687–699, 2007. `doi:10.1016/j.jss.2006.08.039`.

20 Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. URL: `http://www.fabriziomontesi.com/files/choreographic_programming.pdf`.

21 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 115–126. ACM, 2012. `doi:10.1145/2364527.2364545`.

**22** Matthias Neubauer and Peter Thiemann. An implementation of session types. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004. `doi:10.1007/978-3-540-24836-1_5`.

**23** Randy Pollack. Closure under alpha-conversion. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1993. `doi:10.1007/3-540-58085-9_82`.

**24** Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montréal. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. `doi:10.1007/978-3-540-71067-7_6`.

**25** Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019. `doi:10.1017/S0956796818000229`.

**26** Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement. In Hongseok Yang, editor, *Programming Languages and Systems – 26th European Symposium on Programming, ESOP 2017, Uppsala, Sweden. Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 909–936. Springer, 2017. URL: `http://doi.org/10.1007/978-3-662-54434-1_34`, `doi:10.1007/978-3-662-54434-1_34`.

**27** Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In Ekaterina Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal*, pages 19:1–19:15. ACM, 2019. URL: `http://doi.org/10.1145/3354166.3354184`, `doi:10.1145/3354166.3354184`.

**28** Rob J. van Glabbeek. On the validity of encodings of the synchronous in the asynchronous $\pi$-calculus. *Inf. Process. Lett.*, 137:17–25, 2018. `doi:10.1016/j.ipl.2018.04.015`.

**29** Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. `doi:10.1016/j.tcs.2006.06.028`.

**30** Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. `doi:10.1145/3371119`.

**31** Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007. `doi:10.1016/j.entcs.2007.02.056`.