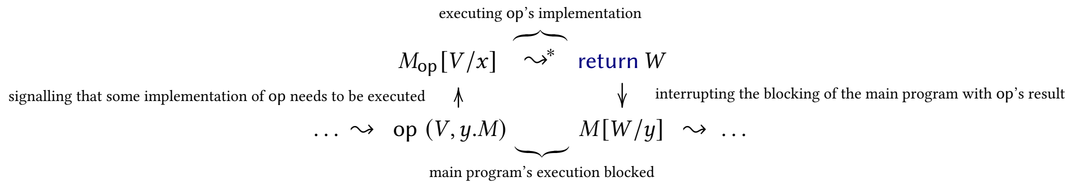
Higher-Order Asynchronous Effects (standard 30 min talk proposal)

DANEL AHMAN, MATIJA PRETNAR, and JANEZ RADEŠČEK, University of Ljubljana

A growing number of language designers and programmers have started to embrace *algebraic effects* (Plotkin and Power 2002) and *effect handlers* (Plotkin and Pretnar 2013) as a useful and flexible programming abstraction because they allow programmers to uniformly express a wide range of effectful behaviour, such as state, rollbacks, exceptions, and nondeterminism (Bauer and Pretnar 2015), concurrency (Dolan et al. 2018), statistical probabilistic programming (Bingham et al. 2019), and even quantum computation (Staton 2015). However, despite them covering many examples, the treatment of algebraic effects has remained *synchronous* in nature, meaning that when executing code with algebraic effects, an algebraic operation op's continuation is *blocked* until op is propagated to some implementation of it (e.g., an effect handler or a runner (Ahman and Bauer 2020)) and that implementation finishes executing. While such behaviour is necessary in some situations, it needlessly forces all uses of algebraic effects to be synchronous. To address this shortcoming, we recently showed how to also accommodate *asynchrony* within algebraic effects (Ahman and Pretnar 2021). In the first part of the talk, we will recall the key points of this work, before discussing some extensions to it to enable programmers to write more realistic asynchronous programs more naturally.

A brief recap of asynchronous effects

Our approach to asynchronous effects is based on decoupling of the execution of algebraic operation calls into three phases:



and observing that these phases can be separately captured by natural *non-blocking* programming abstractions, which we formalise in a core calculus for asynchronous effects, called $\lambda_{\text{æ}}$, accompanied by a prototype implementation available at https://github.com/matijapretnar/aeff/. We demonstrate the flexibility of $\lambda_{\text{æ}}$ using examples ranging from a multi-party web application, to preemptive multi-threading, to executing remote function calls, to implementing a parallel variant of runners.

Signals. In order to indicate that an operation's implementation needs to be executed, we allow our programs to issue *signals*, written $\uparrow \mathsf{op}\ (V, N)$. These are similar to algebraic operation calls in that once issued, they start propagating *outwards*.

Interrupts. The recipient of a signal (e.g., op's implementation) sees it as an *interrupt*, written $\downarrow \mathsf{op}\ (W, N)$. Importantly, while interrupts may again look like algebraic operation calls, then operationally they instead behave like *effect handling*, by propagating *inwards* into a computation, e.g., as $\downarrow \mathsf{op}\ (W, \mathsf{return}\ V) \rightsquigarrow \mathsf{return}\ V$ and $\downarrow \mathsf{op}\ (W, \uparrow \mathsf{op}'\ (V, N)) \rightsquigarrow \uparrow \mathsf{op}'\ (V, \downarrow \mathsf{op}\ (W, N))$.

Interrupt handlers. Programs can react to interrupts by installing *interrupt handlers*, written $\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$, where M is the code that gets executed when the interrupt handler is triggered (see reinstallable interrupt handlers below). The continuation N can refer to the result of handling the interrupt using the variable p bound in it. Importantly for the type safety of $\lambda_{\text{æ}}$, we do not assign p an arbitrary type, but a distinguished *promise type* $\langle X \rangle$, where X is the type of values provided by the handler. $\lambda_{\text{æ}}$'s type system then ensures that signal payloads cannot refer to these promise-typed variables, making it type safe to propagate them past handlers, as $\mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \uparrow \mathsf{op}'\ (V, N) \rightsquigarrow \uparrow \mathsf{op}'\ (V, \mathsf{promise}\ (\mathsf{op}\ x \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N)$. To extract the actual result from the promise-typed variable p, the continuation N can use the $\mathsf{await}\ p\ \mathsf{until}\ \langle x \rangle\ \mathsf{in}\ N'$ construct, which blocks the execution of N' until it is provided with a fulfilled promise $\langle V \rangle$ in place of p. Importantly, in contrast to await, *signals, interrupts, and interrupt handlers never block their continuations* N, providing the desired asynchrony.

