Logical Methods in Computer Science
Volume 20, Issue 3, 2024, pp. 26:1–26:50
https://lmcs.episciences.org/

Submitted Jul. 27, 2023
Published Sep. 23, 2024

HIGHER-ORDER ASYNCHRONOUS EFFECTS *

DANEL AHMAN [a] AND MATIJA PRETNAR [b,c]

[a] University of Tartu, Institute of Computer Science, Narva mnt 18, Tartu, Estonia
e-mail address: danel.ahman@ut.ee

[b] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, Ljubljana, Slovenia
e-mail address: matija.pretnar@fmf.uni-lj.si

[c] Institute of Mathematics, Physics and Mechanics, Jadranska 19, Ljubljana, Slovenia

Abstract. We explore asynchronous programming with algebraic effects. We complement their conventional synchronous treatment by showing how to naturally also accommodate asynchrony within them, namely, by decoupling the execution of operation calls into signalling that an operation's implementation needs to be executed, and interrupting a running computation with the operation's result, to which the computation can react by installing interrupt handlers. We formalise these ideas in a small core calculus and demonstrate its flexibility using examples ranging from a multi-party web application, to pre-emptive multi-threading, to (cancellable) remote function calls, to a parallel variant of runners of algebraic effects. In addition, the paper is accompanied by a formalisation of the calculus's type safety proofs in Agda, and a prototype implementation in OCaml.

1. Introduction

Effectful programming abstractions are at the heart of many modern general-purpose programming languages. They can increase expressiveness by giving programmers access to first-class (delimited) continuations, but often they simply help programmers to write cleaner code, e.g., by avoiding having to manage a program's memory explicitly in state-passing style, or getting lost in callback hell while programming asynchronous computations.

An increasing number of language designers and programmers are starting to embrace *algebraic effects*, where one uses algebraic operations [PP02] and effect handlers [PP13] to uniformly, modularly, and user-definably express a wide range of effectful behaviour, ranging from basic examples such as state, rollbacks, exceptions, and nondeterminism [BP15], to

Key words and phrases: algebraic effects, asynchrony, concurrency, interrupt handling, signals, promises.

* This paper is an extended version of our previous work [AP21]: it simplifies the meta-theory, removes the reliance on general recursion for reinstalling interrupt handlers, adds state to reinstallable interrupt handlers, and extends the calculus with higher-order signal and interrupt payloads, and with dynamic process creation.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 834146 . This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

DOI:10.46298/LMCS-20(3:26)2024

advanced applications in concurrency [SDW+21] and statistical probabilistic programming [BCJ+19], and even quantum computation [Sta15].

While covering many examples, the conventional treatment of algebraic effects is *synchronous* by nature. In it effects are invoked by placing operation calls in one's code, which then propagate outwards until they trigger the actual effect, finally yielding a result to the rest of the computation that has been *waiting* in a blocked state the whole time. While blocking the computation is indeed sometimes necessary, e.g., in the presence of general effect handlers that can execute their continuation any number of times, it forces all uses of algebraic effects to be synchronous, even when this is not necessary, e.g., when the effect involves executing a remote query to which a response is not needed (immediately).

Motivated by the recent interest in the combination of asynchrony and algebraic effects [Lei17, SDW+21], in this paper we explore what it takes to accompany the synchronous treatment of algebraic effects with an *asynchronous* one (in terms of language design, safe programming abstractions, and a self-contained core calculus). At the heart of our approach is the decoupling of the execution of algebraic operation calls into (i) *signalling* that some implementation of an operation needs to be executed, and (ii) *interrupting* a running computation with its result, to which the computation can react by (iii) *installing interrupt handlers*. Importantly, we show that our approach is flexible enough that not all signals need to have a corresponding interrupt, and vice versa, allowing us to also model *spontaneous behaviour*, such as a user clicking a button or the environment pre-empting a thread.

While we are not the first ones to work on asynchrony for algebraic effects, the prior work in this area (in the context of general effect handlers) has achieved it by simply *delegating* the actual asynchrony to the respective language backends [Lei17, SDW+21]. In contrast, in this paper we demonstrate how to capture the combination of asynchrony and algebraic effects in a *self-contained* core calculus. It is important to emphasise that our aim is not to replace general effect handlers, but instead to *complement* them with robust primitives tailored to asynchrony—as we highlight throughout the paper, our proposed approach is algebraic by design, so as to be ready for future extensions with general effect handlers.

Paper Structure. In Section 2, we give a high-level overview of our approach to asynchrony for algebraic effects. In Sections 3 and 4, we recap our previous work [AP21] on asynchronous algebraic effects using a core calculus, $\lambda_{æ}$, equipped with a small-step operational semantics and a type-and-effect system. In Section 5, we explore extensions of $\lambda_{æ}$ necessary to accommodate reinstallable interrupt handlers, higher-order signal and interrupt payloads, and the dynamic creation of processes, and prove their type safety. In Section 6, we show how these extensions can be used in examples such as pre-emptive multi-threading, remote function calls, and a parallel variant of runners of algebraic effects, simplifying the examples in our prior work [AP21]. We conclude by discussing related and future work in Section 7.

Code. The paper is accompanied by a *formalisation* of $\lambda_{æ}$'s type safety proofs in Agda [Ahm24], and a *prototype implementation* of $\lambda_{æ}$ in OCaml, called Æff [Pre24].

In Agda, we consider only the well-typed syntax of a variant of $\lambda_{æ}$ in which the subtyping rule manifests as an explicit coercion. Working with such well-typed syntax is a standard approach for making it easier to manage a de Bruijn indices-based representation of free and bound variables [WKS22]. Meanwhile, the Æff implementation provides an interpreter and a simple typechecker, but does not yet support inferring or checking effect annotations. Æff also provides a web interface that allows users to interactively click through their programs'

executions. It also comes with implementations of all the examples we present in this paper. Separately, Poulson [Pou20] has shown how to implement $\lambda_{æ}$ in Frank [CLMM20].

2. Asynchronous Effects, by Example

We begin with a high-level overview of how we model asynchrony within algebraic effects.

2.1. Conventional Algebraic Effects Are Synchronous by Nature.

We first recall the basic ideas of programming with algebraic effects, illustrating that their traditional treatment is synchronous by nature. For an in-depth overview, we refer the reader to a tutorial on effect handlers [Pre15], and to the seminal papers of the field [PP02, PP13].

In this algebraic treatment, sources of computational effects are modelled using signatures of *operation symbols* $\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}$. For instance, one models S-valued state using operations $\mathsf{get} : 1 \to S$ and $\mathsf{set} : S \to 1$, and E-valued exceptions using a single operation $\mathsf{raise} : E \to 0$.

Programmers can then invoke the effect that an operation $\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}$ models by placing an *operation call* $\mathsf{op}\ (V, y.M)$ in their code. Here, the parameter value V has type A_{op}, and the variable y, which is bound in the continuation M, has type B_{op}. For instance, for the set operation, the parameter value V would be the new value of the store, and for the get operation, the variable y would be bound to the current value of the store.

A program written in terms of operation calls is by itself just an inert piece of code. To execute it, programmers have to provide *implementations* for the operation calls appearing in it. The idea is that an implementation of $\mathsf{op}\ (V, y.M)$ takes V as its input, and its output gets bound to y. For instance, this could take the form of defining a suitable effect handler [PP13], but could also be given by calls to runners of algebraic effects [AB20], or simply by invoking some (default) top-level (native) implementation. What is important is that some pre-defined piece of code $M_{\mathsf{op}}[V/x]$ gets executed in place of every operation call $\mathsf{op}\ (V, y.M)$.

Now, what makes the conventional treatment of algebraic effects *synchronous* is that the execution of an operation call $\mathsf{op}\ (V, y.M)$ *blocks* until some implementation of op returns a value W to be bound to y, so that the execution of the continuation $M[W/y]$ could proceed [KLO13, BP14]. Conceptually, this kind of blocking behaviour can be illustrated as

$$\begin{array}{ccccc} & & M_{\mathsf{op}}[V/x] & \rightsquigarrow^* & \mathsf{return}\ W \\ & & \uparrow & & \downarrow \\ \cdots & \rightsquigarrow & \mathsf{op}\ (V, y.M) & & M[W/y] \ \rightsquigarrow\ \cdots \end{array} \tag{2.1}$$

where $\mathsf{return}\ W$ is a computation that causes no effects and simply returns the value W.

While blocking the execution of the rest of the computation is needed in the presence of general effect handlers that can execute their continuation any number of times, e.g., when simulating nondeterminism [PP13], it forces all uses of algebraic effects to be synchronous, even when this is not necessary, e.g., when the effect in question involves executing a remote query to which a response is not needed immediately, or sometimes never at all.

In the rest of this section, we describe how we decouple the invocation of an operation call from the act of receiving its result, and how we give programmers a means to block execution only when it is necessary. While we end up surrendering some of effect handlers' generality, such as having access to the continuation that captures the rest of the computation to be handled, then in return we get a natural and robust formalism for asynchronous programming.

2.2. **Outgoing Signals and Incoming Interrupts.** We begin by observing that the execution of an operation call $\mathsf{op}\ (V, y.M)$, as depicted in (2.1), consists of *three distinct phases*: (i) signalling that an implementation of op needs to be executed with parameter V (the up-arrow), (ii) executing this implementation (the horizontal arrow), and (iii) interrupting the blocked computation M with a value W (the down-arrow). In order to overcome the unwanted side-effects of blocking execution at every operation call, we decouple these phases into separate programming concepts, allowing M to proceed executing even if (ii) has not yet completed and (iii) taken place. In particular, we decouple an operation call into issuing an *outgoing signal*, written $\uparrow \mathsf{op}\,(V, M)$, and receiving an *incoming interrupt*, written $\downarrow \mathsf{op}\,(W, M)$.

It is important to note that while we have used the execution of operation calls to motivate the introduction of signals and interrupts as programming concepts, *not all issued signals need to have a corresponding interrupt response*, and *not all interrupts need to be responses to issued signals*, allowing us to also model spontaneous behaviour, such as a user clicking a button or the environment pre-empting a thread.

When *issuing a signal* $\uparrow \mathsf{op}\,(V, M)$, the value V is called a *payload*, such as a location to be looked up or a message to be displayed, aimed at whoever is listening for the given signal. We use the $\uparrow$-notation to indicate that signals issued in sub-computations propagate outwards—in this sense signals behave just like conventional algebraic operation calls.

Since no additional variables are bound in the continuation M, it is naturally possible to continue executing M straight after the signal has been issued, as depicted below:

$$\cdots \rightsquigarrow \overset{\mathsf{op}\,V\uparrow}{\uparrow \mathsf{op}\,(V, M)} \rightsquigarrow M \rightsquigarrow \cdots$$

This crucially differs from the usual treatment of algebraic effects, which though being able to simulate our approach [Pou20], find asynchronous evaluation of continuations undesirable. For example, even if in the (conventional) operation call $\mathsf{op}\ (V, y.M)$ the continuation M does not depend on y, M can cause further effects, leading to unexpected behaviour if M performs those effects before or after the handler for op is evaluated.

As a *running example*, let us consider a computation $M_{\text{feedClient}}$, which lets a user scroll through a seemingly infinite feed of data, e.g., by repeatedly clicking a "next page" button. For efficiency, $M_{\text{feedClient}}$ does not initially cache all the data available on a server, but instead requests a new batch of data each time scrolling through the data is nearing the end of the cache. To communicate with the outside world, $M_{\text{feedClient}}$ can issue a signal

$$\uparrow \mathsf{request}\,(\mathit{offset}, M_{\text{feedClient}})$$

to request a new batch of data starting from the given *offset*, or a different signal

$$\uparrow \mathsf{display}\,(\mathit{message}, M_{\text{feedClient}})$$

to display a string *message* to the user. In both cases, the continuation *does not wait* for an acknowledgement that the signal was received, but instead continues to provide a seamless experience to the user. It is however worth noting that these signals differ in what $M_{\text{feedClient}}$ expects of them: to the $\mathsf{request}$ signal, it expects a response at some future point in its execution, while it does not expect any response to the $\mathsf{display}$ signal, illustrating that not every issued signal needs an immediate response, and that some do not need one at all.

When the outside world wants to get the attention of a computation, be it in response to a signal or spontaneously, it happens by *propagating an interrupt* $\downarrow \mathsf{op}\,(W, M)$ to the computation. Here, the value W is again called a *payload*, while M is the computation

receiving the interrupt. It is important to note that unlike signals, interrupts are not triggered by the computation itself, but are instead issued by the *outside world*, and can thus interrupt any sequence of evaluation steps, e.g., as depicted in

$$\cdots \rightsquigarrow M \overset{\downarrow \mathsf{op}\, W}{\rightsquigarrow} \downarrow \mathsf{op}\,(W, M) \rightsquigarrow \cdots$$

In our running example, there are two interrupts of interest that $M_{\text{feedClient}}$ might receive:

$$\downarrow \mathsf{response}\,(\mathit{newBatch}, M)$$

which delivers a batch of new data to replenish $M_{\text{feedClient}}$'s cache, and

$$\downarrow \mathsf{nextItem}\,((), M)$$

with which the user requests to see the next data item. In both cases, the continuation M represents the state of $M_{\text{feedClient}}$ at the time of receiving the interrupt.

We use the $\downarrow$-notation to indicate that interrupts propagate inwards into subcomputations, trying to reach anyone listening for them, and only get discarded when they reach a return. Programmers are not expected to write interrupts explicitly in their programs—instead, interrupts are usually induced by signals issued by other parallel processes, as explained next.

2.3. **A Signal for the Sender Is an Interrupt to the Receiver.** As noted above, the computations we consider do not evolve in isolation, instead they also communicate with the outside world, by issuing outgoing signals and receiving incoming interrupts.

We model the outside world by composing individual computations into *parallel processes* $P, Q, \ldots$. To keep the presentation clean and focussed on the asynchrony of algebraic effects, we consider a very simple model of parallelism: a process is either one of the computations being run in parallel, written $\mathsf{run}\ M$, or the parallel composition of two processes, written $P \,||\, Q$. Later, in Section 5.4, we show how to also accommodate dynamic process creation.

To capture the signals and interrupts based interaction of processes, our operational semantics includes rules for *propagating outgoing signals* from individual computations to processes, *turning processes' outgoing signals into incoming interrupts* for their surrounding world, and *propagating incoming interrupts* from processes to individual computations. For instance, in our running example, $M_{\text{feedClient}}$'s request for new data is executed as follows:

$$\begin{aligned}
& \mathsf{run}\,(\uparrow \mathsf{request}\,(V, M_{\text{feedClient}})) \,||\, \mathsf{run}\ M_{\text{feedServer}} \\
\rightsquigarrow\ & (\uparrow \mathsf{request}\,(V, \mathsf{run}\ M_{\text{feedClient}})) \,||\, \mathsf{run}\ M_{\text{feedServer}} \\
\rightsquigarrow\ & \uparrow \mathsf{request}\,\big(V, \mathsf{run}\ M_{\text{feedClient}} \,||\, \downarrow \mathsf{request}\,(V, \mathsf{run}\ M_{\text{feedServer}})\big) \\
\rightsquigarrow\ & \uparrow \mathsf{request}\,\big(V, \mathsf{run}\ M_{\text{feedClient}} \,||\, \mathsf{run}\,(\downarrow \mathsf{request}\,(V, M_{\text{feedServer}}))\big)
\end{aligned}$$

Here, the first and the last reduction step respectively propagate signals outwards and interrupts inwards. The middle reduction step corresponds to what we call a *broadcast rule*—it turns an outward moving signal in one of the processes into an inward moving interrupt for the process parallel to it, while continuing to propagate the signal outwards to any further parallel processes. The active redexes in these rules are highlighted in grey.

2.4. **Promising To Handle Interrupts.** So far, we have shown that our computations can issue outgoing signals and receive incoming interrupts, and how these evolve and get communicated when executing parallel processes, but we have not yet said anything about how computations can actually *react* to incoming interrupts of interest.

In order to react to interrupts, our computations can install *interrupt handlers*, written

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

that should be read as: "we promise to handle a future incoming interrupt named op using the computation M in the continuation N, with x bound to the payload of the interrupt". Fulfilling this promise consists of executing M and binding its result to the promise variable p in N when a corresponding interrupt arrives, as captured by the following reduction rule:

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M[V/x]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$

Interrupts that do not match a given interrupt handler ($\mathsf{op} \neq \mathsf{op}'$) simply move past it:

$$\downarrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}'\,(V, N)$$

It is worth noting that the interrupt itself *keeps propagating inwards* into the sub-computation N, where it can trigger further interrupt handlers installed for the given interrupt. Allowing the interrupts to always keep propagating inwards is a natural design choice, as it connects the behaviour of our interrupts with the behaviour of deep effect handling [PP13] (see Section 3.2), and it is crucial for certain examples (see Section 6.5).

In order to skip certain interrupt handlers for some op, one can carry additional data in op's payload (e.g., a thread ID) and then condition the (non-)triggering of those interrupt handlers on this data, e.g., as we demonstrate in Section 6.1. This is analogous to how one controls which particular operation calls are handled with ordinary effect handlers [KLO13].

Interrupt handlers differ from conventional algebraic operation calls (see Section 2.1) in two important aspects. First, they enable *user-side post-processing* of received data, using M, while in operation calls the result is immediately bound in the continuation. Second, and more importantly, their semantics is *non-blocking*. In particular, we have a congruence rule

$$N \rightsquigarrow N' \qquad \text{implies} \qquad \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N'$$

meaning that the continuation N, and thus the whole computation, can make progress even though no interrupt op has been propagated to the computation from the outside world.

As the observant reader might have noticed, the non-blocking behaviour of interrupt handling means that our operational semantics has to work on *open terms* because the variable p can appear free in both N and N' in the congruence rule given above. However, it is important to note that p is not an arbitrarily typed variable, but in fact gets assigned a distinguished *promise type* $\langle X \rangle$ for some value type X—we shall crucially make use of this typing of p in the proof of type safety for our $\lambda_{æ}$-calculus (see Theorem 3.2 and 5.1). Furthermore, since it is the computation M that fulfils the promise (either by supplying a value or returning another promise), it also needs to have the same return type $\langle X \rangle$.

2.5. **Blocking on Interrupts Only When Necessary.** As noted earlier, installing an interrupt handler means making a promise to handle a given interrupt in the future. To check that an interrupt has been received and handled, we provide programmers a means to selectively *block execution* and *await* a specific promise to be fulfilled, written $\mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M$,

where if V has a promise type $\langle X \rangle$, the variable x bound in M has type X. Importantly, the continuation M is executed only when the await is handed a *fulfilled promise* $\langle V \rangle$, as

$$\mathsf{await}\ \langle V \rangle\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M \rightsquigarrow M[V/x]$$

In our example of scrolling through a seemingly infinite feed, $M_{\text{feedClient}}$ could use await to block until it has received an initial configuration, such as the batch size used by $M_{\text{feedServer}}$.

As the terminology suggests, this part of $\lambda_{æ}$ is strongly influenced by existing work on *futures and promises* [Sch02] for structuring concurrent programs, and their use in modern languages, such as in Scala [HPM+20]. While prior work often models promises as writeable, single-assignment references, we instead use the substitution of values for ordinary immutable variables (of distinguished promise type) to model that a promise gets fulfilled exactly once. This way we achieve the standard reading of promises without needing a stateful operational semantics and a non-trivial type system to enforce the single-assignment behaviour [AFH+18].

2.6. **Reinstalling Interrupt Handlers.** As seen in the reduction rule

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M[V/x]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$

the interrupt handler is *not reinstalled by default*. The programmers can selectively reinstall interrupt handlers using general recursion [AP21], or use the extension of $\lambda_{æ}$ with *reinstallable interrupt handlers* we propose in this paper (see Section 5.1 for details), which have the form

$$\mathsf{promise}\ (\mathsf{op}\ x\, r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

These behave similarly to ordinary interrupt handlers, except that the handling computation M has access to an additional variable r bound to a function that reinstalls the handler when called. Specifically, triggering a reinstallable interrupt handler has the following form:

$$\begin{aligned}&\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x\, r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)\\ &\quad\rightsquigarrow \mathsf{let}\ p = M\big[V/x, \big(\mathsf{fun}\ \_ \mapsto \mathsf{promise}\ (\mathsf{op}\ x\, r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p\big)/r\big]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)\end{aligned}$$

Further, in examples we often find it useful to also pass data between subsequent reinstalls of an interrupt handler. Programmers can achieve this by working with an additionally assumed primitive notion of memory references [AP21], or by using a *stateful variant of reinstallable interrupt handlers* that we propose in this paper. The latter have the form

$$\mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in}\ N$$

where S is the type of state associated with a particular interrupt handler, W is the interrupt handler's state at the time of its next triggering, the variable s gives the interrupt handler code M access to the state, and the state can be updated by reinstalling the handler with an updated value using r. Specifically, the interrupt handler triggering rule now has the form

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\, W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M\big[V/x, R/r, W/s\big]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$

where R denotes a function that reinstalls the interrupt handler with an updated state value:

$$R \stackrel{\text{def}}{=} \mathsf{fun}\ (s' : S) \mapsto \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ s'\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

For brevity, we often omit the S-annotation in examples when it is clear from the context.

