Report from Dagstuhl Seminar 21292

Scalable Handling of Effects

Edited by

Danel Ahman[1], Amal Ahmed[2], Sam Lindley[3], and Andreas Rossberg[4]

1 **University of Ljubljana, SI,** danel.ahman@fmf.uni-lj.si
2 **Northeastern University – Boston, US,** amal@ccs.neu.edu
3 **University of Edinburgh, GB,** sam.lindley@ed.ac.uk
4 **Dfinity – Zürich, CH,** rossberg@mpi-sws.org

Abstract

Built on solid mathematical foundations, effect handlers offer a uniform and elegant approach to programming with user-defined computational effects. They subsume many widely used programming concepts and abstractions, such as actors, async/await, backtracking, coroutines, generators/iterators, and probabilistic programming. As such, they allow language implementers to target a single implementation of effect handlers, freeing language implementers from having to maintain separate ad hoc implementations of each of the features listed above.

Due to their wide applicability, effect handlers are enjoying growing interest in academia and industry. For instance, several effect handler oriented research languages are under active development (such as Eff, Frank, and Koka), as are effect handler libraries for mainstream languages (such as C and Java), effect handlers are seeing increasing use in probabilistic programming tools (such as Uber's Pyro), and proposals are in the pipeline to include them natively in low-level languages (such as WebAssembly). Effect handlers are also a key part of Multicore OCaml, which incorporates an efficient implementation of them for uniformly expressing user-definable concurrency models in the language.

However, enabling effect handlers to scale requires tackling some hard problems, both in theory and in practice. Inspired by experience of developing, programming with, and reasoning about effect handlers in practice, we identify five key problem areas to be addressed at this Dagstuhl Seminar in order to enable effect handlers to scale: Safety, Modularity, Interoperability, Legibility, and Efficiency. In particular, we seek answers to the following questions:

- How can we enforce safe interaction between effect handler programs and external resources?
- How can we enable modular use of effect handlers for programming in the large?
- How can we support interoperable effect handler programs written in different languages?
- How can we write legible effect handler programs in a style accessible to mainstream programmers?
- How can we generate efficient code from effect handler programs?

Seminar July 18–23, 2021 – http://www.dagstuhl.de/21292
2012 ACM Subject Classification Theory of computation → Control primitives; Theory of computation → Program semantics
Keywords and phrases continuations, Effect handlers, Wasm
Digital Object Identifier 10.4230/DagRep.11.7.54
Edited in cooperation with Hillerström, Daniel

1 Executive Summary

Danel Ahman (University of Ljubljana, SI)
Amal Ahmed (Northeastern University – Boston, US)
Sam Lindley (University of Edinburgh, GB)
Andreas Rossberg (Dfinity – Zürich, CH)

Algebraic effects and effect handlers are currently enjoying significant interest in both academia and industry as a modular programming abstraction for expressing and incorporating user-defined computational effects in programming languages. For example, there are a number of effect handler oriented languages in development (such as Eff, Frank, and Koka); there exist effect handler libraries for mainstream languages (such as C and Java); effect handlers are a key part of languages such as Multicore OCaml (and indeed they are due to appear in the production release of OCaml next year); effect handlers are being increasingly used in statistical probabilistic programming (such as Uber's Pyro tool); and proposals are in the works to include effect handlers in new low-level languages (such as WebAssembly). While effect handlers have solid mathematical foundations and have been extensively experimented in prototype languages and on smaller examples, enabling effect handlers to scale still requires tackling some hard problems. To this end, this Dagstuhl Seminar 21292 "Scalable Handling of Effects" focused on addressing the following key problem areas for scalability: Safety, Modularity, Interoperability, Legibility, and Efficiency.

This seminar followed the earlier successful Dagstuhl Seminars 16112 "From Theory to Practice of Algebraic Effects and Handlers" and 18172 "Algebraic Effect Handlers go Mainstream", which were respectively dedicated to the foundations of algebraic effects and to the introduction of them into mainstream languages. In contrast to these previous two seminars which took place in person at Schloss Dagstuhl, the current seminar was organised fully online due to the SARS-CoV-2 pandemic. As the seminar was attended by participants from a wide range of time zones (ranging from the West coast of the US all the way to Japan), coming up with a schedule that was suitable for everybody was a challenge. In the end, we decided to have three scheduled two-hour sessions each day, with impromptu informal discussions also happening between-times. These sessions were: (i) 15:00-17:00 CEST, which were deemed the Core Hours, where all participants were most likely to be able to present; (ii) 10:00-12:00 CEST, which was most suitable for participants from Asia and Europe; and (iii) 17:30-19:30 CEST, which was most suitable for participants from America and Europe. The Core Hours included talks, breakouts, and discussions of interest to the widest audience, with more specialised talks and breakouts taking place in the other two daily scheduled blocks. Talks were recorded so that participants could catch up due to being in an incompatible time zone, then deleted at the end of the week.