Parallel processes. In order to model the environment of a program, $\lambda_{\text{æ}}$ also includes a very simple model of parallelism, given by *parallel processes* consisting of individual computations ($\mathsf{run}\ M$) and parallel compositions of processes ($P \mid\mid Q$), and also outwards propagating signals and inwards propagating interrupts. In contrast to many other process calculi, our parallel composition operation $\mid\mid$ does not perform any synchronisation, but instead it turns outwards propagating signals in one process into inwards propagating signals for all processes parallel to it, e.g., as $\uparrow \mathsf{op}\ (V, P) \mid\mid Q \rightsquigarrow \uparrow \mathsf{op}\ (V, P \mid\mid \downarrow \mathsf{op}\ (V, Q))$.

Now, while $\lambda_{\text{æ}}$ can indeed be used to capture a wide range of asynchronous examples (see above), it also has many notable limitations: server-like programs make excessive use of general recursion to define appropriate interrupt handlers; payloads

of signals and interrupts have to be ground values to ensure type safety; and it is not possible to dynamically create new parallel processes. In the second part of the talk, we will present our ongoing work on removing these limitations from $\lambda_{æ}$.

Reinstallable interrupt handlers

As noted above, many server-like programs written in $\lambda_{æ}$ end up making excessive use of general recursion, so as to reinstall interrupt handler(s) after the server has finished processing the client's request. Unfortunately, this results in programmers defining many auxiliary recursive 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 asynchrony.

To address the above issues, we propose extending $\lambda_{æ}$'s interrupt handlers with the ability to *reinstall* themselves, by allowing the handler code M to bind an additional variable k in $\mathsf{promise}\ (\mathsf{op}\ x\ k \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N$. In contrast to effect handlers, k does not refer to the continuation of the program (i.e., N), but instead to the act of reinstalling the given interrupt handler. To make this intuition more concrete, the triggering of these interrupt handlers is captured operationally as follows

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

and non-matching interrupts are simply propagated inwards past the interrupt handler as before, i.e., if $\mathsf{op} \neq \mathsf{op}'$, then $\downarrow \mathsf{op}'\ (W, \mathsf{promise}\ (\mathsf{op}\ x\ k \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N) \rightsquigarrow \mathsf{promise}\ (\mathsf{op}\ x\ k \mapsto M)\ \mathsf{as}\ p\ \mathsf{in} \downarrow \mathsf{op}'\ (W, N)$. Server-like processes can then be written more concisely, e.g., as $\mathsf{promise}\ \big(\mathsf{request}\ x\ k \mapsto \text{handle the request; issue a response signal;}\ k\ ()\big)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{return}\ ()$.

Higher-order payloads via Fitch-style modal types

In order to ensure that values which are meant to be mobile (e.g., payloads of signals) cannot refer to the promise-typed variables bound by interrupt handlers, $\lambda_{æ}$'s type system imposes a very strong syntactic restriction on the types of the payloads of signals and interrupts: they have to be ground types A, given by finite sums and products of base types. As a result, e.g., one can only send the arguments needed for the execution of remote function calls but not the functions themselves.