2.7. **Putting It All Together.** We conclude this overview by showing how to implement the example of a user scrolling through a seemingly infinite feed of data in our $\lambda_{æ}$-calculus.

For a simpler exposition, we allow ourselves access to mutable references, with which we communicate data between different interrupt handlers, though the same can be achieved by rolling one's own state. For passing data between subsequent reinstalls of the same interrupt handler, we use the state-passing features of interrupt handlers introduced above.

While having explicit continuations in operation calls, signals, interrupt handlers, and when awaiting promises to be fulfilled makes the meta-theory of the underlying calculus cleaner (see Section 3.2), in programming we prefer to use *generic* versions of them, i.e., ones with trivial continuations [PP03]. In particular, we define and use the syntactic sugar:

$$\uparrow \mathsf{op}\, V \stackrel{\text{def}}{=} \uparrow \mathsf{op}\,(V, \mathsf{return}\,())$$

$$\mathsf{promise}\,(\mathsf{op}\, x\, r \mapsto M) \stackrel{\text{def}}{=} \mathsf{promise}\,(\mathsf{op}\, x\, r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

$$\mathsf{promise}\,(\mathsf{op}\, x\, r\, s \mapsto M)\ @\ W \stackrel{\text{def}}{=} \mathsf{promise}\,(\mathsf{op}\, x\, r\, s \mapsto M)\ @\ W\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

$$\mathsf{await}\ V \stackrel{\text{def}}{=} \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ \mathsf{return}\ x$$

2.7.1. *Client.* We implement the client computation $M_{\mathsf{feedClient}}$ as the function feedClient defined below. For presentation purposes, we split its definition between multiple code blocks.

First, the client initialises some auxiliary references, issues a signal to the server to ask for the data batch size that it uses, and then installs a corresponding interrupt handler:

```
let feedClient () =
  let cachedData = ref [] in
  let requestInProgress = ref false in
  ↑ batchSizeRequest ();
  let batchSizePromise = promise (batchSizeResponse batchSize ↦ return ⟨batchSize⟩) in
  ...
```

While the server is asynchronously responding to the batch size request, the client sets up an auxiliary function requestNewData, with which it can request new data from the server:

```
  ...
  let requestNewData offset =
    requestInProgress := true;
    ↑ request offset;
    promise (response newBatch ↦
      cachedData := !cachedData @ newBatch;
      requestInProgress := false;
      return ⟨()⟩
    )
  in
  ...
```

Here, we first set a flag indicating that a new data request is in process, then issue a request signal to the server, and finally install an interrupt handler that updates the cache once a corresponding response interrupt arrives. We note that the client computation does not block

while awaiting new data from the server—instead, it continues executing, notifying the user to wait and try again once the cache temporarily becomes empty (see below).

As a last step of setting itself up, the client blocks until the server has responded with the batch size it uses by awaiting batchSizePromise to be fulfilled, after which the client starts its main loop, which we implement as a simple reinstallable interrupt handler:

```
...
let batchSize = await batchSizePromise in
promise (nextItem _ r currentItem ↦
  let cachedSize = length !cachedData in
  (if (currentItem > cachedSize − batchSize / 2) && (not !requestInProgress) then
     requestNewData (cachedSize + 1)
   else
     return ());
  if currentItem < cachedSize then
    ↑ display (toString (nth !cachedData currentItem));
    r (currentItem + 1)
  else
    ↑ display "please wait a bit and try again";
    r currentItem
) @ 0
```

In it, the client listens for nextItem interrupts from the user to display more data. Once the interrupt arrives, the client checks if its cache is becoming empty, i.e., if the index of the currently viewed item is less than half of the batch size away from the last cached item and if no request for new data has been issued yet. If that happens, the client uses the requestNewData function to request more data from the server, starting with offset cachedSize + 1, which is the index of the first item that is outside of the data cached by the client.

Next, if there is still some data in the cache, the client issues a display signal to show the next data item to the user. If however the cache is empty, the client issues a display signal to show a message to the user asking them to wait and try again. The client then simply reinvokes itself by reinstalling the interrupt handler for nextItem interrupts (by calling r).

Observe that the currentItem counter is initially set to 0 and then passed between subsequent interrupt handler reinstalls using the state-passing features introduced earlier.

2.7.2. *Server.* We implement the server computation $M_{\text{feedServer}}$ as the following function:

```
let feedServer batchSize =
  promise (batchSizeRequest () r ↦
    ↑ batchSizeResponse batchSize;
    r ()
  );
  promise (request offset r ↦
    let payload = map (fun x ↦ 10 ∗ x) (range offset (offset + batchSize − 1)) in
    ↑ response payload;
    r ()
  )
```

where the computation range i j returns a list of integers ranging from i to j (both inclusive).

The server simply installs two reinstallable interrupt handlers: the first one listens for and responds to client's requests about the batch size it uses; and the second one responds to client's requests for new data. Both interrupt handlers then simply reinstall themselves.

The two interrupt handlers share a common pattern of handling the interrupt by issuing a signal and then immediately reinstalling the handler, and it is tempting to avoid the repetition. A dual shared pattern can be found in Section 2.7.1, where issuing a request signal is immediately followed by installing an interrupt handler for its response. However, proper user-defined abstractions capturing these patterns would require operation names to be first-class values, which is not only orthogonal to the issue of asynchrony we are focussing on, but leads to a dependently typed calculus in combination with an effect system.

2.7.3. *User.* We can also simulate the user as a computation. For the sake of simplicity, we allow ourselves general recursion to implement the user behaviour as an infinite loop that every now and then issues a request to the client to display the next data item.

```
let rec user () =
  let rec wait n =
    if n = 0 then return () else wait (n − 1)
  in
  ↑ nextItem (); wait 10; user ()
```

Alternatively, without assuming general recursion, we could have implemented the user instead as two parallel processes that indefinitely ping-pong each other, and occasionally issue nextItem signals to the client (see Section 7 for an example of such non-terminating behaviour). It is also straightforward to extend the user program with a reinstallable handler for display interrupts that simulates displaying the data items received from the client (omitted here).

2.7.4. *Running the Server, Client, and User in Parallel.* Finally, we can simulate our running example in full by running all three computations as parallel processes, as follows:

```
run (feedServer 42) || run (feedClient ()) || run (user ())
```

3. A Calculus for Asynchronous Effects: Values and Computations

Before we focus on extensions necessary for higher-order asynchronous effects in Section 5, we first recap $\lambda_{æ}$, our existing core calculus for programming with first-order asynchronous effects [AP21]. The version we present here differs from the original one in two aspects: we drop the reliance on general recursion, as reinstallable interrupt handlers that we introduce in Section 5.1 are sufficient to express all the existing examples, and we slightly modify the behaviour of the await construct in order to make the meta-theory slightly simpler.

To better explain the different features of the calculus and its semantics, we split the recap of $\lambda_{æ}$ into a *sequential* part (discussed below) and a *parallel* part (discussed in Section 4).

Values

V, W	$::= x$	variable
	$\mid () \mid (V, W)$	unit and pairing
	$\mid \mathsf{inl}_Y\, V \mid \mathsf{inr}_X\, V$	left and right injections
	$\mid \mathsf{fun}\, (x : X) \mapsto M$	function abstraction
	$\mid \langle V \rangle$	fulfilled promise

Computations

M, N	$::= \mathsf{return}\, V$	returning a value
	$\mid \mathsf{let}\, x = M\, \mathsf{in}\, N$	sequencing
	$\mid V\, W$	function application
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{(x, y) \mapsto M\}$	product elimination
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{\}_{Z!(o,\iota)}$	empty elimination
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{\mathsf{inl}\, x \mapsto M, \mathsf{inr}\, y \mapsto N\}$	sum elimination
	$\mid \uparrow \mathsf{op}\, (V, M)$	outgoing signal
	$\mid \downarrow \mathsf{op}\, (V, M)$	incoming interrupt
	$\mid \mathsf{promise}\, (\mathsf{op}\, x \mapsto M)\, \mathsf{as}\, p\, \mathsf{in}\, N$	interrupt handler
	$\mid \mathsf{await}\, V\, \mathsf{until}\, \langle x \rangle\, \mathsf{in}\, M$	awaiting a promise to be fulfilled

Figure 1: Values and Computations.

3.1. **Values and Computations.** We base $\lambda_{æ}$ on the fine-grain call-by-value λ-calculus (FGCBV) [LPT03], and as such, it is a low-level intermediate language to which a corresponding high-level user-facing programming language could be compiled to—this is what happens in our prototype implementation [Pre24].

The syntax of terms is given in Figure 1, stratified into *values* and *computations*, as in FGCBV. While we do not study effect inference in this paper, we equip certain terms with type annotations that in our experience should make it possible to fully infer types.

Values. The values $V, W, \ldots$ are mostly standard. They include variables, introduction forms for sums and products, and functions. The only $\lambda_{æ}$-specific value is $\langle V \rangle$, which denotes a *fulfilled promise*, indicating that the promise of handling some interrupt has been fulfilled with the value V.

Computations. The computations $M, N, \ldots$ also include all standard terms from FGCBV: returning values, sequencing, function application, and elimination forms.

The first two $\lambda_{æ}$-specific computations are *signals* $\uparrow \mathsf{op}\, (V, M)$ and *interrupts* $\downarrow \mathsf{op}\, (V, M)$, where op is drawn from a set Σ of names, V is a data payload, and M is a continuation.

The next $\lambda_{æ}$-specific computation is the *interrupt handler* $\mathsf{promise}\, (\mathsf{op}\, x \mapsto M)\, \mathsf{as}\, p\, \mathsf{in}\, N$, where x is bound in M and p in N. As discussed in the previous section, one should understand this computation as making a promise to handle a future incoming interrupt op by executing the computation M. Sub-computations of the continuation N can then explicitly await, when necessary, for this promise to be fulfilled by blocking on the *promise-typed variable* p

using the final $\lambda_{æ}$-specific computation term, the *awaiting* construct await V until $\langle x \rangle$ in M. It is useful to note that the p used above is an ordinary variable—it just gets assigned the distinguished promise type $\langle X \rangle$ by the interrupt handler (as discussed in Section 3.3).

3.2. **Small-Step Operational Semantics.** We equip $\lambda_{æ}$ with an evaluation contexts based small-step operational semantics, defined using a reduction relation $M \rightsquigarrow N$. The *reduction rules* and *evaluation contexts* are given in Figure 2. We discuss the rules in detail below. Note that since we have chosen to equip effectful constructs with explicit continuations, the evaluation contexts are used to compress four congruence rules into a single one. If instead we took generic versions (like seen in Section 2.7) as primitives, almost all the rules in Figure 2, apart from the ones for standard monadic computations, would need to be phrased in terms of sequential composition (i.e., let), leading to a notably less clear presentation.

Computation Rules. The first group includes *standard reduction rules* from FGCBV, such as β-reducing function applications, sequential composition, and the standard elimination forms. These rules involve standard *capture avoiding substitutions* $V[W/x]$ and $M[W/x]$, defined by straightforward mutual structural recursion on V and M.

Algebraicity. This group of reduction rules *propagates outwards* the signals that have been issued, interrupt handlers that have been installed, and computations awaiting fulfilled promises. While it is not surprising that outgoing signals behave like algebraic *operation calls*, getting propagated outwards as far as possible, then it is much more curious that the natural operational behaviour of interrupt handlers turns out to be the same. As we shall explain in Section 7, despite using the (operating systems inspired) "handler" terminology, mathematically interrupt handlers are in fact a form of scoped algebraic operations [PSWJ18].

In contrast to our original calculus [AP21], the awaiting construct also propagates outwards. Before, awaiting a promise in any subcomputation would block the evaluation immediately, whereas now, we can do the additional outwards propagation steps. Importantly, this does not significantly change the computational behaviour, as after the propagation, the evaluation still blocks as long as the promise is left unfulfilled. The main difference and benefit is that all computations awaiting for a promise variable p now show this explicitly at their top-level, as they are of the form await p until $\langle x \rangle$ in M. This change significantly simplifies the normal forms of computations (see Section 3.4) and the resulting meta-theory.

In the last two algebraicity rules, and other similar ones, we assume Barendregt's variable convention to avoid accidentally capturing free variables when extending the scope of binders.

Commutativity of Signals With Interrupt Handlers. This rule complements the algebraicity rule for signals, by further propagating them outwards, past enveloping interrupt handlers. From the perspective of algebraic effects, this rule is an example of two algebraic operations *commuting* [HPP06]. Since in this rule, the scope of p contracts, the usual variable naming precautions are not sufficient for type safety. Instead, the type system ensures (see Section 3.3) that the (promise-typed) variable p cannot appear in the payload value V.

Standard computation rules

$$(\mathsf{fun}\ (x : X) \mapsto M)\, V \rightsquigarrow M[V/x]$$
$$\mathsf{let}\ x = (\mathsf{return}\ V)\ \mathsf{in}\ N \rightsquigarrow N[V/x]$$
$$\mathsf{match}\ (V, W)\ \mathsf{with}\ \{(x, y) \mapsto M\} \rightsquigarrow M[V/x, W/y]$$
$$\mathsf{match}\ (\mathsf{inl}_Y\ V)\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} \rightsquigarrow M[V/x]$$
$$\mathsf{match}\ (\mathsf{inr}_X\ W)\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} \rightsquigarrow N[W/y]$$

Algebraicity of signals, interrupt handlers, and awaiting

$$\mathsf{let}\ x = (\uparrow \mathsf{op}\,(V, M))\ \mathsf{in}\ N \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{let}\ x = M\ \mathsf{in}\ N)$$
$$\mathsf{let}\ x = (\mathsf{promise}\ (\mathsf{op}\ y \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N_1)\ \mathsf{in}\ N_2 \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ y \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ (\mathsf{let}\ x = N_1\ \mathsf{in}\ N_2)$$
$$\mathsf{let}\ x = (\mathsf{await}\ V\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ M)\ \mathsf{in}\ N \rightsquigarrow \mathsf{await}\ V\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ (\mathsf{let}\ x = M\ \mathsf{in}\ N)$$

Commutativity of signals with interrupt handlers

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\,(V, N) \rightsquigarrow \uparrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$$

Interrupt propagation

$$\downarrow \mathsf{op}\,(V, \mathsf{return}\ W) \rightsquigarrow \mathsf{return}\ W$$
$$\downarrow \mathsf{op}\,(V, \uparrow \mathsf{op}'\,(W, M)) \rightsquigarrow \uparrow \mathsf{op}'\,(W, \downarrow \mathsf{op}\,(V, M))$$
$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M[V/x]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$
$$\downarrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}'\,(V, N) \quad (\mathsf{op} \neq \mathsf{op}')$$
$$\downarrow \mathsf{op}\,(V, \mathsf{await}\ W\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M) \rightsquigarrow \mathsf{await}\ W\ \mathsf{until}\ \langle x \rangle\ \mathsf{in} \downarrow \mathsf{op}\,(V, M)$$

Awaiting a promise to be fulfilled

$$\mathsf{await}\ \langle V \rangle\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M \rightsquigarrow M[V/x]$$

Evaluation context rule

$$\frac{M \rightsquigarrow N}{\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]}$$

where

$$\mathcal{E} ::= [\,] \mid \mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ N \mid \uparrow \mathsf{op}\,(V, \mathcal{E}) \mid \downarrow \mathsf{op}\,(V, \mathcal{E}) \mid \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathcal{E}$$

Figure 2: Small-step Operational Semantics of Computations.

Interrupt Propagation. The handler-operation curiosity does not end with interrupt handlers. This group of reduction rules describes how interrupts are *propagated inwards* into sub-computations. While $\downarrow \mathsf{op}\,(V, M)$ might look like a conventional operation call, then its operational behaviour instead mirrors that of *deep effect handling* [PP13], where one also recursively descends into the computation being handled.

When designing interrupt propagation, we must ensure that each interrupt handler receives a corresponding interrupt, no matter how deep inside the computation we install it. The first reduction rule states that we can safely discard an interrupt when it reaches a trivial, effect-free computation $\mathsf{return}\ W$. The second rule states that we can propagate incoming interrupts past any outward moving signals. The next two rules describe how interrupts interact with interrupt handlers, in particular, that the former behave like effect handling

(when understanding interrupt handlers as generalised algebraic operations). On the one hand, if the interrupt matches the interrupt handler it encounters, the corresponding handler code M is executed, and the interrupt is propagated inwards into the continuation N. On the other hand, if the interrupt does not match the interrupt handler, it is simply propagated past the interrupt handler into N. Finally, to simplify normal forms, we propagate interrupts inside computations awaiting fulfilled promises as well. As with the algebraicity rule, this lets the computation take a single additional step after which the await construct reaches the top and blocks the evaluation.

We have given the interrupt propagation rules only for terms that are in normal form (see Lemma 3.1). For example, we do not push interrupts into the branches of sum elimination. Instead, for terms that are still reducing, interrupts remain as parts of their evaluation contexts and wait for inner interrupt handlers to propagate outwards and meet them.

An alternative design choice for interrupt propagation would be to take inspiration from *shallow interrupt handling* [KLO13], and instead of always propagating the interrupts inwards into the continuations of interrupt handlers, the programmers themselves would have to manually (recursively) reinvoke the interrupts that need to be propagated inwards. In addition to giving an algebraically more natural semantics (due to the relationship with deep effect handling), our choice of allowing interrupts to always propagate inwards provides a more predictable programming model, in which an installed interrupt handler is guaranteed to be executed whenever a corresponding interrupt is received, no matter what other installed interrupt handlers may do on the way. We leave exploring a variant of $\lambda_{æ}$ based on shallow effect handling, and its formal relationship to this paper, for future work.

Awaiting a Promise To Be Fulfilled. In addition to the two rules for outwards propagation, the semantics of the await construct includes a β-rule allowing the blocked computation M to resume executing as $M[V/x]$ when the await in question is given a fulfilled promise $\langle V \rangle$.

Evaluation Contexts. The semantics allows reductions under *evaluation contexts* $\mathcal{E}$. Observe that as discussed earlier, the inclusion of interrupt handlers in the evaluation contexts means that reductions involve potentially open terms. Also, differently from the semantics of conventional operation calls [KLO13, BP14], our evaluation contexts include outgoing signals. As such, the *evaluation context rule* allows the execution of a computation to proceed even if a signal has not yet been propagated to its receiver, or when an interrupt has not yet arrived. Importantly, the evaluation contexts do not include await, so as to model its blocking behaviour. We write $\mathcal{E}[M]$ for the operation of filling the hole $[\,]$ in $\mathcal{E}$ with M.

Non-Confluence. It is worth noting that the asynchronous design means that the operational semantics is *nondeterministic*. More interestingly, the semantics is also *not confluent*.

For one source of non-confluence, let us consider two reduction sequences of a same computation, where for better readability, we highlight the active redex for each step:

$$\begin{aligned}
&\downarrow \mathsf{op}\,(V, \mathsf{promise}\,(\mathsf{op}\,x \mapsto (\mathsf{promise}\,(\mathsf{op}'\,y \mapsto M)\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ M'))\ \mathsf{as}\ p\ \mathsf{in}\ N) \\
\rightsquigarrow\ &\downarrow \mathsf{op}\,(V, \mathsf{promise}\,(\mathsf{op}\,x \mapsto (\mathsf{promise}\,(\mathsf{op}'\,y \mapsto M)\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ M'))\ \mathsf{as}\ p\ \mathsf{in}\ N') \\
\rightsquigarrow\ &\mathsf{let}\ p = (\mathsf{promise}\,(\mathsf{op}'\,y \mapsto M[V/x])\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ M')\ \mathsf{in} \downarrow \mathsf{op}\,(V, N') \\
\rightsquigarrow\ &\mathsf{promise}\,(\mathsf{op}'\,y \mapsto M[V/x])\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ (\mathsf{let}\ p = M'\ \mathsf{in} \downarrow \mathsf{op}\,(V, N'))
\end{aligned}$$

and

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto (\mathsf{promise}\ (\mathsf{op}'\ y \mapsto M)\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ M'))\ \mathsf{as}\ p\ \mathsf{in}\ N)$$
$$\rightsquigarrow \mathsf{let}\ p = (\mathsf{promise}\ (\mathsf{op}'\ y \mapsto M[V/x])\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ M')\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$
$$\rightsquigarrow \mathsf{promise}\ (\mathsf{op}'\ y \mapsto M[V/x])\ \mathsf{as}\ q\ \mathsf{in}\ \mathsf{await}\ q\ \mathsf{until}\ \langle z \rangle\ \mathsf{in}\ (\mathsf{let}\ p = M'\ \mathsf{in} \downarrow \mathsf{op}\,(V, N))$$

Here, both final computations are *temporarily* blocked until an incoming interrupt op' is propagated to them and the variable q gets bound to a fulfilled promise. Until this happens, it is not possible for the blocked continuation N to reduce to N' in the latter final computation.