In order to run a successful virtual Dagstuhl seminar we exploited several different technologies. For talks we used Zoom. For breakouts we used a combination of Zoom and Gather.town, and for asynchronous communication and further discussions we used Zulip. For scheduling purposes, we used the wiki page provided by Dagstuhl.

We collected initial lists of proposed talks and breakout topics before the seminar began using an online form. We extended these throughout the week. We scheduled talks and breakout groups daily depending on audience interest and the participant availability. While the first part of the week was dominated by talks, the second part of the week saw more emphasis on breakouts and discussions. During Friday's Core Hours, the leaders of each breakout group presented a short overview of the discussions and results (11 reports in total). Initially, we were a little unsure about how well breakout sessions would work in a virtual seminar, but as the week went on they became more and more popular and they seemed to go remarkably well. Initially, we mostly used Gather.town and its virtual whiteboards for the breakout sessions. Subsequently, we transitioned to mostly using Zoom breakout rooms (partly because some people had difficulty using Gather.town on their systems).

The seminar was a great success, particularly given the constraints of the virtual format.

There were vibrant discussions around multishot continuations. These are vital for exciting new applications such as probabilistic programming and automatic differentiation, but more research is needed on how to implement them safely and efficiently in different contexts. Flipping perspective, it was mooted that for certain applications, particularly those involving direct interaction with the external world, it might be worthwhile restricting attention to runners, which are even more constrained than effect handlers with singleshot continuations.

There were several discussions relating to usability of effect handlers. These resulted in proposals to design a lecture course on effect handlers and to write a book on how to design effectful programs.

A major area of interest instigated at a prior Dagstuhl Seminar (18172 "Algebraic Effect Handlers go Mainstream") is the addition of effect handlers to WebAssembly. A design is being actively worked on as part of the official WebAssembly development process. At the current seminar we worked out extensions to the existing proposal to accommodate named effect handlers and symmetric stack-switching, both of which promise more efficient execution.

An issue with many existing benchmarks for effect handlers is that they often require installing a range of experimental software and configuring it with just the right settings. In order to make it easier to compare systems and share experimental setups we created the effect handlers benchmarks suite – a repository of benchmarks and systems covering effects and handlers in various programming languages, based on Docker scripts that make it easy for anyone to run the benchmarks and adapt them for their own research. The repository is hosted on GitHub. Since the seminar, 5 systems have been added to the repository and it has been actively updated and maintained by different members of the community.

At the end of the week, there was strong interest among the participants to continue this successful seminar series and submit a proposal for another incarnation, hopefully possible to take place on site in about two years.

2 Table of Contents

3 Overview of Talks

3.1 (Higher-Order) Asynchronous Effects

Danel Ahman (University of Ljubljana, SI)

While covering a large body of examples, the operational treatment of algebraic effects has remained synchronous in nature, meaning that when executing code in a language with algebraic effects, an algebraic operation op's continuation is blocked until (i) op is propagated to some implementation of it (such as an effect handler, a runner [1], or some top-level default implementation), (ii) that implementation finishes executing, and (iii) the original program is interrupted with the implementation's result. In this talk, I gave an overview of our work on accommodating asynchrony within algebraic effects based on observing that the different phases (i)–(iii) of an algebraic operation's execution can be split into separate programming abstractions.

The first half of the talk was based on our recent paper [2]. In this part of the talk, I showed how the different phases (i)–(iii) can be captured in a core λ-calculus for asynchrony using the following programming abstractions:

- *signals*, which programmers can issue to indicate that some operation's implementation needs to be executed, and that behave operationally like algebraic operations (in that they propagate outwards);
- *interrupts*, which are propagated to a program as a result of some other program issuing a corresponding signal, and that behave operationally like effect handling (in that they propagate inwards);
- *interrupt handlers*, which programmers can use to react to interrupts, and that (despite their name) behave like (scoped [4]) algebraic operations (in that they propagate outwards, just like signals); and
- *awaiting*, with which programmers can selectively block a program's execution by explicitly awaiting for one of the promises made by interrupt handlers to be fulfilled.