To address this limitation, we propose extending the payload types A with a Fitch-style *modal (box) type* $[X]$, together with its usual introduction and elimination rules (Clouston 2018), including a corresponding change to the typing rule for variables:

$$\textsc{Box-Intro}\ \frac{\Gamma, \blacksquare \vdash V : X}{\Gamma \vdash [V] : [X]} \qquad \textsc{Box-Elim}\ \frac{\Gamma \vdash V : [X] \qquad \Gamma, x : X \vdash M : Y}{\Gamma \vdash \mathsf{unbox}\ V\ \mathsf{as}\ [x]\ \mathsf{in}\ M : Y} \qquad \textsc{Var}\ \frac{X \text{ is mobile} \quad \text{or} \quad \blacksquare \notin \Gamma'}{\Gamma, x : X, \Gamma' \vdash x : X}$$

In this presentation of modal types, the tokens $\blacksquare$ limit which variables one is allowed to use from a program's context when introducing a boxed value $[V]$. In our setting, this results in programmers being able to use ground- and box-typed variables to construct $[V]$, but not promise-typed ones, capturing our intuition that $[X]$ is the type of (mobile) values of type X that are safe to be sent to other processes, and that promise-typed variables are to be only used for local intra-process synchronisation on the results of executing interrupt handlers. Importantly however, when constructing a (payload) value of type $[X \to Y]$, the boxed function can itself install additional interrupt handlers, it just cannot refer to the results of any enveloping ones.

Dynamic process creation via Fitch-style modal types

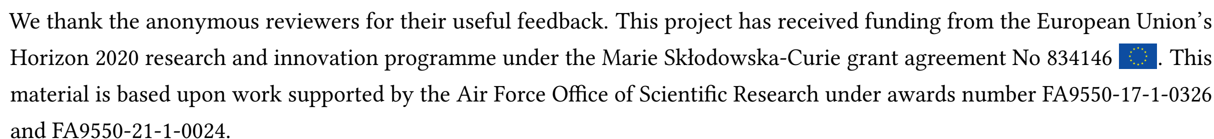
Having extended $\lambda_{æ}$ with modal types to enable higher-order payloads to be sent along with signals and interrupts, it turns out the same mechanism can be reused to extend $\lambda_{æ}$ also with *dynamic process creation*, written $\mathsf{spawn}\ (M, N)$, and typed as

$$\textsc{Spawn}\ \frac{\Gamma, \blacksquare \vdash M : 1 \qquad \Gamma \vdash N : Y}{\Gamma \vdash \mathsf{spawn}\ (M, N) : Y}$$

where M is the new process to be spawned, and N is the continuation of the existing program. Here, modal typing is used to ensure that M cannot refer to any enveloping interrupt handlers, making it safe to propagate it outwards past them, as $\mathsf{promise}\ (\mathsf{op}\ x\ k \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ \mathsf{spawn}\ (N_1, N_2) \rightsquigarrow \mathsf{spawn}\ (N_1, \mathsf{promise}\ (\mathsf{op}\ x\ k \mapsto M)\ \mathsf{as}\ p\ \mathsf{in}\ N_2)$, and eventually to the top-level of an individual computation, where M becomes a new parallel process, as $\mathsf{run}\ (\mathsf{spawn}\ (M, N)) \rightsquigarrow \mathsf{run}\ M \mid\mid \mathsf{run}\ N$.

Using these last two extensions to $\lambda_{æ}$, it becomes possible to generalise our remote function calls example to allow clients to upload their own functions for remote execution, and to create a new process for each such function or its execution.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their useful feedback. 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.

REFERENCES

D. Ahman and A. Bauer. 2020. Runners in action. In *Proc. of 29th European Symp. on Programming, ESOP 2020 (LNCS, Vol. 12075)*. Springer, 29–55.

D. Ahman and M. Pretnar. 2021. Asynchronous Effects. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.

A. Bauer and M. Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20, 1 (Jan. 2019), 973–978.

R. Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Proc. of 21st Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2018 (LNCS, Vol. 10803)*. Springer, 258–275.

S. Dolan, S. Eliopoulos, D. Hillerström, A. Madhavapeddy, K. C. Sivaramakrishnan, and L. White. 2018. Concurrent System Programming with Effect Handlers. In *Proc. of 18th Int. Sym. Trends in Functional Programming, TFP 2017*. Springer, 98–117.

G. D. Plotkin and J. Power. 2002. Notions of Computation Determine Monads. In *Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2002 (LNCS, Vol. 2303)*. Springer, 342–356.

G. D. Plotkin and M. Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4:23 (2013).

S. Staton. 2015. Algebraic Effects, Linearity, and Quantum Programming Languages. In *Proc. of 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2015*. ACM, 395–406.