Another, distinct source of non-confluence concerns the commutativity of outgoing signals with enveloping interrupt handlers. For instance, the following composite computation

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto \uparrow \mathsf{op}'\,(W', M))\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}''\,(W'', N))$$

can nondeterministically reduce to either

$$\uparrow \mathsf{op}'\,(W', \uparrow \mathsf{op}''\,(W'', \mathsf{let}\ p = M\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)))$$

if we first propagate the interrupt op inwards, or to

$$\uparrow \mathsf{op}''\,(W'', \uparrow \mathsf{op}'\,(W', \mathsf{let}\ p = M\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)))$$

if we first propagate the signal op'' outwards. As a result, in the resulting two computations, the signals op' and op'' get issued, and received by other processes, in a different order.

A More Efficient Operational Semantics? Finally, it is worth emphasising that the operational semantics we present in this paper is meant to serve as a declarative reference semantics of $\lambda_{æ}$, and as a means to relate the behaviour of the program constructs specific to $\lambda_{æ}$ to the behaviour of conventional algebraic effects and their handlers. As such, the semantics is clearly not as efficient as one might desire in a real-world implementation. For instance, in the current semantics, signals are propagated out of computations one small step at a time. Instead, one might consider an alternative semantics in which there would be a reduction rule to pull signals out of computations from arbitrary depths. Dually, the propagation of interrupts into computations also happens one small step at a time. Here one might wonder whether it could be possible to use substitution in $\lambda_{æ}$ to make that propagation more efficient, akin to how we currently use substitution to propagate fulfilled promises to sub-computations. Yet another approach could be to model signal and interrupt propagation using shared channels, as noted in Section 7. However, as in this paper our focus is not on the efficiency of the semantics, we leave all such explorations for future work.

3.3. **Type-and-Effect System.** We equip $\lambda_{æ}$ with a type system in the tradition of type-and-effect systems for algebraic effects and effect handlers [BP14, KLO13], by extending the simple type system of FGCBV with annotations about programs' possible effects (such as issued signals and installed interrupt handlers) in function and computation types.

$$\begin{array}{rl}
\text{Ground type } A, B ::= & \mathsf{b} \mid 1 \mid 0 \mid A \times B \mid A + B \\
\text{Signal or interrupt signature:} & \mathsf{op} : A_{\mathsf{op}} \\
\text{Outgoing signal annotations:} & o \in O \\
\text{Interrupt handler annotations:} & \iota \in I \\
\text{Value type } X, Y ::= & A \mid X \times Y \mid X + Y \mid X \to Y \,!\, (o, \iota) \mid \langle X \rangle \\
\text{Computation type:} & X \,!\, (o, \iota) \\
\text{Typing context } \Gamma ::= & \cdot \mid \Gamma, x : X
\end{array}$$

Figure 3: Value and Computation Types.

3.3.1. *Types.* We define types in Figure 3, separated into ground, value, and computation types.

As noted in Section 3.1, $\lambda_{æ}$ is parameterised over a set Σ of signal and interrupt *names*. To each such name $\mathsf{op} \in \Sigma$, we assign a *signature* $\mathsf{op} : A_{\mathsf{op}}$ that specifies the payload type A_{op} of the corresponding signal or interrupt. Crucially, in order to be able to later prove that $\lambda_{æ}$ is type-safe, we must put restrictions on these signatures, as they classify values that may cross interrupt handler or process boundaries. In Section 5.3, we describe the exact reasons behind this restriction, and propose a more flexible type system employing Fitch-style modal types [Clo18]. But for the sake of exposition, we use here the more limited approach from our original work [AP21], and restrict payload types to *ground types* $A, B, \ldots$, which include base, unit, empty, product, and sum types, but importantly exclude promise and function types.

Value types $X, Y, \ldots$ extend ground types with function and promise types. The *function type* $X \to Y \,!\, (o, \iota)$ classifies functions that take X-typed arguments to computations classified by the *computation type* $Y \,!\, (o, \iota)$, i.e., ones that return Y-typed values, while possibly issuing signals specified by o and handling interrupts specified by ι. The *effect annotations* o and ι are drawn from sets O and I whose definitions we discuss in Section 3.3.2. The $\lambda_{æ}$-specific *promise type* $\langle X \rangle$ classifies promises that can be fulfilled by supplying a value of type X.

3.3.2. *Effect Annotations.* We now explain how we define the sets O and I from which we draw the effect annotations we use for specifying functions and computations. Traditionally, effect systems for algebraic effects simply use (flat) sets of operation names for effect annotations [BP14, KLO13]. In $\lambda_{æ}$, however, we need to be more careful, because triggering an interrupt handler executes a computation that can issue potentially different signals and handle different interrupts from the main program, and we would like to capture this in types.

Signal Annotations. First, as outgoing signals do not carry any computational data, we follow the tradition of type-and-effect systems for algebraic effects, and define O to be the *power set* $\mathcal{P}(\Sigma)$. As such, each $o \in O$ is a subset of the signature Σ, specifying which signals a computation might issue (this is an over-approximation of the actually issued signals).

Interrupt Handler Annotations. As observed above, for specifying installed interrupt handlers, we cannot use (flat) sets of interrupt names as the effect annotations $\iota \in I$ if we want to track the nested (and sometimes recursive) effectful structure of interrupt handlers.

Instead, intuitively each $\iota \in I$ is a *possibly infinite* nesting of partial mappings of pairs of O- and I-annotations to names in Σ—these pairs of annotations classify the possible effects of the corresponding interrupt handler code. We use the record notation

$$\iota = \{\mathsf{op}_1 \mapsto (o_1, \iota_1), \ldots, \mathsf{op}_n \mapsto (o_n, \iota_n)\}$$

to mean that ι maps $\mathsf{op}_1, \ldots, \mathsf{op}_n$ to the annotations $(o_1, \iota_1), \ldots, (o_n, \iota_n)$, while any other names in Σ are unannotated, corresponding to no interrupt handlers being installed for these other names. We write $\iota\,(\mathsf{op}_i) = (o_i, \iota_i)$ to mean that the annotation ι maps op_i to (o_i, ι_i).

Formally, we define I as the *greatest fixed point* of a set functor Φ, given by

$$\Phi(X) \stackrel{\text{def}}{=} \Sigma \Rightarrow (O \times X)_\perp$$

where $\Rightarrow$ is exponentiation, $\times$ is Cartesian product, and $(-)_\perp$ is the lifting operation, which we use to represent unannotated names, and which is defined using the disjoint union as $(-) \uplus \{\perp\}$. Formally speaking, I is given by an isomorphism $I \cong \Phi(I)$, but for presentation purposes we leave it implicit and work as if we had a strict equality $I = \Phi(I)$.

Subtyping and Recursive Effect Annotations. Both O and I come equipped with natural *partial orders*: for O, $\sqsubseteq_O$ is given simply by subset inclusion; and for I, the pointwise order $\sqsubseteq_I$ is characterised as follows:

$$\begin{aligned} \iota \sqsubseteq_I \iota' \quad \text{iff} \quad & \forall\,(\mathsf{op} \in \Sigma)\,(o'' \in O)\,(\iota'' \in I).\, \iota\,(\mathsf{op}) = (o'', \iota'') \implies \\ & \exists\,(o''' \in O)\,(\iota''' \in I).\, \iota'\,(\mathsf{op}) = (o''', \iota''') \wedge o'' \sqsubseteq_O o''' \wedge \iota'' \sqsubseteq_I \iota''' \end{aligned}$$

We also use the *product order* $\sqsubseteq_{O\times I}$, defined as $(o, \iota) \sqsubseteq_{O\times I} (o', \iota') \stackrel{\text{def}}{=} o \sqsubseteq_O o' \wedge \iota \sqsubseteq_I \iota'$. In particular, we use $\sqsubseteq_{O\times I}$ to define the subtyping relation for $\lambda_{\text{æ}}$'s computation types.

Furthermore, both O and I carry a *join-semilattice* structure, where $o \sqcup o' \in O$ is given simply by the union of sets $o \cup o'$, while $\iota \sqcup \iota' \in I$ is given pointwise as follows:

$$(\iota \sqcup \iota')(\mathsf{op}) \stackrel{\text{def}}{=} \begin{cases} (o'' \sqcup o''', \iota'' \sqcup \iota''') & \text{if } \iota\,(\mathsf{op}) = (o'', \iota'') \wedge \iota'\,(\mathsf{op}) = (o''', \iota''') \\ (o'', \iota'') & \text{if } \iota\,(\mathsf{op}) = (o'', \iota'') \wedge \iota'\,(\mathsf{op}) = \perp \\ (o''', \iota''') & \text{if } \iota\,(\mathsf{op}) = \perp \wedge \iota'\,(\mathsf{op}) = (o''', \iota''') \\ \perp & \text{if } \iota\,(\mathsf{op}) = \perp \wedge \iota'\,(\mathsf{op}) = \perp \end{cases}$$

Importantly, the partial orders $(O, \sqsubseteq_O)$ and $(I, \sqsubseteq_I)$ are both *ω-complete* and *pointed*, i.e., they form *pointed ω-cpos*, meaning that they have least upper bounds of all increasing ω-chains, and least elements (given by the empty set $\emptyset$ and the constant $\perp$-valued mapping, respectively). As a consequence, and as is well-known, *least fixed points* of continuous (endo)maps on them are then guaranteed to exist [AC98, GHK$^+$03]. For $\lambda_{\text{æ}}$, we are particularly interested in the least fixed points of continuous maps $f : I \to I$, so as to specify and typecheck code examples involving reinstallable interrupt handlers, as we illustrate in Section 5.1.

We also note that if we were only interested in the type safety of $\lambda_{\text{æ}}$, and not in typechecking reinstallable interrupt handler examples, then we would not need $(I, \sqsubseteq_I)$ to be *ω-complete*, and could have instead chosen I to be the *least fixed point* of the set functor Φ defined earlier, which is what we do for simplicity in our Agda formalisation. In this case, each interrupt handler annotation $\iota \in I$ would be a *finite* nesting of partial mappings.

Finally, we envisage that any future full-fledged high-level language based on $\lambda_{æ}$ would allow users to define their (recursive) effect annotations in a small domain-specific language, providing a syntactic counterpart to the domain-theoretic development we use in this paper.

Interrupt Actions. We mimic the act of triggering an interrupt handler for some interrupt op on an effect annotation (o, ι) through an *action* defined as follows:

$$\mathsf{op}{\downarrow}\,(o,\iota) \stackrel{\text{def}}{=} \begin{cases} (o \sqcup o', \iota[\mathsf{op} \mapsto \bot] \sqcup \iota') & \text{if } \iota\,(\mathsf{op}) = (o', \iota') \\ (o, \iota) & \text{otherwise} \end{cases}$$

If (o, ι) lists any interrupt handlers installed for op, then $\iota\,(\mathsf{op}) = (o', \iota')$, where (o', ι') specifies the effects of said handler code. Now, when the inward propagating interrupt reaches those interrupt handlers, it triggers the execution of the corresponding handler code, and thus the entire interrupted computation can also issue signals in o' and handle interrupts in ι'.

The notation $\iota[\mathsf{op} \mapsto \bot]$ sets ι to $\bot$ at op, and leaves it unchanged elsewhere. Mapping op to $\bot$ in the definition of $\downarrow$ captures that the interrupt op triggers all the corresponding interrupt handlers that are installed in the computation that it is propagated to.

3.3.3. *Typing Rules.* We characterise *well-typed values* using the judgement $\Gamma \vdash V : X$ and *well-typed computations* using the judgement $\Gamma \vdash M : X\,!\,(o, \iota)$. In both judgements, Γ is a *typing context*. The rules defining these judgements are respectively given in Figure 4 and 5.

$$\textsc{TyVal-Var}\ \frac{}{\Gamma, x : X, \Gamma' \vdash x : X} \qquad \textsc{TyVal-Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{TyVal-Pair}\ \frac{\Gamma \vdash V : X \qquad \Gamma \vdash W : Y}{\Gamma \vdash (V, W) : X \times Y} \qquad \textsc{TyVal-Promise}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \langle V \rangle : \langle X \rangle}$$

$$\textsc{TyVal-Inl}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \mathsf{inl}_Y\, V : X + Y} \qquad \textsc{TyVal-Inr}\ \frac{\Gamma \vdash W : Y}{\Gamma \vdash \mathsf{inr}_X\, W : X + Y} \qquad \textsc{TyVal-Fun}\ \frac{\Gamma, x : X \vdash M : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{fun}\ (x : X) \mapsto M : X \to Y\,!\,(o, \iota)}$$

Figure 4: Value Typing Rules.

Values. The rules for values are mostly standard. The only $\lambda_{æ}$-specific rule is TyVal-Promise, which states that in order to fulfil a *promise* of type $\langle X \rangle$, one has to supply a value of type X. In the rule TyVal-Var, we emphasise the position of the variable in the context, as it will become important once we extend the calculus with modal types in Section 5.3.

Computations. Analogously to values, the typing rules are standard for computation terms that $\lambda_{æ}$ inherits from FGCBV, with the $\lambda_{æ}$-rules additionally tracking effect information.

The first $\lambda_{æ}$-specific typing rule TyComp-Signal states that in order to issue a signal op in a computation that has type $X\,!\,(o, \iota)$, we must have $\mathsf{op} \in o$ and the type of the payload value has to match op's signature $\mathsf{op} : A_{\mathsf{op}}$.

The rule TyComp-Interrupt is used to type incoming interrupts. In particular, when the outside world propagates an interrupt op to a computation M of type $X\,!\,(o, \iota)$, the resulting computation $\downarrow \mathsf{op}\,(V, M)$ gets assigned the type $X\,!\,\mathsf{op}{\downarrow}\,(o, \iota)$, where the action $\mathsf{op}{\downarrow}\,(o, \iota)$ of the interrupt op on the annotation (o, ι) is given as discussed in Section 3.3.2.

The rule TyComp-Promise states that the interrupt handler code M has to return a fulfilled promise of type $\langle X \rangle$, for some type X, while possibly issuing signals o' and handling

$$\textsc{TyComp-Return}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \mathsf{return}\ V : X\,!\,(o,\iota)} \qquad \textsc{TyComp-Let}\ \frac{\Gamma \vdash M : X\,!\,(o,\iota) \qquad \Gamma, x : X \vdash N : Y\,!\,(o,\iota)}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : Y\,!\,(o,\iota)}$$

$$\textsc{TyComp-Apply}\ \frac{\Gamma \vdash V : X \to Y\,!\,(o,\iota) \qquad \Gamma \vdash W : X}{\Gamma \vdash V\,W : Y\,!\,(o,\iota)} \qquad \textsc{TyComp-MatchPair}\ \frac{\Gamma \vdash V : X \times Y \qquad \Gamma, x : X, y : Y \vdash M : Z\,!\,(o,\iota)}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{(x,y) \mapsto M\} : Z\,!\,(o,\iota)}$$

$$\textsc{TyComp-MatchEmpty}\ \frac{\Gamma \vdash V : 0}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\}_{Z!(o,\iota)} : Z\,!\,(o,\iota)} \qquad \textsc{TyComp-MatchSum}\ \frac{\Gamma \vdash V : X + Y \qquad \Gamma, x : X \vdash M : Z\,!\,(o,\iota) \qquad \Gamma, y : Y \vdash N : Z\,!\,(o,\iota)}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} : Z\,!\,(o,\iota)}$$

$$\textsc{TyComp-Signal}\ \frac{\mathsf{op} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X\,!\,(o,\iota)}{\Gamma \vdash \uparrow \mathsf{op}\,(V,M) : X\,!\,(o,\iota)} \qquad \textsc{TyComp-Interrupt}\ \frac{\Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X\,!\,(o,\iota)}{\Gamma \vdash \downarrow \mathsf{op}\,(V,M) : X\,!\,\mathsf{op}\downarrow(o,\iota)}$$

$$\textsc{TyComp-Promise}\ \frac{(o',\iota') = \iota\,(\mathsf{op}) \qquad \Gamma, x : A_{\mathsf{op}} \vdash M : \langle X \rangle\,!\,(o',\iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y\,!\,(o,\iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y\,!\,(o,\iota)}$$

$$\textsc{TyComp-Await}\ \frac{\Gamma \vdash V : \langle X \rangle \qquad \Gamma, x : X \vdash M : Y\,!\,(o,\iota)}{\Gamma \vdash \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M : Y\,!\,(o,\iota)} \qquad \textsc{TyComp-Subsume}\ \frac{\Gamma \vdash M : X\,!\,(o,\iota) \qquad (o,\iota) \sqsubseteq_{O \times I} (o',\iota')}{\Gamma \vdash M : X\,!\,(o',\iota')}$$

Figure 5: Computation Typing Rules.

interrupts ι', both of which are determined by the effect annotation ι of the entire computation, as $(o', \iota') = \iota\,(\mathsf{op})$. The variable p bound in the continuation, which sub-computations can block on to await op to arrive and be handled, also gets assigned the promise type $\langle X \rangle$.

It is worth noting that we could have had M simply return values of type X, but at the cost of not being able to implement some of the more interesting examples, such as the guarded interrupt handlers defined in Section 6.1. At the same time, for $\lambda_{æ}$'s type safety, it is crucial that p would have remained assigned the distinguished promise type $\langle X \rangle$.

The rule TyComp-Await simply states that after awaiting a promise of type $\langle X \rangle$, the continuation M can refer to the promised value using the variable x of type X.

Finally, the rule TyComp-Subsume allows *subtyping*, required to prove type preservation for rules where an interrupt encounters an interrupt handler. To simplify the presentation, we consider a limited form of subtyping, in which we shallowly relate only effect annotations.

3.4. **Type Safety.** The sequential part of $\lambda_{æ}$ satisfies the expected type safety properties ensuring that "well-typed programs do not go wrong". We split these safety properties into the usual *progress* and *preservation* theorems [WF94]. We omit their proofs [AP21] from this summary, and revisit them in Section 5.5 for the extended version of $\lambda_{æ}$, as the proofs for the extended calculus also apply to the version summarised in this section.

The progress result states that well-typed (and sufficiently) closed computations can either make another step of reduction, or they are already in a well-defined result form (and

thus have correctly stopped reducing). As such, we first need to define when we consider $\lambda_{æ}$-computations to be in *result form* (commonly also called a normal form). We do so using the judgements $\mathsf{CompRes}\langle\Psi \mid M\rangle$, which states that M has reached its final form as an isolated computation term, and $\mathsf{RunRes}\langle\Psi \mid M\rangle$, which states that M has reached the final form of a computation running inside a process with all its signals already having been propagated to other parallel processes (described in more detail in Section 4.4):

$$\frac{\mathsf{CompRes}\langle\Psi \mid M\rangle}{\mathsf{CompRes}\langle\Psi \mid \uparrow \mathsf{op}\,(V, M)\rangle} \qquad \frac{\mathsf{RunRes}\langle\Psi \mid M\rangle}{\mathsf{CompRes}\langle\Psi \mid M\rangle} \qquad \frac{}{\mathsf{RunRes}\langle\Psi \mid \mathsf{return}\ V\rangle}$$

$$\frac{\mathsf{RunRes}\langle\Psi \cup \{p\} \mid N\rangle}{\mathsf{RunRes}\langle\Psi \mid \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N\rangle} \qquad \frac{p \in \Psi}{\mathsf{RunRes}\langle\Psi \mid \mathsf{await}\ p\ \mathsf{until}\ \langle x\rangle\ \mathsf{in}\ M\rangle}$$

In these judgements, Ψ is a set of (promise-typed) variables p that have been bound by interrupt handlers enveloping the given computation. Intuitively, these judgements express that a computation M is in a (top-level) result form $\mathsf{CompRes}\langle\Psi \mid M\rangle$ when, considered as a tree, it has a shape in which *all* signals are towards the root, interrupt handlers are in the intermediate nodes, and the leaves contain return values and computations that are temporarily blocked while awaiting one of the promise-typed variables p in Ψ to be fulfilled.