The resulting system achieves asynchrony by ensuring that signals, interrupts, and interrupt handlers never block the execution of their continuations (apart from when asked to do so by explicitly awaiting). In order to model a program's environment, such as the implementation of some algebraic operation, our core calculus also included a simple form of parallel processes. In the talk I also demonstrated the wide applicability of the proposed system: not only can we implement tail-resumptive algebraic operation calls, but we can also implement much more involved examples, such as (cancellable) remote function calls, multi-party web applications, non-blocking post-processing of promises, and preemptive multi-tasking.

In the second part of the talk, I presented our ongoing work on resolving the shortcomings we have since identified in our original system. These included: needing general recursion in the core calculus due to its heavy usage in examples; not being able to pass higher-order values in the payloads of signals and interrupts so as to ensure type safety; and not being able to dynamically spawn new parallel processes. First, in order to remove general recursion from the core calculus, we extended interrupt handlers to a notion of *reinstallable interrupt handlers*, in which the interrupt handler code is allowed to selectively reinstall the given interrupt handler, covering the uses of general recursion in our example programs. Next, in order to support higher-order signal and interrupt payloads in a type-safe manner, we extended our core calculus and the allowed payload types with a *Fitch-style modal* $\Box$*-type* [3], with which one can box up values of arbitrary types as payloads, while the type system guarantees that these values do not refer to any promise-typed binders in interrupt handlers (whose scope such payloads need to be able to escape). Finally, we also extended the core calculus with a programming abstraction for *spawning new parallel processes*, again using the technology involved in Fitch-style modal types to ensure type safety.

References

1 Ahman D., Bauer A. (2020) Runners in Action. In: Müller P. (eds) Programming Languages and Systems. ESOP 2020. Lecture Notes in Computer Science, vol 12075. Springer, Cham.
2 Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. Proc. ACM Program. Lang. 5, POPL, Article 24 (January 2021), 28 pages.
3 Clouston R. (2018) Fitch-Style Modal Lambda Calculi. In: Baier C., Dal Lago U. (eds) Foundations of Software Science and Computation Structures. FoSSaCS 2018. Lecture Notes in Computer Science, vol 10803. Springer, Cham.
4 Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18). Association for Computing Machinery, New York, NY, USA, 809–818.

3.2 The real world cannot be handled

Andrej Bauer (University of Ljubljana, SI)

Joint work of Danel Ahman, Andrej Bauer
Main reference Danel Ahman, Andrej Bauer: "Runners in Action", in Proc. of the Programming Languages and Systems – 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Lecture Notes in Computer Science, Vol. 12075, pp. 29–55, Springer, 2020.
URL https://doi.org/10.1007/978-3-030-44914-8_2

The language top level, or its runtime environment, is the interface to external resources, which are not subject to the usual rules of handlers or a monad in the language, and therefore should not be modeled as such. We should take the question "how to properly model and implement the runtime environment" seriously and apply available technology to develop modular and conceptually clean notion of "runtime environment". One such proposal was made by Danel Ahman and myself in our "Runners in action" paper. I would like to take the opportunity to discuss alternatives and to advertise the question as an interesting and important one.

3.3 Taming Higher-Order Control and State with Precise Effect Dependencies

Oliver Bracevac (Purdue University – West Lafayette, US)

This talk presents a novel ownership-style type system that tracks sets of term variables and assigns per-variable usage effects (e.g., reads, writes, kills) as determined by a user-defined effect quantale structure. For instance, through "kill effects," we support linear tracking of destructive updates to convert data structures between mutable and immutable accesses without copying. Compared to previous works on ownership, our system has a particularly lightweight type- and term-level footprint due subtyping and Scala/DOT-style abstract "self-aliases" to model escaping closures. Combining ownership-style reasoning and effects opens up interesting new avenues for the compilation of impure higher-order languages. Our type system gives rise to a novel typed graph IR that infers precise local effect dependencies, finally leading to affordable and aggressive global optimizations for impure higher-order programs. The graph IR is part of the newest version of the Scala LMS compiler framework and its optimizations enable significant speedups in real-world effectful programs.