The new reduction rules that propagate the awaiting construct out of sequencing and interrupts into the awaiting construct ensure the explicit form of all blocking computations and considerably simplify the definition of $\mathsf{RunRes}\langle\Psi \mid M\rangle$ compared to the previous version of our work [AP21]. The finality of these result forms is captured by the next lemma.

Lemma 3.1. *Given Ψ and M, such that $\mathsf{CompRes}\langle\Psi \mid M\rangle$, then there is no N with $M \rightsquigarrow N$.*

Using the result forms, the progress theorem for the sequential part of $\lambda_{æ}$ is as follows:

Theorem 3.2 (Progress for computations)**.** *Given a well-typed computation*

$$p_1 : \langle X_1\rangle, \ldots, p_n : \langle X_n\rangle \vdash M : Y\,!\,(o, \iota)$$

then either

(a) *there exists a computation N, such that $M \rightsquigarrow N$, or*
(b) *the computation M is in a result form, i.e., we have $\mathsf{CompRes}\langle\{p_1, \ldots, p_n\} \mid M\rangle$.*

In particular, with the empty context, we get the usual progress statement, which states that $\vdash M : X\,!\,(o, \iota)$ implies that either $M \rightsquigarrow N$ for some N or that $\mathsf{CompRes}\langle\emptyset \mid M\rangle$ holds. This implies that any promise variable which we are awaiting to be fulfilled must correspond to one of the installed interrupt handlers. Additionally, the type system ensures that all outgoing signals are listed in o and all installed interrupt handlers are specified in ι.

The type preservation result is standard and says that reduction preserves well-typedness.

Theorem 3.3 (Preservation for computations)**.** *Given a computation $\Gamma \vdash M : X\,!\,(o, \iota)$, such that M can reduce as $M \rightsquigarrow N$, then we have $\Gamma \vdash N : X\,!\,(o, \iota)$.*

4. A Calculus for Asynchronous Effects: Parallel Processes

We now describe the parallel part of $\lambda_{æ}$. Similarly to the sequential part, we present the corresponding syntax, small-step semantics, type-and-effect system, and type safety results.

4.1. **Parallel Processes.** To keep the presentation focussed on the asynchronous use of algebraic effects, we consider a very simple model of parallelism: a process is either an *individual computation* or the *parallel composition* of two processes. To facilitate interactions, processes also contain outward propagating *signals* and inward propagating *interrupts*.

In detail, the syntax of *parallel processes* is given by the following grammar:

$$P, Q \ ::= \ \mathsf{run}\ M \ \mid \ P \parallel Q \ \mid \ \uparrow \mathsf{op}\,(V, P) \ \mid \ \downarrow \mathsf{op}\,(V, P)$$

Note that processes do not include interrupt handlers—these are local to computations.

Here the number and hierarchy of processes running in parallel is fixed—a limitation that we address in Section 5.4 by introducing a means to dynamically create new processes.

4.2. **Small-Step Operational Semantics.** We equip the parallel part of $\lambda_{æ}$ with a small-step operational semantics that naturally extends the semantics of $\lambda_{æ}$'s sequential part from Section 3.2. The semantics is defined using a reduction relation $P \rightsquigarrow Q$, as given in Figure 6.

Individual computations

$$\frac{M \rightsquigarrow N}{\mathsf{run}\ M \rightsquigarrow \mathsf{run}\ N}$$

Signal hoisting

$$\mathsf{run}\ (\uparrow \mathsf{op}\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{run}\ M)$$

Broadcasting

$$\uparrow \mathsf{op}\,(V, P) \parallel Q \rightsquigarrow \uparrow \mathsf{op}\,(V, P \parallel \downarrow \mathsf{op}\,(V, Q))$$
$$P \parallel \uparrow \mathsf{op}\,(V, Q) \rightsquigarrow \uparrow \mathsf{op}\,(V, \downarrow \mathsf{op}\,(V, P) \parallel Q)$$

Interrupt propagation

$$\downarrow \mathsf{op}\,(V, \mathsf{run}\ M) \rightsquigarrow \mathsf{run}\ (\downarrow \mathsf{op}\,(V, M))$$
$$\downarrow \mathsf{op}\,(V, P \parallel Q) \rightsquigarrow \downarrow \mathsf{op}\,(V, P) \parallel \downarrow \mathsf{op}\,(V, Q)$$
$$\downarrow \mathsf{op}\,(V, \uparrow \mathsf{op}'\,(W, P)) \rightsquigarrow \uparrow \mathsf{op}'\,(W, \downarrow \mathsf{op}\,(V, P))$$

Evaluation context rule

$$\frac{P \rightsquigarrow Q}{\mathcal{F}[P] \rightsquigarrow \mathcal{F}[Q]}$$

where

$$\mathcal{F} \ ::= \ [\,] \ \mid \ \mathcal{F} \parallel Q \ \mid \ P \parallel \mathcal{F} \ \mid \ \uparrow \mathsf{op}\,(V, \mathcal{F}) \ \mid \ \downarrow \mathsf{op}\,(V, \mathcal{F})$$

Figure 6: Small-Step Operational Semantics of Processes.

Individual Computations. This rule states that, as processes, individual computations evolve according to the small-step operational semantics $M \rightsquigarrow N$ we defined in Section 3.2.

Signal Hoisting. This rule propagates signals out of individual computations. Note that we only hoist those signals that have propagated to the outer boundary of a computation.

Broadcasting. These rules turn outward moving signals in one process into inward moving interrupts for the process parallel to it, while continuing to propagate the signals outwards to any further parallel processes. The latter ensures that the semantics is compositional.

Interrupt Propagation. These three rules simply propagate interrupts inwards into individual computations, into all branches of parallel compositions, and past any issued signals.

Evaluation Contexts. Analogously to the semantics of computations, the semantics of processes presented here also includes an evaluation context rule, which allows reductions under *evaluation contexts* $\mathcal{F}$. Observe that compared to the evaluation contexts for computations, those for processes are more standard, in the sense that they do not bind variables.

4.3. **Type-and-Effect System.** Analogously to its sequential part, we also equip $\lambda_{æ}$'s parallel part with a type-and-effect system.

Types. The *process types* are designed to match their parallel structure, and are given by

$$C,\, D \;::=\; X \mathbin{!!} (o, \iota) \;\;\big|\;\; C \mathbin{||} D$$

Namely, $X \mathbin{!!} (o, \iota)$ is a type of an individual computation of type $X \mathbin{!} (o, \iota)$, and $C \mathbin{||} D$ is the type of the parallel composition of two processes that respectively have types C and D.

Typing Judgements. *Well-typed processes* are characterised using the judgement $\Gamma \vdash P : C$. The typing rules are given in Figure 7. While our processes are not currently higher-order, we allow non-empty contexts Γ to model using libraries and top-level function definitions.

$$\textsc{TyProc-Run}\quad \frac{\Gamma \vdash M : X \mathbin{!} (o, \iota)}{\Gamma \vdash \mathsf{run}\ M : X \mathbin{!!} (o, \iota)} \qquad \textsc{TyProc-Par}\quad \frac{\Gamma \vdash P : C \qquad \Gamma \vdash Q : D}{\Gamma \vdash P \mathbin{||} Q : C \mathbin{||} D}$$

$$\textsc{TyProc-Signal}\quad \frac{\mathsf{op} \in \mathsf{signals\text{-}of}(C) \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash P : C}{\Gamma \vdash \uparrow \mathsf{op}\,(V, P) : C} \qquad \textsc{TyProc-Interrupt}\quad \frac{\Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash P : C}{\Gamma \vdash \downarrow \mathsf{op}\,(V, P) : \mathsf{op} \downarrow C}$$

Figure 7: Process Typing Rules.

The rules TyProc-Run and TyProc-Par capture the earlier intuition about the types of processes matching their parallel structure. The rules TyProc-Signal and TyProc-Interrupt are similar to the corresponding computation typing rules from Figure 5.

The *signal annotations* of a process type used in TyProc-Signal are calculated as

$$\mathsf{signals\text{-}of}(X \mathbin{!!} (o, \iota)) \stackrel{\text{def}}{=} o \qquad \mathsf{signals\text{-}of}(C \mathbin{||} D) \stackrel{\text{def}}{=} \mathsf{signals\text{-}of}(C) \sqcup \mathsf{signals\text{-}of}(D)$$

and the *action of interrupts* on process types extends the action on effect annotations as

$$\mathsf{op} \downarrow (X \mathbin{!!} (o, \iota)) \stackrel{\text{def}}{=} X \mathbin{!!} (\mathsf{op} \downarrow (o, \iota)) \qquad \mathsf{op} \downarrow (C \mathbin{||} D) \stackrel{\text{def}}{=} (\mathsf{op} \downarrow C) \mathbin{||} (\mathsf{op} \downarrow D)$$

by propagating the interrupt towards the types of individual computations.

It is worth noting that Figure 7 does not include an analogue of the computation subtyping rule TyComp-Subsume. This choice is deliberate because as we shall see below, *process types reduce* in conjunction with the processes they are assigned to, and the outcome of process type reduction is generally neither a sub- nor supertype of the original type.

4.4. **Type Safety.** We conclude summarising the meta-theory of $\lambda_{æ}$ by stating the type safety of its parallel part. Analogously to Section 3.4, we once again split type safety into separate *progress* and *preservation* results, and relegate their proofs to Section 5.5.

We characterise the *result forms* of processes by defining two judgements, $\mathsf{ProcRes}\langle P\rangle$ and $\mathsf{ParRes}\langle P\rangle$, and by using the judgement $\mathsf{RunRes}\langle\Psi \mid M\rangle$ from Section 3.4, as follows:

$$\frac{\mathsf{ProcRes}\langle P\rangle}{\mathsf{ProcRes}\langle\uparrow \mathsf{op}\,(V, P)\rangle} \qquad \frac{\mathsf{ParRes}\langle P\rangle}{\mathsf{ProcRes}\langle P\rangle} \qquad \frac{\mathsf{RunRes}\langle\emptyset \mid M\rangle}{\mathsf{ParRes}\langle\mathsf{run}\ M\rangle} \qquad \frac{\mathsf{ParRes}\langle P\rangle \qquad \mathsf{ParRes}\langle Q\rangle}{\mathsf{ParRes}\langle P \parallel Q\rangle}$$

These judgements express that a process P is in a (top-level) result form $\mathsf{ProcRes}\langle P\rangle$ when, considered as a tree, it has a shape in which *all* signals are towards the root, parallel compositions are in the intermediate nodes, and individual computation results are at the leaves. Importantly, the computation results $\mathsf{RunRes}\langle\emptyset \mid M\rangle$ we use in this definition are those from which all signals have been propagated out of (as discussed in Section 3.4).

Again, these result forms are operationally final, as captured by the next lemma.

Lemma 4.1. *Given a process P, such that $\mathsf{ProcRes}\langle P\rangle$, then there is no Q such that $P \rightsquigarrow Q$.*

We are now ready to state the progress theorem for the parallel part of $\lambda_{æ}$, which applies to closed processes and takes the expected form:

Theorem 4.2 (Progress for processes)**.** *Given a well-typed process $\vdash P : C$, then either*

(a) *there exists a process Q, such that $P \rightsquigarrow Q$, or*
(b) *the process P is already in a (top-level) result form, i.e., we have $\mathsf{ProcRes}\langle P\rangle$.*

The preservation theorem for processes that we state below is somewhat non-standard since term reductions also evolve effect annotations. In particular, the broadcast rule

$$\uparrow \mathsf{op}\,(V, P) \parallel Q \rightsquigarrow \uparrow \mathsf{op}\,(V, P \parallel \downarrow \mathsf{op}\,(V, Q))$$

and its symmetric counterpart from Figure 6 introduce new inward propagating interrupts in their right-hand sides that originally do not exist in their left-hand sides. As a result, compared to the types one assigns to the left-hand sides of these reduction rules, the types assigned to their right-hand sides will need to feature corresponding type-level actions of these interrupts. We formalise this idea using a *process type reduction* relation $C \rightsquigarrow D$:

$$\frac{}{X \mathbin{!!} (o, \iota) \rightsquigarrow X \mathbin{!!} (o, \iota)} \qquad \frac{}{X \mathbin{!!} (\mathsf{ops}\downarrow\downarrow (o, \iota)) \rightsquigarrow X \mathbin{!!} (\mathsf{ops}\downarrow\downarrow (\mathsf{op}\downarrow (o, \iota)))} \qquad \frac{C \rightsquigarrow C' \qquad D \rightsquigarrow D'}{C \parallel D \rightsquigarrow C' \parallel D'}$$

where we write $\mathsf{ops}\downarrow\downarrow (o, \iota)$ for a recursively defined *action of a list of interrupts* on (o, ι):

$$[]\downarrow\downarrow (o, \iota) \stackrel{\text{def}}{=} (o, \iota) \qquad (\mathsf{op} :: \mathsf{ops})\downarrow\downarrow (o, \iota) \stackrel{\text{def}}{=} \mathsf{op}\downarrow (\mathsf{ops}\downarrow\downarrow (o, \iota))$$

Intuitively, $C \rightsquigarrow D$ describes how process types reduce by being acted upon by freshly arriving interrupts. It is important that we introduce interrupts under an arbitrary enveloping sequence of interrupt actions, and not simply as $X \mathbin{!!} (o, \iota) \rightsquigarrow X \mathbin{!!} (\mathsf{op}\downarrow (o, \iota))$, because we want to ensure that these actions preserve type reductions (see Lemma 5.6 (3)), which in turn ensures type preservation of reductions under arbitrary evaluation contexts $\mathcal{F}$.

Using the process type reduction relation, we state the preservation theorem for the parallel part of $\lambda_{æ}$ as follows:

Theorem 4.3 (Preservation for processes)**.** *Given a well-typed process $\Gamma \vdash P : C$, such that P can reduce as $P \rightsquigarrow Q$, then there exists a process type D, such that the process type C can reduce as $C \rightsquigarrow D$, and we can type the resulting process as $\Gamma \vdash Q : D$.*

5. Higher-Order Extensions

While $\lambda_{æ}$, as introduced in our original work [AP21] and summarised in the previous two sections, can be used to naturally capture a wide range of asynchronous examples, it also has many notable *limitations*: interrupt handlers disappear immediately after being triggered by a matching interrupt, payloads of signals and interrupts have to be ground values, and it is not possible to dynamically create new parallel processes. In this section we introduce and discuss a number of *higher-order extensions* of $\lambda_{æ}$ that resolve these limitations. Below we discuss each of these extensions individually, with the full extended calculus given in Appendix A. We highlight the parts of $\lambda_{æ}$ that change in this section's extensions with a grey background.

5.1. Reinstallable Interrupt Handlers.

We recall from the reduction rules in Figure 2 that once an interrupt reaches a matching interrupt handler, the handling computation is executed and the handler is removed. However, the example from Section 2.7 shows that we often want to keep the handler around, e.g., to handle further interrupts of the same kind. One option to achieve this is through general recursion [AP21]. Unfortunately, this results in programmers defining many auxiliary functions, obfuscating the resulting code. Furthermore, the heavy reliance on general recursion makes it difficult to justify leaving it out of the core calculus, despite it being an orthogonal concern to many programming abstractions and, in particular, to how we model asynchrony in $\lambda_{æ}$ based on algebraic effects—this is of course not to say that a higher-level language based on $\lambda_{æ}$ could not include general recursion.

Instead, in this paper we propose extending $\lambda_{æ}$'s interrupt handlers with the ability to *reinstall* themselves, by extending the syntax for interrupt handlers given in Section 3

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

with an additional variable r bound to a function through which M can reinstall the handler:

$$\mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$$

In contrast to the continuation/resumption variables of ordinary effect handlers, here the variable r does not refer to the continuation of the interrupt at the time triggering, but instead to the act of reinstalling the given interrupt handler. Concretely, the triggering of reinstallable interrupt handlers is captured operationally with the following reduction rule:

$$\begin{aligned}&\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)\\ &\quad \rightsquigarrow \mathsf{let}\ p = M[V/x, (\mathsf{fun}\ () \mapsto \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p)/r]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)\end{aligned}$$

All other reduction rules remain the same, except that interrupt handlers are extended with the additional variables r. Server-like processes can then be written more concisely, as

$$\mathsf{promise}\ (\mathsf{request}\ x\ r \mapsto \text{handle the } \mathsf{request}\text{; issue a } \mathsf{response} \text{ signal; } r\,())\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ ()$$

In light of the similarity between interrupt propagation and deep effect handling, as discussed in Section 3.2, this reinstalling behaviour can be understood as an effect handler re-calling (in its corresponding operation case) the algebraic operation that it is handling, such as, an exception handler handling an exception and then re-raising it at the end for other, external exception handlers.

The typing rule for reinstallable interrupt handlers is also quite interesting:

TyComp-RePromise

$$\frac{(o', \iota') \sqsubseteq_{O \times I} \iota\,(\mathsf{op}) \qquad \Gamma, x : A_{\mathsf{op}}, r : 1 \to \langle X \rangle\,!\,\big(\emptyset, \{\mathsf{op} \mapsto (o', \iota')\}\big) \vdash M : \langle X \rangle\,!\,(o', \iota') \qquad \Gamma, p : \langle X \rangle \vdash N : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N : Y\,!\,(o, \iota)}$$

First, observe that the context in which we type the interrupt handler code M is now extended with the variable r, which denotes a function triggered by application to the unit value $() : 1$. The function does not emit any signals nor install any handlers apart from the one in question for op, therefore its effect annotation is $\big(\emptyset, \{\mathsf{op} \mapsto (o', \iota')\}\big)$, as expected.

Second, we have relaxed the requirement $(o', \iota') = \iota\,(\mathsf{op})$. We now only require the effect annotation (o', ι') of the handler code M to be contained in what the effect annotation (o, ι) of the continuation N, and thus of the entire composite computation $\mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$, assigns to op, i.e., $(o', \iota') \sqsubseteq_{O \times I} \iota\,(\mathsf{op})$. The reason lies in the proof of type preservation (see Theorem 5.4) when propagating unhandled interrupts past handlers:

$$\downarrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x\ r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}'\,(V, N)$$

On the left-hand side of this reduction rule, the effect annotation of the continuation of promise is (o, ι), while on the right-hand side it is $\mathsf{op}' \downarrow (o, \iota)$. This mismatch did not pose a problem earlier [AP21] as subtyping allowed us to increase the effect annotation of M to $\pi_2\,(\mathsf{op}' \downarrow (o, \iota))(\mathsf{op})$. Now on the other hand, as M's effect annotation is also present in the type of r, it appears both co- and contravariantly, and is thus not safe to increase. However, the tight coupling of the effect annotations is not really essential, as for safety it is enough that the annotation of the continuation simply encompasses any effects that M may trigger.

As noted in Section 3.3.2, assigning types to reinstallable handlers requires us to consider least fixed points of continuous maps on the ω-cpo $(I, \sqsubseteq_I)$ of interrupt handler annotations. As an example, we recall the following fragment of the server code from Section 2.7.2:

```
...
promise (batchSizeReq () r ↦
   ↑ batchSizeResp batchSize;
   r ()
)
...
```

Here, the interrupt handler for batchSizeReq reinstalls itself immediately after issuing a batchSizeResp signal. Due to its recursive definition, it should not be surprising that this handler's effect annotation is given recursively, in particular, if we want to give it a more precise type-level specification than one which simply states that any effect is possible.

To that end, we assign this interrupt handler the effect annotation $(\emptyset, \iota_{\mathrm{b}})$, where

$$\iota_{\mathrm{b}} = \{\ \mathsf{batchSizeReq} \mapsto (\{\mathsf{batchSizeResp}\}, \{\ \mathsf{batchSizeReq} \mapsto (\{\mathsf{batchSizeResp}\},\ \ldots\ )\ \})\ \}$$

More precisely, ι_{b} is the least fixed point of the following continuous map on I:

$$\iota \mapsto \{\ \mathsf{batchSizeReq} \mapsto (\{\mathsf{batchSizeResp}\}, \iota)\ \} : I \to I$$

This least fixed point exists because I is an ω-cpo and the map is continuous (see Section 3.3.2).

Returning to the example above, the effect annotation $(\emptyset, \iota_{\mathrm{b}})$ specifies that the interrupt handler does not issue any signals at the top level, and that every batchSizeReq interrupt causes a batchSizeResp signal to be issued and the interrupt handler to be reinstalled.

The examples of reinstallable interrupt handlers that we discuss in Section 6 have their effect annotations assigned analogously, also as least fixed points of continuous maps on I.

5.2. **Stateful Reinstallable Interrupt Handlers.** When working with reinstallable interrupt handlers, it is often useful, and sometimes even necessary, to be able to pass data between subsequent reinstalls of a handler. For example, in Section 6.4 we use reinstallable interrupt handlers to implement a pseudorandom number generator in which it is crucial to be able to pass and update a seed value between reinstalls of an interrupt handler. As another example, consider wanting to react to only the first n interrupts of a particular kind—here it is useful if we could pass and decrease a counter between handler reinstalls.

In our original work [AP21], such state-passing behaviour was achieved by passing the relevant state values as arguments to the general-recursive functions that implemented the reinstalling of interrupt handlers. However, with reinstallability of interrupt handlers being now a primitive feature of $\lambda_{æ}$, we want a similarly primitive approach to managing state.

To this end, we extend the reinstallable interrupt handlers of last section with *state*:

$$\mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N$$

Here S denotes the type of state associated with a particular interrupt handler (S can be an arbitrary value type), s is a variable bound in the interrupt handler code M, giving it access to the handler's state at the time of triggering, and V is the value of state that is used at the next triggering of the interrupt handler. The state can be updated between subsequent reinstalls of the interrupt handler by calling the reinstallation function r with the new state value—r's domain is now S instead of 1. This behaviour is summarised by the reduction rule

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M\big[V/x, R/r, W/s\big]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$

where R denotes a function that reinstalls the interrupt handler with an updated state value:

$$R \stackrel{\mathrm{def}}{=} \mathsf{fun}\ (s' : S) \mapsto \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ s'\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

Needing to know S for the function abstraction in R necessitates the type annotation on this variant of interrupt handlers. All other reduction rules remain unchanged, except that interrupt handlers now include additional variables, type annotations, and values for states.

The typing rule for stateful reinstallable interrupt handlers is a straightforward extension of the typing rule for reinstallable interrupt handlers we presented in the previous section:

TyComp-ReStPromise

$$\frac{(o', \iota') \sqsubseteq_{O \times I} \iota\,(\mathsf{op}) \qquad \Gamma, x : A_{\mathsf{op}}, r : S \to \langle X \rangle\,!\,\big(\emptyset, \{\mathsf{op} \mapsto (o', \iota')\}\big), s : S \vdash M : \langle X \rangle\,!\,(o', \iota') \qquad \Gamma \vdash V : S \qquad \Gamma, p : \langle X \rangle \vdash N : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N : Y\,!\,(o, \iota)}$$

Observe that as noted above, the domain of r is no longer fixed to the unit type 1 but it can now be any value type S. If we pick $S \stackrel{\mathrm{def}}{=} 1$, we recover the stateless reinstallable interrupt handlers of the previous section, with all the highlighted parts trivialising. Therefore, as a convention, when working with reinstallable interrupt handlers with trivial state, we use the syntax introduced in the previous section, i.e., $\mathsf{promise}\ (\mathsf{op}\ x\, r \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$. Further, in examples we often omit the state type annotation S when it is clear from the context.

We now illustrate the use of stateful reinstallable interrupt handlers via the example mentioned earlier, of a program reacting to only the first n interrupts of a particular kind:

```
promise (op x r m ↦
    if (m > 0) then
        comp;
        r (m − 1)
    else
        return ⟨()⟩
) @ n
```

This interrupt handler carries a natural number counter as its state, which it uses to determine whether the handler computation comp should be run. The counter is originally set to the value n and then decremented each time the interrupt handler is reinstalled (using r (m − 1)). When the counter reaches 0, comp is not run and the interrupt handler is no longer reinstalled. More examples of stateful reinstallable interrupt handlers can be found in Section 2.7 and 6.

Finally, it is also worth noting that while the syntax of our stateful reinstallable interrupt handlers is somewhat similar to parameterised effect handlers [PP13], there is a subtle but important difference. Namely, as discussed in Section 3.2, interrupt handlers behave like algebraic operation calls and it is instead the interrupts that behave like effect handling. Thus, in light of the discussion in Section 5.1 about what reinstalling means, the stateful nature of our reinstallable interrupt handlers corresponds to changing a (state) parameter of an algebraic operation when it is re-called by the corresponding effect handler, and not to including and passing state values in effect handlers. In particular, any interrupt and its payload is passed to the continuation of an interrupt handler unchanged irrespectively of any state changes that happen when this interrupt handler is triggered and (possibly) reinstalled.

5.3. **Fitch-Style Modal Types.** The next limitation of $\lambda_{æ}$ we address is the restriction of signal and interrupt payloads to ground types, i.e., finite sums and products of base types. The reason behind this restriction lies in the propagation of signals past interrupt handlers:

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\ (V, N) \rightsquigarrow \uparrow \mathsf{op}'\ (V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$$

Here, we want to ensure that the value V on the left-hand side does not refer to the promise-typed variable p, otherwise the right-hand side would be ill-scoped. Note that the issue remains exactly the same when considering reinstallable or stateful interrupt handlers.

For example, consider a signal/interrupt $\mathsf{op} : \mathsf{int}$ carrying an integer payload, and $\mathsf{op}' : \langle \mathsf{int} \rangle$ carrying an integer-typed promise as a payload. Then, the computation

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto \mathsf{return}\ \langle 1 \rangle)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\ (p, \mathsf{return}\ 2)$$

which simply sends the promise-typed variable $p : \langle \mathsf{int} \rangle$ back in a signal payload, would be well-typed if no restrictions were put on signal and interrupt signatures. However, if this were allowed, then by the above reduction rule, this computation would step to

$$\uparrow \mathsf{op}'\ (p, \mathsf{promise}\ (\mathsf{op}\ x \mapsto \mathsf{return}\ \langle 1 \rangle)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ 2)$$

where now the payload p has escaped the binding scope of the interrupt handler, violating scope and type safety.

We run into similar problems when considering examples where payloads are higher-order, e.g., when wanting to send functions in payloads for remote execution in other processes. For

example, take op as before and consider a function-carrying signal/interrupt $\mathsf{op}' : (1 \to \mathsf{int})$, where for brevity, we omit the effect annotation in the function type. Then, the computation

$$\begin{aligned}&\mathsf{promise}\ (\mathsf{op}\ x \mapsto \mathsf{return}\ \langle 1 \rangle)\ \mathsf{as}\ p\ \mathsf{in}\\ &\mathsf{let}\ f = \mathsf{return}\ (\mathsf{fun}\ () \mapsto \mathsf{await}\ p)\ \mathsf{in}\\ &\uparrow \mathsf{op}'\ (f, \mathsf{return}\ 2)\end{aligned}$$

would again be well-typed if no restrictions were put on signal and interrupt signatures. At the same, it would first β-reduce the sequential composition to

$$\mathsf{promise}\ (\mathsf{op}\ x \mapsto \mathsf{return}\ \langle 1 \rangle)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\ ((\mathsf{fun}\ () \mapsto \mathsf{await}\ p), \mathsf{return}\ 2)$$

and then step to the computation

$$\uparrow \mathsf{op}'\ ((\mathsf{fun}\ () \mapsto \mathsf{await}\ p), \mathsf{promise}\ (\mathsf{op}\ x \mapsto \mathsf{return}\ \langle 1 \rangle)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ 2)$$

which is again ill-typed due to p escaping the interrupt handler's binding scope.

Restricting V to ground values is a simple way that ensures type-safety [AP21], but as a result, e.g., one can only send the arguments needed for the execution of remote function calls but not the functions themselves. When relaxing the payload restrictions, the type-system needs to track not only the use of promise-typed variables bound by interrupt handlers, but as f in the above example shows, also the use of any other variables that may depend on them. An elegant way of achieving this is a *Fitch-style modal type system* [Clo18], where the typing context Γ can contain (lock) tokens 🔓, which delimit the extent to which variables are allowed to be used in terms. In particular, terms can refer only to variables introduced after the last 🔓, and to a restricted subset of variables introduced before it.

Specifically, we extend the grammar of typing contexts to

$$\Gamma \ ::= \cdot \ \mid\ \Gamma, x : X \ \mid\ \Gamma, 🔓$$

and change the typing rule for variables to

$$\textsc{TyVal-Var}\ \frac{X \text{ is mobile} \quad \text{or} \quad 🔓 \notin \Gamma'}{\Gamma, x : X, \Gamma' \vdash x : X}$$

This means that we can refer only to variables introduced after the last 🔓, or to variables with *mobile types* A, defined as an extension of ground types with a *modal (box) type* $[X]$:

$$A, B \ ::= \ \mathsf{b} \ \mid\ 1 \ \mid\ 0 \ \mid\ A \times B \ \mid\ A + B \ \mid\ [X]$$

As with ground types, every mobile type is automatically also a value type, including $[X]$.

Note that $[X]$ is a mobile type even if X is not. Equally importantly, neither promise nor function types are mobile on their own. When combined with how the context is delimited using 🔓 in the typing rule TyVal-Box given below, these properties of mobile types ensure that signal payloads, which are typed with mobile types, cannot use promise-typed variables bound by enveloping interrupt handlers. Consequently, it is safe to propagate signals with mobile payloads past any enveloping interrupt handlers and eventually to other processes. In its essence, this is similar to the use of modal types in distributed [Mur08] and reactive programming [Kri13, BGM19] to classify values that can travel through space and time.

The type $[X]$ has a value constructor and a corresponding computation for elimination:

$$\begin{aligned}V, W \ &::= \ \cdots \ \mid\ [V]\\ M, N \ &::= \ \cdots \ \mid\ \mathsf{unbox}\ V\ \mathsf{as}\ [x]\ \mathsf{in}\ M\end{aligned}$$

with the evident reduction rule given by

$$\mathsf{unbox}\ [V]\ \mathsf{as}\ [x]\ \mathsf{in}\ M \rightsquigarrow M[V/x]$$

and with no associated evaluation contexts. More importantly, it is the typing rules that ensure one is allowed to box a value only when all the variables used in it have mobile types:

$$\textsc{TyVal-Box}\ \frac{\Gamma, \text{🔓} \vdash V : X}{\Gamma \vdash [V] : [X]} \qquad \textsc{TyComp-Unbox}\ \frac{\Gamma \vdash V : [X] \qquad \Gamma, x : X \vdash M : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{unbox}\ V\ \mathsf{as}\ [x]\ \mathsf{in}\ M : Y\,!\,(o, \iota)}$$

Since this prevents us from constructing boxed values that would refer to a promise-typed variable, we can safely extend payloads from ground to mobile types. Crucially however, when constructing a (payload) value of type $[X \to Y\,!\,(o, \iota)]$, the boxed function can itself install additional interrupt handlers, it just cannot refer to the results of any enveloping ones.

5.4. **Dynamic Process Creation.** It turns out that the same Fitch-style modal typing mechanism can be reused to extend $\lambda_{æ}$'s computations also with *dynamic process creation*:

$$M, N \ ::= \ \cdots \ \mid \ \mathsf{spawn}\ (M, N)$$

Here, M is the new computation to be spawned and N is the continuation of the existing program. Operationally, spawned computations propagate out of subcomputations as follows:

$$\mathsf{let}\ x = (\mathsf{spawn}\ (M_1, M_2))\ \mathsf{in}\ N \rightsquigarrow \mathsf{spawn}\ (M_1, \mathsf{let}\ x = M_2\ \mathsf{in}\ N)$$

$$\mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{spawn}\ (N_1, N_2) \rightsquigarrow \mathsf{spawn}\ (N_1, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N_2)$$

$$\downarrow \mathsf{op}\ (V, \mathsf{spawn}\ (M, N)) \rightsquigarrow \mathsf{spawn}\ (M, \downarrow \mathsf{op}\ (V, N))$$

This allows the newly spawned computation to reach the top-level of the program, where it becomes a new parallel process, as expressed by the following reduction rule for processes:

$$\mathsf{run}\ (\mathsf{spawn}\ (M, N)) \rightsquigarrow \mathsf{run}\ M \parallel \mathsf{run}\ N$$

Analogously to signals, the natural semantics of spawn is non-blocking. Consequently, we also include it in the definition of evaluation contexts of the sequential part of $\lambda_{æ}$:

$$\mathcal{E} \ ::= \ \cdots \ \mid \ \mathsf{spawn}\ (M, \mathcal{E})$$

The addition of spawn also requires us to extend the result forms of computations with

$$\frac{\mathsf{CompRes}\langle \Psi \mid N \rangle}{\mathsf{CompRes}\langle \Psi \mid \mathsf{spawn}\ (M, N) \rangle}$$

Finally, the typing rule for spawn is defined as follows:

$$\textsc{TyComp-Spawn}\ \frac{\Gamma, \text{🔓} \vdash M : X\,!\,(o, \iota) \qquad \Gamma \vdash N : Y\,!\,(o', \iota')}{\Gamma \vdash \mathsf{spawn}\ (M, N) : Y\,!\,(o', \iota')}$$

Here, we first use Fitch-style modal typing to ensure that M cannot refer to promise-typed variables bound by any enveloping interrupt handlers, making it safe to propagate it outwards, past them. This contrasts with other traditional concurrent/parallel languages, such as CML [Rep93], where no modal typing is needed because spawned processes do not need to be (operationally) propagated past any binding constructs local to individual processes.

What is perhaps even more surprising is that the type of the whole computation depends only on the type of the continuation N, and not on the type of the spawned computation M. This is because spawning M impacts only the execution of enveloping processes rather than of N itself. In that sense, one can see the spawning of M as a side-effect of N, and one possibility would be to extend effect annotations to a form (o, ι, ς), where

$$\varsigma = \{X_1 \,!\, (o_1, \iota_1, \varsigma_1), \ldots, X_n \,!\, (o_n, \iota_n, \varsigma_n)\}$$

tracks the types and effects of spawned processes (including any additional processes ς_i they may further spawn). However, this makes the type system significantly more complicated and brings few additional assurances. Indeed, at the process level, where spawned processes begin executing, the effect information is already very coarse since we need to account for actions of incoming interrupts. For that reason, we opt for a simpler, yet still type-sound solution (see Theorem 5.10), and instead extend the process type reduction relation with additional rules that allow spontaneously adding an arbitrary process type in parallel:

$$\frac{}{X \,!!\, (o, \iota) \rightsquigarrow (X \,!!\, (o, \iota)) \,||\, (Y \,!!\, (o', \iota'))} \qquad \frac{}{Y \,!!\, (o', \iota') \rightsquigarrow (X \,!!\, (o, \iota)) \,||\, (Y \,!!\, (o', \iota'))}$$

5.5. **Type Safety.** With all the higher-order extensions in place, we now prove type safety for the full, final version of $\lambda_{æ}$—first for computations and then for parallel processes.

While for brevity we do not repeat them here, we note that the finality results we proved about the result forms in Lemma 3.1 and 4.1 also hold for this extended version of $\lambda_{æ}$.

5.5.1. *Computations.* We recall that as standard, we split type safety into proofs of progress and preservation, with the former stated as follows (see also the discussion in Section 3.4):

Theorem 5.1 (Progress for computations)**.** *Given a well-typed computation*

$$p_1 : \langle X_1 \rangle, \ldots, p_n : \langle X_n \rangle \vdash M : Y \,!\, (o, \iota)$$

then either

(a) *there exists a computation N, such that $M \rightsquigarrow N$, or*
(b) *the computation M is in a result form, i.e., we have* $\mathsf{CompRes}\langle\{p_1, \ldots, p_n\} \,|\, M\rangle$.

Proof. The proof is standard and proceeds by induction on the derivation of $\Gamma \vdash M : Y \,!\, (o, \iota)$. For instance, if the derivation ends with a typing rule for function application or pattern-matching, we use an auxiliary canonical forms lemma to show that the value involved is either a function abstraction or in constructor form—thus M can β-reduce and we prove (a). Here we crucially rely on the context Γ having the specific form $p_1 : \langle X_1 \rangle, \ldots, p_n : \langle X_n \rangle$, with all the variables assigned promise types. If the derivation ends with TyComp-Await, we use a canonical forms lemma to show that the promise value is either a variable in Γ, in which case we prove (b), or in constructor form, in which case we prove (a). If the derivation ends with a typing rule for any of the terms figuring in the evaluation contexts $\mathcal{E}$, we proceed based on the outcome of using the induction hypothesis on the corresponding continuation. □

The results that we present in this section (and that we summarised in Section 3.4) use standard *substitution lemmas*. For instance, given $\Gamma, x : X, \Gamma' \vdash M : Y \,!\, (o, \iota)$ and $\Gamma \vdash V : X$, then we can show that $\Gamma, \Gamma' \vdash M[V/x] : Y \,!\, (o, \iota)$. In addition, we use standard *typing inversion lemmas*. For example, given a computation $\Gamma \vdash \downarrow \mathsf{op}\,(V, M) : X \,!\, (o, \iota)$, then we can show that $\Gamma \vdash V : A_{\mathsf{op}}$ and $\Gamma \vdash M : X \,!\, \mathsf{op}{\downarrow}\,(o', \iota')$, such that $\mathsf{op}{\downarrow}\,(o', \iota') \sqsubseteq_{O \times I} (o, \iota)$.

Furthermore, we use *strengthening lemmas* for promise-typed variables, such as if we have $\Gamma, p : \langle X \rangle, \Gamma' \vdash V : Y$, and if Γ' contains 🔓 or if Y is a mobile type, then also $\Gamma, \Gamma' \vdash V : Y$.

We also note that the action $\mathsf{op}{\downarrow}(-)$ has various useful properties that we use below (where we write π_1 and π_2 for the projections associated with the Cartesian product $O \times I$):

Lemma 5.2.

(1) $o \sqsubseteq_O \pi_1 (\mathsf{op}{\downarrow}(o, \iota))$
(2) *If* $\iota(\mathsf{op}) = (o', \iota')$, *then* $(o', \iota') \sqsubseteq_{O \times I} \mathsf{op}{\downarrow}(o, \iota)$
(3) *If* $\mathsf{op} \neq \mathsf{op}'$ *and* $(o', \iota') \sqsubseteq_{O \times I} \iota(\mathsf{op}')$, *then* $(o', \iota') \sqsubseteq_{O \times I} (\pi_2 (\mathsf{op}{\downarrow}(o, \iota)))(\mathsf{op}')$

Next, as the proof of type preservation proceeds by induction on reduction steps, we find it useful to define an auxiliary *typing judgement for evaluation contexts*, written

$$\Gamma \vdash [\, \Gamma' \mid X \,!\, (o, \iota) \,]\ \mathcal{E} : Y \,!\, (o', \iota')$$

which we then use to prove the evaluation context rule case of the preservation proof. In this judgement, Γ' is the context of variables bound by the interrupt handlers in $\mathcal{E}$, and $X \,!\, (o, \iota)$ is the type of the hole $[\,]$. This judgement is defined using rules similar to those for typing computations, including subtyping, e.g., for interrupt handlers we have the following rule:

$$\frac{\begin{array}{c}(o'', \iota'') \sqsubseteq_{O \times I} \iota'(\mathsf{op}) \\ \Gamma, x : A_{\mathsf{op}}, r : S \to \langle Y \rangle \,!\, \big(\emptyset, \{\mathsf{op} \mapsto (o'', \iota'')\}\big), s : S \vdash M : \langle Y \rangle \,!\, (o'', \iota'') \\ \Gamma \vdash V : S \qquad \Gamma, p : \langle Y \rangle \vdash [\, \Gamma' \mid X \,!\, (o, \iota) \,]\ \mathcal{E} : Z \,!\, (o', \iota')\end{array}}{\Gamma \vdash [\, p : \langle Y \rangle, \Gamma' \mid X \,!\, (o, \iota) \,]\ \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ \mathcal{E} : Z \,!\, (o', \iota')}$$

The typing of evaluation contexts is straightforwardly related to that of computations:

Lemma 5.3.

$$\big(\Gamma \vdash \mathcal{E}[M] : Y \,!\, (o', \iota')\big) \iff \exists \Gamma', X, o, \iota.\ \big(\Gamma \vdash [\, \Gamma' \mid X \,!\, (o, \iota) \,]\ \mathcal{E} : Y \,!\, (o', \iota')\big)\ \wedge\ \big(\Gamma, \Gamma' \vdash M : X \,!\, (o, \iota)\big)$$

We are now ready to prove the type preservation theorem for the sequential part of $\lambda_{æ}$.