3.4 Higher-order Programming with Effects and Handlers – with First-Class Functions

Jonathan Immanuel Brachthäuser (EPFL – Lausanne, CH)

Joint work of Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, Aleksander Boruch-Gruszecki
Main reference Jonathan Immanuel Brachthäuser, Philipp Schuster, Klaus Ostermann: "Effects as capabilities: effect handlers and lightweight effect polymorphism", Proc. ACM Program. Lang., Vol. 4(OOPSLA), pp. 126:1–126:30, 2020.
URL https://doi.org/10.1145/3428194

3.4.1 Introduction

Reasoning about the use of external resources is an important aspect of many practical applications. Examples range from memory management, controlling access to privileged resources like file handles or sockets, to analyzing the potential presence or absence of side effects.

3.4.1.1 Effect Systems encourage type-based reasoning

Effect systems extend the static guarantees of type systems to additionally track the use of effects [3]. They typically augment the type of functions with information about which effects the function might use. The fundamental idea of enhancing types with additional information is also one of the biggest problems of effect systems. Types quickly become verbose, difficult to understand, and difficult to reason about – especially in the presence of effect-polymorphic higher-order functions [7, 6, 1].

3.4.1.2 Capabilities encourage scope-based reasoning

Capabilities offer an alternative way to control the way resources are used. In this model, one can access resources and effects only through capabilities [4]. Thus, restricting access to capabilities restricts effects: a program can only perform effects of capabilities it can use. Some capabilities have a limited lifetime and should not leave a particular scope – for instance, if they are used to emulate checked exceptions. In these cases, treating capabilities as second-class values [5] provides such static guarantees. From a language designer's perspective, capabilities and second-class values offer an interesting alternative to effect systems: programmers can reason about effects the same way they reason about bindings. Additionally, second-class values admit a lightweight form of effect polymorphism without extending the language with effect variables or effect abstraction [1]. While lightweight, such systems severely restrict expressivity.

3.4.2 Comonadic type systems enable transitioning between type-based and scope-based reasoning

These systems allow programmers to reason about *purity* in an impure languages [2]. A special type constructor Safe witnesses the fact that its values are constructed without using any (impure) capabilities. Values of type Safe are introduced and eliminated with special language constructs. Importantly, explicit box introduction and elimination marks the transition between reasoning about effects by which capabilities are currently in scope, and reasoning about effects by types that witness the potential use of capabilities (that is, *impurity*). The type system presented by [2] only supports a binary distinction between *pure* values and *impure* values, which is not fine-grained enough for many practical applications – for instance, *effect masking*, or local handling of effects.

3.4.3 This Talk

In this talk, we draw inspiration from all three lines of research and present a calculus System C that aims at striking the balance between expressivity and simplicity. In particular, we combine and generalize the work by [5] and [2] to obtain a lightweight, capability-based alternative to effect systems. System C is based on the following design decisions:

3.4.3.1 Second-class values

Following [5], we distinguish between functions that can be treated as first-class values, and functions that are second-class. (To highlight this difference, we explicitly refer to second-class functions as *blocks*.) Thus, we avoid confronting programmers with the ceremony associated with tracking capabilities in types as much as possible. In particular, blocks can freely close over capabilities and effectful computations can simply use all capabilities in their lexical scope, with no visible type-level machinery to keep track of either fact.

3.4.3.2 Capability sets

Based on the work by [5] we annotate each binding in the typing context with additional information. However, we do not only track whether a bound variable is first- or second-class, but also track over which capabilities it closes. That is, we augment bindings (*e.g.*, $f :^{\mathcal{C}} \sigma$) in the typing context with *capability sets* (*e.g.*, $\mathcal{C}$). This information is annotated at the binder and is not part of the type. We will see that this is important for ergonomics as users are never confronted with this information. It is only necessary to check and guarantee effect safety.

3.4.3.3 Boxes

Blocks can freely close over other capabilities. However, they cannot be returned from a function or stored in a field. To recover these abilities we generalize the work by [2]: System C features explicit boxing and unboxing language constructs. Boxing converts a second-class value to a first-class value, reifying the contextual information annotated on the binder into the boxed value's type (*e.g.*, $f :^{\mathcal{C}} \sigma \vdash \mathbf{box}\, f : \sigma\, \mathbf{at}\, \mathcal{C}$). That is, instead of completely preventing first-class values from closing over capabilities, the capabilities they closed over are tracked in their types. To use a boxed block, we have to unbox it. We make sure to only perform this operation when the capabilities (*e.g.*, $\mathcal{C}$) are still in scope, which guarantees effect safety. The **box** and **unbox** constructs allow programmers to freely move between tracking capabilities implicitly, via scope, or explicitly, via type.