Theorem 5.4 (Preservation for computations)**.** *Given a computation* $\Gamma \vdash M : X \,!\, (o, \iota)$, *such that* M *can reduce as* $M \rightsquigarrow N$, *then we have* $\Gamma \vdash N : X \,!\, (o, \iota)$.

Proof. The proof is standard and proceeds by induction on the derivation of $M \rightsquigarrow N$, using typing inversion lemmas based on the structure forced on M by the last rule used in $M \rightsquigarrow N$.

There are four cases of interest in this proof. The first two concern the interaction of interrupts and interrupt handlers. On the one hand, if the derivation of $\rightsquigarrow$ ends with

$$\mathsf{op}{\downarrow}(V, \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @\ W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M[V/x, R/r, W/s]\ \mathsf{in}\ \mathsf{op}{\downarrow}(V, N)$$

where R is a function that reinstalls the interrupt handler, then in order to type the right-hand side of this rule, we use subtyping with Lemma 5.2 (2) to show that M's effect information is included in that of $\mathsf{op}{\downarrow}(V, N)$, i.e., in $\mathsf{op}{\downarrow}(o, \iota)$. On the other hand, given the rule

$$\mathsf{op}'{\downarrow}(V, \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{op}'{\downarrow}(V, N) \quad (\mathsf{op} \neq \mathsf{op}')$$

then in order to type the right-hand side, we use subtyping with Lemma 5.2 (3), so as to show that after acting on (o, ι) with op', op remains mapped to M's effect information.

The third case of interest concerns the commutativity of signals with interrupt handlers:

$$\mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\ (W, N) \rightsquigarrow \uparrow \mathsf{op}'\ (W, \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N)$$

where in order to type the signal's payload W in the right-hand side of this rule, it is crucial that the promise-typed variable p cannot appear in W—this is ensured by our modal type system that restricts the signatures $\mathsf{op} : A_{\mathsf{op}}$ to mobile types. As a result, we can strengthen the typing context of W by removing the promise-typed variable p from it. We also use an analogous context strengthening argument for N_1 when given the other commutativity rule

$$\mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{spawn}\ (N_1, N_2) \rightsquigarrow \mathsf{spawn}\ (N_1, \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N_2)$$

Finally, in the evaluation context case, we use the induction hypothesis with Lemma 5.3. □

Interestingly, the proof of Theorem 5.4 tells us that if one were to consider a variant of $\lambda_{æ}$ in which the TyComp-Subsume rule appeared as an explicit coercion term $\mathsf{coerce}_{(o,\iota) \sqsubseteq_{O \times I} (o',\iota')}\ M$, which is the style we use in our Agda formalisation [Ahm24], then the right-hand sides of the two interrupt propagation rules highlighted in the above proof would also need to involve such coercions, corresponding to the two uses of Lemma 5.2. This however means that other computations involved in these reduction rules would also need to be type-annotated accordingly, so as to determine the data to be used in these coercions.

5.5.2. *Processes.* For the parallel part of $\lambda_{æ}$, we again first prove the progress theorem.

Theorem 5.5 (Progress for processes)**.** *Given a well-typed process $\vdash P : C$, then either*

(a) *there exists a process Q, such that $P \rightsquigarrow Q$, or*
(b) *the process P is already in a (top-level) result form, i.e., we have* $\mathsf{ProcRes}\langle P \rangle$.

Proof. The proof is unsurprising and proceeds by induction on the derivation of $\vdash P : C$. In the base case, when the derivation ends with the TyProc-Run rule and $P = \mathsf{run}\ M$, we use Theorem 5.1. In the other cases, we simply use the induction hypothesis. □

To prove preservation, we first focus on properties of the process type reduction $C \rightsquigarrow D$.

Lemma 5.6.

(1) *Process types can remain unreduced, i.e., $C \rightsquigarrow C$, for any process type C.*
(2) *Process types can reduce by being acted upon, i.e., $C \rightsquigarrow \mathsf{op}{\downarrow}C$, for any* op *and C.*
(3) *Process types can reduce under enveloping actions, i.e., $C \rightsquigarrow D$ implies $\mathsf{op}{\downarrow}C \rightsquigarrow \mathsf{op}{\downarrow}D$.*
(4) *Process type reduction can introduce signals but does not erase them, i.e., $C \rightsquigarrow D$ implies* $\mathsf{signals\text{-}of}(C) \sqsubseteq_O \mathsf{signals\text{-}of}(D)$.

The interesting case in the proof of Lemma 5.6 (3) is when the enveloped reduction $C \rightsquigarrow D$ introduces an interrupt op' under some sequence of interrupts ops, as follows:

$$X \mathbin{!!} \mathsf{ops}{\downarrow\downarrow}\,(o, \iota) \rightsquigarrow X \mathbin{!!} \mathsf{ops}{\downarrow\downarrow}\,(\mathsf{op}'{\downarrow}\,(o, \iota))$$

To prove this case, we simply prepend op to the list ops and reapply the same rule, as

$$\begin{array}{ccc} \mathsf{op}{\downarrow}\,(X \mathbin{!!} \mathsf{ops}{\downarrow\downarrow}\,(o, \iota)) & & \mathsf{op}{\downarrow}\,(X \mathbin{!!} \mathsf{ops}{\downarrow\downarrow}\,(\mathsf{op}'{\downarrow}\,(o, \iota))) \\ \| & & \| \\ X \mathbin{!!} (\mathsf{op} :: \mathsf{ops}){\downarrow\downarrow}\,(o, \iota) & \rightsquigarrow & X \mathbin{!!} (\mathsf{op} :: \mathsf{ops}){\downarrow\downarrow}\,(\mathsf{op}'{\downarrow}\,(o, \iota)) \end{array}$$

Observe that defining $C \rightsquigarrow D$ using a simpler basic rule $X \mathbin{!!} (o, \iota) \rightsquigarrow X \mathbin{!!} (\mathsf{op}' {\downarrow}\, (o, \iota))$ would not have been sufficient to prove this case, i.e., $\mathsf{op} {\downarrow}\, (X \mathbin{!!} (o, \iota)) \rightsquigarrow \mathsf{op} {\downarrow}\, (X \mathbin{!!} (\mathsf{op}' {\downarrow}\, (o, \iota)))$.

For the proof of Lemma 5.6 (4), we generalise Lemma 5.2 (1) to lists of actions.

Lemma 5.7. $\pi_1\,(\mathsf{ops} {\downarrow\downarrow}\, (o, \iota)) \sqsubseteq_O \pi_1\,(\mathsf{ops} {\downarrow\downarrow}\, (\mathsf{op} {\downarrow}\, (o, \iota)))$

As with computations, it is useful to define a separate *typing judgement for evaluation contexts*, this time written $\Gamma \vdash [\, C \,]\ \mathcal{F} : D$, together with an analogue of Lemma 5.3, which we omit here. Instead, we observe that this typing judgement preserves process type reduction.

Lemma 5.8. *Given* $\Gamma \vdash [\, C \,]\ \mathcal{F} : D$ *and* $C \rightsquigarrow C'$*, then there exists* D' *with* $D \rightsquigarrow D'$*, and we have* $\Gamma \vdash [\, C' \,]\ \mathcal{F} : D'$*.*

Process types also satisfy an analogue of Lemma 5.2 (1), which shows that the action $\mathsf{op} {\downarrow}\, (-)$ of interrupts on process types does not erase any already specified outgoing signals.

Lemma 5.9. *For any* C *and* op*, we have* $\mathsf{signals\text{-}of}(C) \sqsubseteq_O \mathsf{signals\text{-}of}(\mathsf{op} {\downarrow}\, C)$.

Finally, using the results above, we prove type preservation for the parallel part of $\lambda_{æ}$.

Theorem 5.10 (Preservation for processes)**.** *Given a well-typed process* $\Gamma \vdash P : C$*, such that* P *can reduce as* $P \rightsquigarrow Q$*, then there exists a process type* D*, such that the process type* C *can reduce as* $C \rightsquigarrow D$*, and we can type the resulting process as* $\Gamma \vdash Q : D$*.*

Proof. The proof proceeds by induction on the derivation of $P \rightsquigarrow Q$, using auxiliary typing inversion lemmas depending on the structure forced upon P by the last rule used in $P \rightsquigarrow Q$.

For most of the cases, we can pick D to be C and use Lemma 5.6 (1). For process creation, i.e., for the interaction of run and spawn, we define D by composing C in parallel with the spawned process's type, and build $C \rightsquigarrow D$ using the new type reduction rule

$$\frac{}{Y \mathbin{!!} (o', \iota') \rightsquigarrow (X \mathbin{!!} (o, \iota)) \mid\mid (Y \mathbin{!!} (o', \iota'))}$$

that we introduced in Section 5.4.

For the broadcast rules, we define D by introducing the corresponding interrupt, and build $C \rightsquigarrow D$ using the parallel composition rule together with Lemma 5.6 (2).

For the evaluation context rule, we use Lemma 5.8 in combination with the induction hypothesis. Finally, in order to discharge effect annotations-related side-conditions when commuting incoming interrupts with outgoing signals, we use Lemma 5.9. □

6. Asynchronous Effects in Action

We now show examples of the kinds of programs one can write in $\lambda_{æ}$. Similarly to Section 2.7, we again allow ourselves access to mutable references as a matter of convenience. We use these references only for (function call) counters and for communicating data between different parts of a program—passing data between subsequent reinstalls of the same interrupt handler is dealt with using the stateful reinstallable interrupt handlers introduced in Section 5.2.

In addition to the generic versions of constructs defined in Section 2.7, we further use

$$\mathsf{spawn}\ M \stackrel{\text{def}}{=} \mathsf{spawn}\ (M, \mathsf{return}\ ())$$

$$\mathsf{unbox}\ V \stackrel{\text{def}}{=} \mathsf{unbox}\ V\ \mathsf{as}\ [x]\ \mathsf{in}\ \mathsf{return}\ x$$

6.1. **Guarded Interrupt Handlers.** Before diving into the examples, we note that we often want the triggering of interrupt handlers to be conditioned on not only the names of interrupts, but also on the payloads that they carry. In order to express such more fine-grained interrupt handler triggering behaviour, we shall use a *guarded interrupt handler*:

```
promise (op x r s when guard ↦ comp) @ v
```

which is simply a syntactic sugar for the following stateful interrupt handler that reinstalls itself until the boolean guard becomes true, in which case it executes the handler code comp:

```
promise (op x r s ↦ if guard then comp else r s) @ v
```

where x and s are bound both in guard and comp. This means that the handler triggering can be conditioned both on the payload and state values. Meanwhile, r is bound only in comp. Also, note that regardless whether guard is true, every interrupt gets propagated into cont.

As guarded interrupt handlers repeatedly reinstall themselves, they get assigned recursive effect annotations, as discussed in Section 5.1. For example, if comp has type $\langle X \rangle \,!\, (o, \iota)$, then the corresponding guarded interrupt handler gets assigned the type $\langle X \rangle \,!\, (\emptyset, \iota_h)$, where

$$\iota_h = \{\mathsf{op} \mapsto (o, \iota \sqcup \{\mathsf{op} \mapsto (o, \iota \sqcup \{\mathsf{op} \mapsto (o, \cdots)\})\})\}$$

is the least fixed point of the continuous map $\iota' \mapsto \{\mathsf{op} \mapsto (o, \iota \sqcup \iota')\} : I \to I$. As such, the type $\langle X \rangle \,!\, (\emptyset, \iota_h)$ specifies that the installation of the guarded interrupt handler does not issue any signals by itself, and that the arrival of any op interrupt causes either the effects (o, ι) of comp to happen, or the interrupt handler to be reinstalled. Observe that as a consequence, some of the recursive encoding leaks via ι_h into the type of guarded interrupt handlers.

Similarly to reinstallable interrupt handlers, we write promise (op x r when guard ↦ comp) when the state associated with the guarded interrupt handler is trivial and can be omitted.

6.2. **Pre-Emptive Multi-Threading.** Multi-threading remains one of the most exciting applications of algebraic effects, with the possibility of modularly and user-definably expressing many evaluation strategies being the main reason for the extension of OCaml with effect handlers [OCa]. These evaluation strategies are however *cooperative* in nature, where each thread needs to explicitly yield back control, stalling other threads until then.

While it is possible to simulate *pre-emptive multi-threading* within the usual treatment of algebraic effects, it requires a low-level access to the specific runtime environment, so as to inject yields into the currently running computation. In contrast, implementing pre-emptive multi-threading in $\lambda_{æ}$ is quite straightforward, and importantly, possible within the language itself—the injections into the running computation take the form of incoming interrupts.

For the purpose of modelling pre-emptive multi-threading, let us consider two interrupts, stop : 1 and go : 1, that communicate to a thread whether to *pause* or *resume* execution. For example, these interrupts might originate from a timer process being run in parallel.

At the core of our implementation of pre-emptive multi-threading is the computation term waitForStop () that is defined as the following reinstallable interrupt handler:

```
let waitForStop () =
  promise (stop _ r ↦
    let p = promise (go _ ↦ return ⟨()⟩) in
    await p;
    r ()
  )
```

which first installs an interrupt handler for stop, letting subsequent computations run their course. Once a stop interrupt arrives, the interrupt handler for it is triggered and the next one for go is installed. In contrast to the interrupt handler for stop, we now start awaiting the promise p. This means that any subsequent computations are blocked until a go interrupt is received, after which we reinstall the interrupt handler for stop and repeat the cycle.

To *initiate the pre-emptive behaviour* for some computation comp, we run the program

```
waitForStop (); comp
```

The algebraicity reduction rules for interrupt handlers ensure that they propagate out of waitForStop and eventually encompass the entire composite computation, including comp. It is important to note that in contrast to the usual effect handlers based encodings of multi-threading, waitForStop does not need any access to a thunk fun () ↦ comp representing the threaded computation. In particular, the computation comp that we want to pre-empt can be completely unaware of the multi-threaded behaviour, both in its definition and type.

This approach can be easily extended to multiple threads, by using interrupts' payloads to communicate thread IDs. To this end, we can consider interrupts stop : int and go : int, and use guarded interrupt handlers to define a thread ID sensitive version of waitForStop:

```
let waitForStop threadID =
  promise (stop threadID' r when threadID = threadID' ↦
    let p = promise (go threadID' when threadID = threadID' ↦ return ⟨()⟩) in
    await p;
    r ()
  )
```

with the triggering of the interrupt handlers being conditional on the received thread IDs.

6.3. **Remote Function Calls.** One of the main uses of asynchronous computation is to offload the execution of *long-running functions* to remote processes. Below we show how to implement this in $\lambda_{æ}$ in a way that requires minimal cooperation from the remote process.

For a simpler exposition, we assume a fixed (mobile) result type A shared by all functions that we may wish to execute remotely. For communicating a function to be executed to the remote process, we assume a signal call : $[1 \to 1\,!\,(o, \iota)]$. Finally, for communicating the remote function call's A-typed result back to the caller, we assume a signal result : $A \times$ int.

The caller then calls functions f remotely through a wrapper function, remoteCall, which issues a call signal, installs a handler for a result interrupt, and returns a thunk that can be used to block the caller program's execution and await the remote function's result:

```
let remoteCall f =
    let callNo = !callCounter in callCounter := !callCounter + 1;
    let task = [ fun _ → let g = unbox f in
                         let res = g () in
                         ↑ result (res, callNo) ]
    in
    ↑ call task;
    let resultPromise = promise (result (y, callNo') when callNo = callNo' ↦ return ⟨y⟩) in
    let awaitResult () = await resultPromise in
    return awaitResult
```

Observe that the function f is not sent directly in the payload of the call signal to the remote process. Instead, call's payload combines the task of executing f with issuing a result signal with the function's result. This ensures that the result is always sent back to the caller, and the callee process can have a very simple implementation (see below). In addition, this combination explains why the signature of call does not mention A. Further, we note that in order to ensure that the payload is a boxed value, as required by call's signature, the function f has to be passed to remoteCall in a boxed form (notice the use of unbox in task).

To avoid the results of earlier remote function calls from fulfilling the promises of later ones, we assign to each call a unique identifier, which we implement using a counter local to the caller process. The identifier is passed together with the result and a guarded interrupt handler is used to ensure that only the result of the correct call is awaited. Note that this policy is again enforced by the caller and does not require any cooperation from the callee.

We also note that the effect annotation (o, ι) in call's signature can be used to limit the effects the caller may trigger in the callee process—it also influences the effects of functions f that one can call the remoteCall wrapper with. In order to be able to communicate the remote function's result back to the caller in task, o should include at least the result signal.

For instance, one may then call remote functions in their code as follows:

```
let subtally = remoteCall [ fun () → query "SELECT count(col) FROM table WHERE cond" ] in
let tally = remoteCall [ fun () → query "SELECT count(col) FROM table" ] in
printf "Percentage: %d" (100 * subtally () / tally ())
```

In the *callee process*, we simply install an interrupt handler that spawns a new process for executing the received function and then immediately recursively reinstalls itself, as follows:

```
promise (call boxedTask r ↦
    spawn (let f = unbox boxedTask in
           f ());
    r ()
)
```

Observe that as the payload of the call interrupt is received in a boxed form, it has to be unboxed before we are able to execute its underlying function. Here it is important that this unboxing happens inside the argument of spawn and not before the call to spawn. Namely, as the argument of spawn has to be mobile, its context is delimited by 🔓 (as discussed in Section 5.4) and therefore it can only refer to variables with mobile types bound outside of it, and whereas the type of boxedTask is mobile, the type of the underlying function is not.

This example can be naturally generalised to allow the remotely executed functions to take non-unit arguments: on the one hand, simply by passing arguments to the callee using the remoteCall wrapper function, or on the other hand, by defining separate wrapper functions for communicating the function and a particular call's arguments to the callee one at a time. We omit this generalisation here, but refer the reader to our original work [AP21] for an example of remote function calls being triggered by passing a particular call's arguments to the callee. However, it is important to highlight that whereas in our original work we were limited to only sending arguments to a fixed remote function, the modal boxed types adopted in this paper would enable the caller to dynamically also pass functions to the callee.

Unlike effect handlers, our interrupt handlers have very limited control over the execution of their continuation. Regardless, we can still simulate *cancellations of asynchronous computations* using the ideas behind our implementation of pre-emptive multithreading that we described in Section 6.2. Specifically, we modify the remoteCall wrapper function so that it returns an additional *cancellation thunk*, which can be used to cancel the computation:

```
let remoteCancellableCall f =
  let callNo = !callCounter in callCounter := !callCounter + 1;
  let task = [ fun _ → waitForCancel callNo;
                       let g = unbox f in
                       let res = g () in
                       ↑ result (res, callNo) ]
  in
  ↑ call task;
  let resultPromise = promise (result (y, callNo') when callNo = callNo' ↦ return ⟨y⟩) in
  let awaitResult () = await resultPromise in
  let cancelCall () = ↑ cancel callNo in
  return (awaitResult, cancelCall)
```

and where the function used to implement cancellations in the payload of call is defined as

```
let waitForCancel callNo =
  promise (cancel callNo' when callNo = callNo' →
    let p = promise (impossible _ → return ⟨()⟩) in
    await p;
    return ⟨()⟩
  )
```

for which we assume two additional signals: cancel : int and impossible : 0.

The callee code remains unchanged. Running each remote call in a separate process ensures that each cancel interrupt affects only one remote function call. In our original work [AP21], where all remote calls were executed in a single process, we additionally needed an auxiliary reinvoker process to continue executing the non-cancelled remote function calls.

Finally, we observe that the cancelled computation is only *perpetually stalled* (indefinitely awaiting the impossible interrupt, which can never be propagated to the process due to its 0-typed signature) but not discarded completely, leading to a memory leak. We conjecture that extending $\lambda_{æ}$ with interrupts and interrupt handlers that have greater control over their continuations could lead to a more efficient, memory leak-free code for the callee site.

6.4. **Runners of Algebraic Effects.** Next, we show how to use $\lambda_{æ}$ to implement a parallel variant of *runners of algebraic effects* [AB20]. These are a natural mathematical model and programming abstraction for resource management based on algebraic effects, and correspond to effect handlers that resume continuations (at most) once in a tail call position.

In a nutshell, for a signature of operation symbols $\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}$, a *runner* $\mathcal{R}$ comprises a family of stateful functions $\overline{\mathsf{op}}_{\mathcal{R}} : A_{\mathsf{op}} \times R \to B_{\mathsf{op}} \times R$, called *co-operations*, where R is the type of *resources* that the particular runner manipulates. In the more general setting, the co-operations also model other, external effects, such as native calls to the operating system, and can furthermore raise exceptions—all of which we shall gloss over here.

Given a runner $\mathcal{R}$, the programmer is provided with a construct

$$\mathsf{using}\ \mathcal{R}\ @\ V_{\mathrm{init}}\ \mathsf{run}\ M\ \mathsf{finally}\ \{\mathsf{return}\ x\ @\ r_{\mathrm{fin}} \mapsto N\}$$

which runs M using $\mathcal{R}$, with resources initially set to V_{init}; and finalises the return value (bound to x) and final resources (bound to r_{fin}) using the computation N, e.g., ensuring that all file handles get closed. This is a form of effect handling: it executes M by invoking co-operations in place of operation calls, while doing resource-passing under the hood. Below we show by means of examples how one can use $\lambda_{æ}$ to naturally separate $\mathcal{R}$ and M into different processes. For simplicity, we omit the initialisation and finalisation phases.

For our first example, let us consider a runner that implements a *pseudorandom number generator* by providing a co-operation for $\mathsf{random} : 1 \to \mathsf{int}$, which we can implement as

```
let linearCongruenceGeneratorRunner modulus a c initialSeed =
  promise (randomReq callNo r seed ↦
    let seed' = (a * seed + c) mod modulus in
    ↑ randomRes (seed, callNo);
    r seed'
  ) @ initialSeed
```

It is given by a recursive interrupt handler, which listens for $\mathsf{randomReq} : \mathsf{int}$ requests issued by clients, and itself issues $\mathsf{randomRes} : \mathsf{int} \times \mathsf{int}$ responses. The resource that this runner manages is the seed, which it passes between subsequent co-operation calls using the state-passing features provided by our reinstallable interrupt handlers. The seed is originally set to initialSeed, recalculated during each execution of the interrupt handler, and passed to the next co-operation call by reinstalling the interrupt handler with the updated seed value.

In the client, we implement operation calls random () as discussed in Section 2.2, by decoupling them into signals and interrupt handling. We use guarded interrupt handlers and call identifiers to avoid a response to one operation call fulfilling the promises of other ones.

```
let random () =
  let callNo = !callCounter in callCounter := callNo + 1;
  ↑ randomReq callNo;
  let p = promise (randomRes (n, callNo') when callNo = callNo' ↦ return ⟨n mod 10⟩) in
  await p
```

As a second example of runners, we show that this parallel approach to runners naturally extends to multiple co-operations. Specifically, we implement a *runner for a heap*, which provides co-operations for the following three operation symbols:

$$\mathsf{alloc} : \mathsf{int} \to \mathsf{loc} \qquad \mathsf{lookup} : \mathsf{loc} \to \mathsf{int} \qquad \mathsf{update} : \mathsf{loc} \times \mathsf{int} \to 1$$

We represent the co-operations using a signal/interrupt pair (opReq, opRes) with respective payload types payloadReq × int and payloadRes × int, tagged with call identifiers, and where

```
type payloadReq = AllocReq of int | LookupReq of loc | UpdateReq of loc * int
type payloadRes = AllocRes of loc | LookupRes of int | UpdateRes of unit
```

The resulting runner is implemented by pattern-matching on the payload value as follows:

```
let heapRunner initialHeap =
  promise (opReq (payloadReq, callNo) r heap ↦
    let heap', payloadRes =
      match payloadReq with
      | AllocReq v ↦
          let heap', l = allocHeap heap v in
          return (heap', AllocRes l)
      | LookupReq l ↦
          let v = lookupHeap heap l in
          return (heap, LookupRes v)
      | UpdateReq (l, v) ↦
          let heap' = updateHeap heap l v in
          return (heap', UpdateRes ())
    in
    ↑ opRes (payloadRes, callNo);
    r heap'
  ) @ initialHeap
```

The resource that this runner manages is the heap—it is initially set to initialHeap, and then updated and passed between subsequent co-operation calls analogously to the seed in the previous example. On the client side, the operation calls for allocation, lookup, and update are also implemented similarly to how random () was defined in the previous example.

Finally, we note that we could have instead used three signal/interrupt pairs and split heapRunner into three distinct reinstallable interrupt handlers, one for each of the three co-operations. However, then we would not have been able to use the state-passing provided by our interrupt handlers and we would have had to store the heap in the memory instead.

6.5. **Non-Blocking Post-Processing of Promised Values.** As discussed in Section 2.4, interrupt handlers differ from ordinary operation calls by allowing user-side post-processing of received data in the handler code. In this example, we show that $\lambda_{æ}$ is flexible enough to modularly perform *further non-blocking post-processing* of this data anywhere in a program.

For instance, let us assume we are writing a program that contains an interrupt handler (for some op) that promises to return us a list of integers. Let us further assume that at some later point in the program we decide that we want to further process this list if and when it becomes available, e.g., by using some of its elements to issue an outgoing signal. Of course, we could do this by going back and changing the definition of the original interrupt handler, but this would not be very modular; nor do we want to block the entire program's execution (using await) until the op interrupt arrives and the concrete list becomes available.

Instead, we can define a generic combinator for *non-blocking post-processing* of promises

```
process_op p with (⟨x⟩ ↦ comp)
```

that takes an earlier made promise p (which we assume originates from handling the specified interrupt op), and makes a new promise to execute the post-processing code comp[v/x] once p gets fulfilled with some value v. Under the hood, $\mathsf{process}_{\mathsf{op}}$ is simply a syntactic sugar for

```
promise (op _ ↦ let x = await p in let y = comp in return ⟨y⟩)
```

While $\mathsf{process}_{\mathsf{op}}$ involves an await, it gets exposed only after op is received, but by that time p will have been fulfilled with some v by an earlier interrupt handler, and thus await can reduce.

Returning to post-processing a list of integers promised by some interrupt handler, below is an example showing the use of the $\mathsf{process}_{\mathsf{op}}$ combinator and how to *chain together multiple post-processing computations* (filtering, folding, and issuing a signal), in the same spirit as how one is taught to program compositionally with futures and promises [HPM+20]:

```
let p = promise (op x ↦ initialHandler) in
...
let q = process_op p with (⟨is⟩ ↦ filter (fun i ↦ i > 0) is) in
let r = process_op q with (⟨js⟩ ↦ fold (fun j j' ↦ j * j') 1 js) in
process_op r with (⟨k⟩ ↦ ↑ productOfPositiveElements k);
...
```

For this to work, it is crucial that incoming interrupts behave like (deep) effect handling and propagate into continuations (see Section 3.2) so that all three post-processing computations get executed, in their program order, when an interrupt op is propagated to the program.

7. Conclusion

We have shown how to incorporate asynchrony within algebraic effects, by decoupling the execution of operation calls into signalling that an operation's implementation needs to be executed, and interrupting a running computation with the operation's result, to which it can react by installing interrupt handlers. We have shown that our approach is flexible enough that not all signals have to have a matching interrupt, and vice versa, allowing us to also model spontaneous behaviour, such as a user clicking a button or the environment pre-empting a thread. We have formalised these ideas in a small calculus, called $\lambda_{æ}$, and demonstrated its flexibility on a number of examples. We have also accompanied the paper with an Agda formalisation of $\lambda_{æ}$'s type safety and a prototype implementation of $\lambda_{æ}$.

Compared to our original work [AP21], in this extended version we have simplified the meta-theory of $\lambda_{æ}$, removed the reliance on general recursion for reinstalling interrupt handlers, added a notion of state to reinstallable interrupt handlers, and extended $\lambda_{æ}$ with higher-order signal and interrupt payloads, and with dynamic process creation. However, various future work directions still remain. We discuss these and related work below.

Asynchronous Effects. As asynchrony is desired in practice, it is no surprise that Koka [Lei17] and OCaml [DEH+17, SDW+21], the two largest implementations of algebraic effects and effect handlers, have been extended accordingly. In Koka, algebraic operations reify their continuation into an explicit callback structure that is then dispatched to a primitive such as setTimeout in its Node.JS backend. In OCaml, one writes effectful operations in

a direct style, but then uses handlers to access the actual asynchronous I/O through calls to an external library such as LIBUV. Both approaches thus *delegate* the actual asynchrony to existing concepts in their backends. In contrast, using $\lambda_{æ}$, we can express such backend features solely within the core calculus and the prototype implementation of it.

Further, in $\lambda_{æ}$, we avoid having to manually use (un)masking to disable asynchronous effects in unwanted places, which can be a very tricky business to get right [DEH+17]. Instead, by design, interrupts in $\lambda_{æ}$ *never* influence running code unless the code has an explicit interrupt handler installed, and they *always* wait for any potential handler to present itself during execution (recall that they get discarded only when reaching a return).

Finally, it is also worth discussing how signals and interrupts in $\lambda_{æ}$ compare to asynchronous exceptions, e.g., as found in Haskell [MJMR01]. The two mechanisms are similar in that both are issued outside of the running process. While asynchronous exceptions are thrown to a specific thread, we can simulate this in our broadcast-based semantics by carrying extra identifying information in signal and interrupt payloads, as discussed in Section 6.1. There is however a crucial difference between the two approaches: while interrupts only affect a given computation when a matching interrupt handler is installed, and they get always discarded when they reach the program's return clause, then asynchronous exceptions behave in the exact opposite way, causing the program to stop with a thrown exception unless the asynchronous exception is caught and handled away by the programmer.

Message-Passing. While in this paper we have focussed on the foundations of asynchrony in the context of programming with algebraic effects, the ideas we propose have also many common traits with concurrency models based on *message-passing*, such as the Actor model [HBS73], the π-calculus [MPW92], and the join-calculus [FG96], just to name a few. Namely, one can view the issuing of a signal $\uparrow \mathsf{op}\,(V, M)$ as sending a message, and handling an interrupt $\downarrow \mathsf{op}\,(W, M)$ as receiving a message, both along a channel named op. In fact, we believe that in our prototype implementation we could replace the semantics presented in the paper with an equivalent one based on shared channels (one for each op), to which the installed interrupt handlers could subscribe to. Instead of propagating signals first out of and then back into processes, they would then be sent directly to channels where interrupt handlers immediately receive them, drastically reducing the cost of communication.

Comparing $\lambda_{æ}$ to the Actor model, we see that the run M processes evolve in their own bubbles, and only communicate with other processes via signals and interrupts, similarly to actors. However, in contrast to messages not being required to be ordered in the Actor model, in our parallel composition operation $P \,||\, Q$, the process Q receives interrupts in the same order as the respective signals are issued by P (and vice versa). This communication ordering could be relaxed by allowing signals to be hoisted out of computations from deeper than just the top level, or by extending the operational semantics of $\lambda_{æ}$ with commutativity rules for signals. Another difference with actors is that by default $\lambda_{æ}$-computations react to interrupts sequentially—this difference can be remedied by writing programs in a style in which interrupts are handled in parallel in dynamically spawned dedicated processes.

It is worth noting that our interrupt handlers are similar to the message receiving construct in the π-calculus, in that they both synchronise with matching incoming interrupts or messages. However, the two constructs are also different, in that interrupt handlers allow reductions to take place under them and non-matching interrupts to propagate past them. Further, our interrupt handlers are also similar to join definitions in the join-calculus, describing how to react when a corresponding interrupt arrives or join pattern appears, where

in both cases the reaction could involve effectful code. To this end, our interrupt handlers resemble join definitions with simple one-channel join patterns. However, where the two constructs differ is that join definitions additionally serve to define new (local) channels, similarly to the restriction operator in the π-calculus, whereas we assume a fixed global set of channels (i.e., signal and interrupt names op). We expect that extending $\lambda_{æ}$ with local algebraic effects [Sta13, BPPS19] could help us fill this gap between the formalisms.

Scoped Operations. As noted in Section 3.2, despite their name, interrupt handlers behave like algebraic operations, not like effect handlers. However, one should also note that they are not conventional operations as they carry computational data that sequential composition does not interact with, and that executes only when a corresponding interrupt is received.

Such generalised operations are known in the literature as *scoped operations* [PSWJ18], a leading example of which is $\mathsf{spawn}\ (M, N)$. Further recalling Section 3.2, despite their appearance, incoming interrupts behave computationally like effect handling, not like algebraic operations. In fact, it turns out they correspond to effect handling induced by an instance of *scoped effect handlers* [PSWJ18]. Compared to ordinary effect handlers, scoped effect handlers explain both how to interpret operations and their scopes. In our setting, this corresponds to triggering interrupt handlers and executing the corresponding handler code.

It would be interesting to extend $\lambda_{æ}$ both with scoped operations having more general signatures, and with effect handlers for them, e.g., to allow preventing the propagation of incoming interrupts into continuations, discarding the continuation of a cancelled remote call, and techniques such as masking or reordering interrupts according to priority levels.

Denotational Semantics. In this paper we study only the operational side of $\lambda_{æ}$, and leave developing its denotational semantics for the future. In light of how we have motivated the $\lambda_{æ}$-specific programming constructs, and based on the above discussion, we expect the denotational semantics to take the form of an algebraically natural *monadic semantics*, where the monad would be given by an instance of the one studied in the case of scoped operations [PSWJ18] (quotiented by the commutativity of signals and interrupt handlers, and extended with nondeterminism to model different evaluation outcomes). Incoming interrupts would be modelled as homomorphisms induced by scoped algebras, while for parallel composition, we could consider all nondeterministic interleavings of (the outgoing signals of) individual computations, similarly to how it can be done in the context of general effect handlers [Plo12, LMM17]. Finally, we expect to be able to take inspiration for the denotational semantics of the promise type from that of modal logics and modal types.

Reasoning About Asynchronous Effects. In addition to using $\lambda_{æ}$'s type-and-effect system only for specification purposes (such as specifying that $M : X \,!\, (\emptyset, \{\})$ raises no signals and installs no interrupt handlers), we wish to make further use of it for validating *effect-dependent optimisations* [KP12]. For instance, whenever $M : X \,!\, (o, \iota)$ and $\iota\,(\mathsf{op}) = \bot$, we would like to know that $\downarrow \mathsf{op}\,(V, M) \rightsquigarrow^* M$. One way to validate such optimisations is to develop an adequate denotational semantics, and then use a semantic *computational induction* principle [BP14, PP08]. For $\lambda_{æ}$, this would amount to only having to prove the optimisations for return values, signals, and interrupt handlers. Another way to validate effect-dependent optimisations would be to define a suitable logical relation for $\lambda_{æ}$ [BHN14].

In addition to optimisations based on $\lambda_{æ}$'s existing effect system, we plan to refine the current "broadcast everything everywhere" communication strategy, e.g., by extending process

types with *communication protocols* inspired by session types [HVK98], or adding *restriction operations* like in CCS [Mil80] and suitably reflecting their use in the effect annotations.

Strong Normalisation for Computations. In addition to getting an overall more principled core calculus, one of the motivations for introducing reinstallable interrupt handlers to $\lambda_{æ}$ and removing general recursion (compared to our original work [AP21]) was that the sequential part of the resulting calculus ought to be strongly normalising, i.e., there should be no infinite reduction sequences for computations. Intuitively, strong normalisation should follow from interrupt handlers getting reinstalled only when a corresponding interrupt is propagated to the computation, and no single interrupt can reinstall a particular interrupt handler more than once. We leave making this argument formal for future work. We expect to be able to build on $\top\top$-lifting style logical relation proofs of strong normalisation [LS05].

However, even after making the above-mentioned changes to $\lambda_{æ}$, its parallel part of course remains non-terminating—simply consider two parallel processes built from reinstallable interrupt handlers that indefinitely exchange ping-pong signals with each other, e.g., as

```
run (↑ ping (); promise (pong _ r → ↑ ping (); r ()))
||
run (↑ pong (); promise (ping _ r → ↑ pong (); r ()))
```

Acknowledgements

We thank the anonymous reviewers, Otterlo IFIP WG 2.1 meeting participants, and Andrej Bauer, Gavin Bierman, Žiga Lukšič, Janez Radešček, and Alex Simpson for their useful feedback.

References

[AB20] D. Ahman and A. Bauer. Runners in action. In *Proc. of 29th European Symp. on Programming, ESOP 2020*, volume 12075 of *LNCS*, pages 29–55. Springer, 2020. doi:10.1007/978-3-030-44914-8_2.

[AC98] R. M. Amadio and P-L. Curien. *Domains and Lambda Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. doi:10.1017/CBO9780511983504.

[AFH+18] D. Ahman, C. Fournet, C. Hritcu, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL):65:1–65:30, 2018. doi:10.1145/3158153.

[Ahm24] D. Ahman. Agda formalisation of the $\lambda_{æ}$-calculus. Available at https://github.com/danelahman/higher-order-aeff-agda/releases/tag/lmcs, 2024.

[AP21] D. Ahman and M. Pretnar. Asynchronous effects. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021. doi:10.1145/3434305.

[BCJ+19] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20(1):973–978, January 2019.

[BGM19] P. Bahr, C. Graulund, and R. E. Mogelberg. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3(ICFP):109:1–109:27, 2019. doi:10.1145/3341713.

[BHN14] N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. In *Proc. of 41st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2014*, pages 619–632. ACM, 2014. doi:10.1145/2535838.2535869.

[BP14] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. doi:10.2168/LMCS-10(4:9)2014.

[BP15] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015. doi:10.1016/j.jlamp.2014.02.001.

[BPPS19] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Abstracting algebraic effects. *Proc. ACM Program. Lang.*, 3(POPL):6:1–6:28, 2019. doi:10.1145/3290319.

[CLMM20] L. Convent, S. Lindley, C. McBride, and C. McLaughlin. Doo bee doo bee doo. *J. Funct. Program.*, 30:e9, 2020. doi:10.1017/S0956796820000039.

[Clo18] R. Clouston. Fitch-style modal lambda calculi. In *Proc. of 21st Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2018*, volume 10803 of *LNCS*, pages 258–275. Springer, 2018. doi:10.1007/978-3-319-89366-2_14.

[DEH+17] S. Dolan, S. Eliopoulos, D. Hillerström, A. Madhavapeddy, K. C. Sivaramakrishnan, and L. White. Concurrent system programming with effect handlers. In *Revised Selected Papers from 18th Int. Symp. on Trends in Functional Programming - 18th International Symposium, TFP 2017*, volume 10788 of *Lecture Notes in Computer Science*, pages 98–117. Springer, 2017. doi:10.1007/978-3-319-89719-6_6.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. of 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'96*, pages 372–385. ACM, 1996. doi:10.1145/237721.237805.

[GHK+03] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Number 93 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. doi:10.1017/CBO9780511542725.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proc. of 3rd Int. Joint Conf. on Artificial Intelligence, IJCAI'73*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[HPM+20] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Scala documentation: Futures and promises. Available online at https://docs.scala-lang.org/overviews/core/futures.html, July 2020.

[HPP06] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006. doi:10.1016/J.TCS.2006.03.013.

[HVK98] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of 7th European Symp. on Programming, ESOP 1998*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

[KLO13] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

[KP12] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proc. of 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2012*, pages 349–360. ACM, 2012. doi:10.1145/2103656.2103698.

[Kri13] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013*, pages 221–232. ACM, 2013. doi:10.1145/2500365.2500588.

[Lei17] D. Leijen. Structured asynchrony with algebraic effects. In *Proc. of 2nd ACM SIGPLAN Int. Wksh. on Type-Driven Development, TyDe@ICF 2017*, pages 16–29. ACM, 2017. doi:10.1145/3122975.3122977.

[LMM17] S. Lindley, C. McBride, and C. McLaughlin. Do be do be do. In *Proc. of 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL 2017*, pages 500–514. ACM, 2017. doi:10.1145/3009837.3009897.

[LPT03] P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

[LS05] S. Lindley and I. Stark. Reducibility and $\top\top$-lifting for computation types. In *Proc. of 7th Int. Conf. of Typed Lambda Calculi and Applications, TLCA 2005*, volume 3461 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2005. doi:10.1007/11417170_20.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

[MJMR01] S. Marlow, S. L. Peyton Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *Proc. of of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285. ACM, 2001. doi:10.1145/378795.378858.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

[Mur08] T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2008.

[OCa] OCaml Development Team. OCaml 5.0.0 release notes. Available online at https://ocaml.org/releases/5.0.0.