3.4.4 Discussion

In the talk, we will show how natural scope-based reasoning and precise type-based reasoning can co-exist in the same language and how programs can switch between them. We initially developed System C as a basis for adding first-class functions back to the Effekt language [1] – hence the title of this proposal. However, we believe that our system has broader applicability and we invite the participants to discuss the calculus, its limitations, and areas of application.

References

1 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (Nov. 2020).

2 Vikraman Choudhury and Neel Krishnaswami. 2020. Recovering Purity with Comonads and Capabilities. *Proc. ACM Program. Lang.* 4, ICFP, Article 111 (Aug. 2020).

3 J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proc. of the Symposium on Principles of Programming Languages* (POPL '88). ACM, New York, NY, USA, 47–57.

4 Mark Samuel Miller. 2006. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. Dissertation. *Johns Hopkins University*, Baltimore, Maryland, USA. AAI3245526.

5 Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, New York, NY, USA, 234–251.

6 Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proc. of the European Conference on Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–282.

7 Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *Proc. of the Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 281–295.

3.5 A Separation Logic for Effect Handlers

Paulo Emílio de Vilhena (INRIA – Paris, FR)

Joint work of Paulo Emílio de Vilhena, François Pottier
Main reference Paulo Emílio de Vilhena, François Pottier: "A separation logic for effect handlers", Proc. ACM Program. Lang., Vol. 5(POPL), pp. 1–28, 2021.
URL https://doi.org/10.1145/3434314

A program logic is a pair of a language, for writing the specification of a program, and a set of inference rules, for proving such specifications. In this talk, I present a program logic for a programming language with support for both effect handlers and higher-order state. I will begin with the motivation for this line of work – why is it interesting, or even useful, to conduct this research? I will then give an overview of the logic – how does it extend previous logics and what are its novel notions and main reasoning principles? Finally, I will present my vision for future research based upon this work: the verification of interesting applications of handlers, the design of extensions of the logic and its application to the study of effect systems.

3.6 Problems with resources and effects

Stephen Dolan (Jane Street – London, GB)

Two useful applications for effects are in controlling nondeterministic search (using continuations that are resumed multiple times), and organising programs that use asynchronous I/O (which manipulate stateful resources).

Problems arise when trying to do both in the same language: it is difficult to maintain guarantees of linearity and uniqueness in the presence of continuations that may resume more than once. I will present several tricky programs that mix these features, explain the problems they pose for the current crop of type systems, and leave their solution as a challenge for the audience.

3.7 Probabilistic Programming

Maria Gorinova (University of Edinburgh, GB)

Probabilistic programming aims to democratise Bayesian statistics and inference by providing a programming interface to the problem of probabilistic modelling. The user can specify their model, typically by describing the generative process of the data, and, in a perfect world, obtain inference results automatically. However, Bayesian inference is a challenging task, and it often needs to be tailored to the specific model in order to be efficient.

In this talk, I will discuss how effect handlers have been utilised to implement the backend of a few probabilistic programming languages, including Edward2 and Pyro. I will give several examples of common to probabilistic programming model transformation, which can be easily and compactly implemented using effect handlers. I will argue that such an effect-handling based backed provides the right set of abstractions for probabilistic programming users to be able to write and optimise model-specific and efficient inference strategies.

3.8 Composing UNIX with Effect Handlers

Daniel Hillerström (University of Edinburgh, GB)

3.8.1 Introduction

In functional programming effect handlers are often explained in terms of *folds* or *case-splits* over computational trees (depending on whether the handlers in question are *deep* or *shallow*) [7, 11, 12]. In imperative programming effect handlers are often explained as a slight operational extension of exception handlers endowed with the ability to resume exception-raising computations [10]. A compelling programming paradigm-agnostic way to explain effect handlers is to explain them as tiny composable operating systems, where we may view effectful operations as *system calls*, whose implementations are given by the *ambient environment*. In this analogy effect handlers play the role as the ambient environment. A richer ambient environment may be obtained by composing ever so more effect handlers. One can take this analogy quite literally, and use it to model the essence of an operating system such as UNIX by extending a feature-limited basis with more features by composing more handlers.