[Plo12] G. D. Plotkin. Concurrency and the algebraic theory of effects. Invited talk at the 23rd Int. Conf. on Concurrency Theory, CONCUR 2012, 2012.

[Pou20] L. Poulson. Asynchronous effect handling. Master's thesis, School of Informatics, University of Edinburgh, 2020.

[PP02] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2002*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

[PP03] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

[PP08] G. D. Plotkin and M. Pretnar. A logic for algebraic effects. In *Proc. of 23th Ann. IEEE Symp. on Logic in Computer Science, LICS 2008*, pages 118–129. IEEE, 2008. doi:10.1109/LICS.2008.45.

[PP13] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4:23), 2013. doi:10.2168/LMCS-9(4:23)2013.

[Pre15] M. Pretnar. An introduction to algebraic effects and handlers. Invited tutorial paper. *Electr. Notes Theor. Comput. Sci.*, 319:19–35, 2015. doi:10.1016/j.entcs.2015.12.003.

[Pre24] M. Pretnar. Programming language ÆFF. Available at https://github.com/matijapretnar/aeff/releases/tag/lmcs, 2024.

[PSWJ18] M. Piróg, T. Schrijvers, N. Wu, and M. Jaskelioff. Syntax and semantics for operations with scopes. In *Proc. of 33rd Annual ACM/IEEE Symp. on Logic in Computer Science, LICS 2018*, pages 809–818. ACM, 2018. doi:10.1145/3209108.3209166.

[Rep93] J. H. Reppy. Concurrent ML: design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1993. doi:10.1007/3-540-56883-2_10.

[Sch02] J. Schwinghammer. A concurrent lambda-calculus with promises and futures. Master's thesis, Programming Systems Lab, Universität des Saarlandes, 2002.

[SDW+21] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *PLDI*, pages 206–221. ACM, 2021. doi:10.1145/3453483.3454039.

[Sta13] S. Staton. Instances of computational effects: An algebraic perspective. In *Proc. of 28th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2013*, pages 519–519. IEEE, 2013. doi:10.1109/LICS.2013.58.

[Sta15] S. Staton. Algebraic effects, linearity, and quantum programming languages. In *Proc. of 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2015*, pages 395–406. ACM, 2015. doi:10.1145/2676726.2676999.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

[WKS22] P. Wadler, W. Kokke, and J. G. Siek. *Programming Language Foundations in Agda*. August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

Appendix A. The Full Calculus for Higher-Order Asynchronous Effects

In this appendix we present $\lambda_{æ}$ with all the higher-order extensions discussed in Section 5.

A.1. **Terms.**

Values

V, W	$::= x$	variable
	$\mid () \mid (V, W)$	unit and pairing
	$\mid \mathsf{inl}_Y\, V \mid \mathsf{inr}_X\, V$	left and right injections
	$\mid \mathsf{fun}\, (x : X) \mapsto M$	function abstraction
	$\mid \langle V \rangle$	fulfilled promise
	$\mid [V]$	boxed value

Computations

M, N	$::= \mathsf{return}\, V$	returning a value
	$\mid \mathsf{let}\, x = M\, \mathsf{in}\, N$	sequencing
	$\mid V\, W$	function application
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{(x, y) \mapsto M\}$	product elimination
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{\}_{Z!(o,\iota)}$	empty elimination
	$\mid \mathsf{match}\, V\, \mathsf{with}\, \{\mathsf{inl}\, x \mapsto M, \mathsf{inr}\, y \mapsto N\}$	sum elimination
	$\mid \uparrow \mathsf{op}\, (V, M)$	outgoing signal
	$\mid \downarrow \mathsf{op}\, (V, M)$	incoming interrupt
	$\mid \mathsf{promise}\, (\mathsf{op}\, x\, r\, s \mapsto M)\, @_S\, V\, \mathsf{as}\, p\, \mathsf{in}\, N$	stateful reinstallable interrupt handler
	$\mid \mathsf{await}\, V\, \mathsf{until}\, \langle x \rangle\, \mathsf{in}\, M$	awaiting a promise to be fulfilled
	$\mid \mathsf{unbox}\, V\, \mathsf{as}\, [x]\, \mathsf{in}\, M$	unboxing a mobile value
	$\mid \mathsf{spawn}\, (M, N)$	dynamic process creation

Processes

P, Q	$::= \mathsf{run}\, M$	individual computation
	$\mid P \mid\mid Q$	parallel composition
	$\mid \uparrow \mathsf{op}\, (V, P)$	outgoing signal
	$\mid \downarrow \mathsf{op}\, (V, P)$	incoming interrupt

A.2. **Types.**

$$\text{Mobile type } A, B ::= \mathsf{b} \mid 1 \mid 0 \mid A \times B \mid A + B \mid [X]$$

$$\text{Signal or interrupt signature: } \mathsf{op} : A_{\mathsf{op}}$$

$$\text{Value type } X, Y, S ::= A \mid X \times Y \mid X + Y \mid X \to Y\, !\, (o, \iota) \mid \langle X \rangle$$

$$\text{Computation type: } X\, !\, (o, \iota)$$

$$\text{Process type } C, D ::= X\, !!\, (o, \iota) \mid C \mid\mid D$$

$$\text{Typing context } \Gamma ::= \cdot \mid \Gamma, x : X \mid \Gamma, 🔓$$

A.3. **Type System.**

Values

$$\textsc{TyVal-Var}\ \frac{X \text{ is mobile} \quad \text{or} \quad 🔓 \notin \Gamma'}{\Gamma, x : X, \Gamma' \vdash x : X} \qquad \textsc{TyVal-Unit}\ \frac{}{\Gamma \vdash () : 1} \qquad \textsc{TyVal-Pair}\ \frac{\Gamma \vdash V : X \qquad \Gamma \vdash W : Y}{\Gamma \vdash (V, W) : X \times Y}$$

$$\textsc{TyVal-Inl}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \mathsf{inl}_Y\, V : X + Y} \qquad \textsc{TyVal-Inr}\ \frac{\Gamma \vdash W : Y}{\Gamma \vdash \mathsf{inr}_X\, W : X + Y} \qquad \textsc{TyVal-Fun}\ \frac{\Gamma, x : X \vdash M : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{fun}\ (x : X) \mapsto M : X \to Y\,!\,(o, \iota)}$$

$$\textsc{TyVal-Promise}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \langle V \rangle : \langle X \rangle} \qquad \textsc{TyVal-Box}\ \frac{\Gamma, 🔓 \vdash V : X}{\Gamma \vdash [V] : [X]}$$

Computations

$$\textsc{TyComp-Return}\ \frac{\Gamma \vdash V : X}{\Gamma \vdash \mathsf{return}\ V : X\,!\,(o, \iota)} \qquad \textsc{TyComp-Let}\ \frac{\Gamma \vdash M : X\,!\,(o, \iota) \qquad \Gamma, x : X \vdash N : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : Y\,!\,(o, \iota)}$$

$$\textsc{TyComp-Apply}\ \frac{\Gamma \vdash V : X \to Y\,!\,(o, \iota) \qquad \Gamma \vdash W : X}{\Gamma \vdash V\, W : Y\,!\,(o, \iota)} \qquad \textsc{TyComp-MatchPair}\ \frac{\Gamma \vdash V : X \times Y \qquad \Gamma, x : X, y : Y \vdash M : Z\,!\,(o, \iota)}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{(x, y) \mapsto M\} : Z\,!\,(o, \iota)}$$

$$\textsc{TyComp-MatchEmpty}\ \frac{\Gamma \vdash V : 0}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\}_{Z!(o,\iota)} : Z\,!\,(o, \iota)} \qquad \textsc{TyComp-MatchSum}\ \frac{\Gamma \vdash V : X + Y \qquad \Gamma, x : X \vdash M : Z\,!\,(o, \iota) \qquad \Gamma, y : Y \vdash N : Z\,!\,(o, \iota)}{\Gamma \vdash \mathsf{match}\ V\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} : Z\,!\,(o, \iota)}$$

$$\textsc{TyComp-Signal}\ \frac{\mathsf{op} \in o \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X\,!\,(o, \iota)}{\Gamma \vdash \uparrow \mathsf{op}\,(V, M) : X\,!\,(o, \iota)} \qquad \textsc{TyComp-Interrupt}\ \frac{\Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash M : X\,!\,(o, \iota)}{\Gamma \vdash \downarrow \mathsf{op}\,(V, M) : X\,!\,\mathsf{op}{\downarrow}\,(o, \iota)}$$

$$\textsc{TyComp-ReStPromise}\ \frac{\begin{array}{c}(o', \iota') \sqsubseteq_{O \times I} \iota\,(\mathsf{op}) \qquad \Gamma, x : A_{\mathsf{op}}, r : S \to \langle X \rangle\,!\,\big(\emptyset, \{\mathsf{op} \mapsto (o', \iota')\}\big), s : S \vdash M : \langle X \rangle\,!\,(o', \iota') \\ \Gamma \vdash V : S \qquad \Gamma, p : \langle X \rangle \vdash N : Y\,!\,(o, \iota)\end{array}}{\Gamma \vdash \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N : Y\,!\,(o, \iota)}$$

$$\textsc{TyComp-Await}\ \frac{\Gamma \vdash V : \langle X \rangle \qquad \Gamma, x : X \vdash M : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{await}\ V\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M : Y\,!\,(o, \iota)} \qquad \textsc{TyComp-Unbox}\ \frac{\Gamma \vdash V : [X] \qquad \Gamma, x : X \vdash M : Y\,!\,(o, \iota)}{\Gamma \vdash \mathsf{unbox}\ V\ \mathsf{as}\ [x]\ \mathsf{in}\ M : Y\,!\,(o, \iota)}$$

$$\textsc{TyComp-Spawn}\ \frac{\Gamma, 🔓 \vdash M : X\,!\,(o, \iota) \qquad \Gamma \vdash N : Y\,!\,(o', \iota')}{\Gamma \vdash \mathsf{spawn}\ (M, N) : Y\,!\,(o', \iota')} \qquad \textsc{TyComp-Subsume}\ \frac{\Gamma \vdash M : X\,!\,(o, \iota) \qquad (o, \iota) \sqsubseteq_{O \times I} (o', \iota')}{\Gamma \vdash M : X\,!\,(o', \iota')}$$

Processes

$$\textsc{TyProc-Run}\ \frac{\Gamma \vdash M : X\,!\,(o,\iota)}{\Gamma \vdash \mathsf{run}\ M : X\,!!\,(o,\iota)} \qquad \textsc{TyProc-Par}\ \frac{\Gamma \vdash P : C \qquad \Gamma \vdash Q : D}{\Gamma \vdash P \,||\, Q : C \,||\, D}$$

$$\textsc{TyProc-Signal}\ \frac{\mathsf{op} \in \mathsf{signals\text{-}of}(C) \qquad \Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash P : C}{\Gamma \vdash \uparrow \mathsf{op}\,(V, P) : C} \qquad \textsc{TyProc-Interrupt}\ \frac{\Gamma \vdash V : A_{\mathsf{op}} \qquad \Gamma \vdash P : C}{\Gamma \vdash \downarrow \mathsf{op}\,(V, P) : \mathsf{op} \downarrow C}$$

A.4. Small-Step Operational Semantics of Computations.

Standard computation rules

$$(\mathsf{fun}\ (x : X) \mapsto M)\, V \rightsquigarrow M[V/x]$$

$$\mathsf{let}\ x = (\mathsf{return}\ V)\ \mathsf{in}\ N \rightsquigarrow N[V/x]$$

$$\mathsf{match}\ (V, W)\ \mathsf{with}\ \{(x, y) \mapsto M\} \rightsquigarrow M[V/x, W/y]$$

$$\mathsf{match}\ (\mathsf{inl}_Y\ V)\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} \rightsquigarrow M[V/x]$$

$$\mathsf{match}\ (\mathsf{inr}_X\ W)\ \mathsf{with}\ \{\mathsf{inl}\ x \mapsto M, \mathsf{inr}\ y \mapsto N\} \rightsquigarrow N[W/y]$$

Algebraicity of signals, interrupt handlers, awaiting, and process creation

$$\mathsf{let}\ x = (\uparrow \mathsf{op}\,(V, M))\ \mathsf{in}\ N \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{let}\ x = M\ \mathsf{in}\ N)$$

$$\mathsf{let}\ x = (\mathsf{promise}\ (\mathsf{op}\ y\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N_1)\ \mathsf{in}\ N_2 \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ y\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ (\mathsf{let}\ x = N_1\ \mathsf{in}\ N_2)$$

$$\mathsf{let}\ x = (\mathsf{await}\ V\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ M)\ \mathsf{in}\ N \rightsquigarrow \mathsf{await}\ V\ \mathsf{until}\ \langle y \rangle\ \mathsf{in}\ (\mathsf{let}\ x = M\ \mathsf{in}\ N)$$

$$\mathsf{let}\ x = (\mathsf{spawn}\ (M, N_1))\ \mathsf{in}\ N_2 \rightsquigarrow \mathsf{spawn}\ (M, \mathsf{let}\ x = N_1\ \mathsf{in}\ N_2)$$

Commutativity of signals and process creation with interrupt handlers

$$\mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\,(W, N) \rightsquigarrow \uparrow \mathsf{op}'\,(W, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N)$$

$$\mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M_1)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{spawn}\ (M_2, N) \rightsquigarrow \mathsf{spawn}\ (M_2, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M_1)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ N)$$

Interrupt propagation

$$\downarrow \mathsf{op}\,(V, \mathsf{return}\ W) \rightsquigarrow \mathsf{return}\ W$$

$$\downarrow \mathsf{op}\,(V, \uparrow \mathsf{op}'\,(W, M)) \rightsquigarrow \uparrow \mathsf{op}'\,(W, \downarrow \mathsf{op}\,(V, M))$$

$$\downarrow \mathsf{op}\,(V, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{let}\ p = M\big[V/x, R/r, W/s\big]\ \mathsf{in} \downarrow \mathsf{op}\,(V, N)$$

$$\textbf{where}\ R \stackrel{\text{def}}{=} \mathsf{fun}\ (s' : S) \mapsto \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ s'\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ p$$

$$\downarrow \mathsf{op}'\,(V, \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ W\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}'\,(V, N) \quad (\mathsf{op} \neq \mathsf{op}')$$

$$\downarrow \mathsf{op}\,(V, \mathsf{await}\ W\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M) \rightsquigarrow \mathsf{await}\ W\ \mathsf{until}\ \langle x \rangle\ \mathsf{in} \downarrow \mathsf{op}\,(V, M)$$

$$\downarrow \mathsf{op}\,(V, \mathsf{spawn}\ (M, N)) \rightsquigarrow \mathsf{spawn}\ (M, \downarrow \mathsf{op}\,(V, N))$$

Awaiting a promise to be fulfilled

$$\mathsf{await}\ \langle V \rangle\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M \rightsquigarrow M[V/x]$$

Unboxing a mobile value

$$\mathsf{unbox}\ [V]\ \mathsf{as}\ [x]\ \mathsf{in}\ M \rightsquigarrow M[V/x]$$

Evaluation context rule

$$\frac{M \rightsquigarrow N}{\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]}$$

where

$$\mathcal{E} ::= [\,] \mid \mathsf{let}\ x = \mathcal{E}\ \mathsf{in}\ N \mid \uparrow \mathsf{op}\,(V, \mathcal{E}) \mid \downarrow \mathsf{op}\,(V, \mathcal{E}) \mid \mathsf{promise}\ (\mathsf{op}\ x\, r\, s \mapsto M)\ @_S\ V\ \mathsf{as}\ p\ \mathsf{in}\ \mathcal{E} \mid \mathsf{spawn}\ (M, \mathcal{E})$$

A.5. Small-Step Operational Semantics of Processes.

Individual computations

$$\frac{M \rightsquigarrow N}{\mathsf{run}\ M \rightsquigarrow \mathsf{run}\ N}$$

Signal hoisting

$$\mathsf{run}\ (\uparrow \mathsf{op}\,(V, M)) \rightsquigarrow \uparrow \mathsf{op}\,(V, \mathsf{run}\ M)$$

Process creation

$$\mathsf{run}\ (\mathsf{spawn}\ (M, N)) \rightsquigarrow \mathsf{run}\ M \parallel \mathsf{run}\ N$$

Broadcasting

$$\uparrow \mathsf{op}\,(V, P) \parallel Q \rightsquigarrow \uparrow \mathsf{op}\,(V, P \parallel \downarrow \mathsf{op}\,(V, Q))$$
$$P \parallel \uparrow \mathsf{op}\,(V, Q) \rightsquigarrow \uparrow \mathsf{op}\,(V, \downarrow \mathsf{op}\,(V, P) \parallel Q)$$

Interrupt propagation

$$\downarrow \mathsf{op}\,(V, \mathsf{run}\ M) \rightsquigarrow \mathsf{run}\ (\downarrow \mathsf{op}\,(V, M))$$
$$\downarrow \mathsf{op}\,(V, P \parallel Q) \rightsquigarrow \downarrow \mathsf{op}\,(V, P) \parallel \downarrow \mathsf{op}\,(V, Q)$$
$$\downarrow \mathsf{op}\,(V, \uparrow \mathsf{op}'\,(W, P)) \rightsquigarrow \uparrow \mathsf{op}'\,(W, \downarrow \mathsf{op}\,(V, P))$$

Evaluation context rule

$$\frac{P \rightsquigarrow Q}{\mathcal{F}[P] \rightsquigarrow \mathcal{F}[Q]}$$

where

$$\mathcal{F} ::= [\,] \mid \mathcal{F} \parallel Q \mid P \parallel \mathcal{F} \mid \uparrow \mathsf{op}\,(V, \mathcal{F}) \mid \downarrow \mathsf{op}\,(V, \mathcal{F})$$

A.6. Process Type Reduction.

$$\frac{}{X \mathbin{!!} (o, \iota) \rightsquigarrow X \mathbin{!!} (o, \iota)} \qquad \frac{}{X \mathbin{!!} (\mathsf{ops} \downarrow\downarrow (o, \iota)) \rightsquigarrow X \mathbin{!!} (\mathsf{ops} \downarrow\downarrow (\mathsf{op} \downarrow (o, \iota)))} \qquad \frac{C \rightsquigarrow C' \qquad D \rightsquigarrow D'}{C \mathbin{||} D \rightsquigarrow C' \mathbin{||} D'}$$

$$\frac{}{X \mathbin{!!} (o, \iota) \rightsquigarrow (X \mathbin{!!} (o, \iota)) \mathbin{||} (Y \mathbin{!!} (o', \iota'))} \qquad \frac{}{Y \mathbin{!!} (o', \iota') \rightsquigarrow (X \mathbin{!!} (o, \iota)) \mathbin{||} (Y \mathbin{!!} (o', \iota'))}$$

A.7. Result Forms.

Computations

$$\frac{\mathsf{CompRes}\langle \Psi \mid M \rangle}{\mathsf{CompRes}\langle \Psi \mid \uparrow \mathsf{op}\, (V, M) \rangle} \qquad \frac{\mathsf{CompRes}\langle \Psi \mid N \rangle}{\mathsf{CompRes}\langle \Psi \mid \mathsf{spawn}\ (M, N) \rangle} \qquad \frac{\mathsf{RunRes}\langle \Psi \mid M \rangle}{\mathsf{CompRes}\langle \Psi \mid M \rangle}$$

$$\frac{}{\mathsf{RunRes}\langle \Psi \mid \mathsf{return}\ V \rangle} \qquad \frac{p \in \Psi}{\mathsf{RunRes}\langle \Psi \mid \mathsf{await}\ p\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ M \rangle}$$

$$\frac{\mathsf{RunRes}\langle \Psi \cup \{p\} \mid N \rangle}{\mathsf{RunRes}\langle \Psi \mid \mathsf{promise}\ (\mathsf{op}\ x\ r\ s \mapsto M)\ @\ V\ \mathsf{as}\ p\ \mathsf{in}\ N \rangle}$$

Processes

$$\frac{\mathsf{ProcRes}\langle P \rangle}{\mathsf{ProcRes}\langle \uparrow \mathsf{op}\, (V, P) \rangle} \qquad \frac{\mathsf{ParRes}\langle P \rangle}{\mathsf{ProcRes}\langle P \rangle} \qquad \frac{\mathsf{RunRes}\langle \emptyset \mid M \rangle}{\mathsf{ParRes}\langle \mathsf{run}\ M \rangle} \qquad \frac{\mathsf{ParRes}\langle P \rangle \qquad \mathsf{ParRes}\langle Q \rangle}{\mathsf{ParRes}\langle P \mathbin{||} Q \rangle}$$