In the following sections we will demonstrate how to use effect handlers to model the essence of an UNIX-y operating system, which we shall call Tiny UNIX (the following sections are excerpts from my PhD dissertation [15]).

3.8.2 Basic i/o

The file system is a cornerstone of UNIX as the notion of *file* in UNIX provides a unified abstraction for storing text, interprocess communication, and access to devices such as terminals, printers, network, etc. We shall take a rather basic view of the file system. In fact, our system shall only contain a single file, and moreover, the system will only support writing operations. This system hardly qualifies as a UNIX file system. Nevertheless, it suffices to demonstrate the core idea, and it is not too difficult to grow it into a model of an actual file system [15]. The basic file system serves a crucial role for development of Tiny UNIX, because it provides the only means for us to be able to observe the effects of processes.

As in UNIX we shall model a file as a list of characters, i.e. $\mathsf{File} := \mathsf{List}\ \mathsf{Char}$. We will use the same model for strings, $\mathsf{String} := \mathsf{List}\ \mathsf{Char}$, such that we can use string literal notation to denote the `"contents of a file"`. The signature of the basic file system will consist of a single operation Write for writing a list of characters to the file.

$$\mathsf{BIO} := \{\mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String}\rangle \twoheadrightarrow 1\}$$

The operation is parameterised by a FileDescr and a character sequence. In this note, we will leave the details of FileDescr abstract as they are really only necessary when one considers a file system with multiple distinct files. We shall assume the existence of a term $\mathsf{stdout} : \mathsf{FileDescr}$ such that we can perform invocations of Write. Let us define a suitable handler for this operation.

$$\begin{aligned}
&\mathsf{basicIO} : (1 \to \alpha!\mathsf{BIO}) \to \langle \alpha; \mathsf{File}\rangle\\
&\mathsf{basicIO}\ m := \ \mathbf{handle}\ m\,\langle\rangle\ \mathbf{with}\\
&\qquad\qquad \mathbf{return}\ res \qquad\qquad\qquad\quad \mapsto \langle res; []\rangle\\
&\qquad\qquad \langle\mathsf{Write}\ \langle\_; cs\rangle \twoheadrightarrow resume\rangle \mapsto \mathbf{let}\ \langle res; file\rangle = resume\,\langle\rangle\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \langle res; cs +\!\!+ file\rangle
\end{aligned}$$

The handler takes as input a computation that produces some value α, and in doing so may perform the BIO effect. The handler ultimately returns a pair consisting of the return value α and the final state of the file. The **return**-case pairs the result res with the empty file $[]$ which models the scenario where the computation m performed no Write-operations, e.g. $\mathsf{basicIO}\,(\lambda\langle\rangle.\langle\rangle) \leadsto^{+} \langle\langle\rangle; \texttt{""}\rangle$. The Write-case extends the file by first invoking the resumption, whose return type is the same as the handler's return type, thus it returns a pair containing the result of m and the file state. The file gets extended with the character sequence cs before it is returned along with the original result of m. Intuitively, we may think of this implementation of Write as a peculiar instance of buffered writing, where the contents of the operation are committed to the file when the computation m finishes.

Let us define an auxiliary function that writes a string to the stdout file.

$$\begin{aligned}
&\mathsf{echo} : \mathsf{String} \to 1!\,\mathsf{BIO}\\
&\mathsf{echo}\ cs := \mathbf{do}\ \mathsf{Write}\,\langle \mathsf{stdout}; cs\rangle
\end{aligned}$$

The function echo is a simple wrapper around an invocation of Write. We can now write some contents to the file and observe the effects.

$$\begin{aligned} & \mathsf{basicIO}\,(\lambda\langle\rangle.\mathsf{echo}\ \texttt{"Hello"}; \mathsf{echo}\ \texttt{"World"}) \\ \rightsquigarrow^+ \ & \langle\langle\rangle; \texttt{"HelloWorld"}\rangle : \langle 1; \mathsf{File}\rangle \end{aligned}$$

3.8.3 Exceptions: non-local exits

A process may terminate successfully by running to completion, or it may terminate with success or failure in the middle of some computation by performing an *exit* system call. The exit system call is typically parameterised by an integer value intended to indicate whether the exit was due to success or failure. By convention, UNIX interprets the integer zero as success and any nonzero integer as failure, where the specific value is supposed to correspond to some known error code.

We can model the exit system call by way of a single operation Exit.

$$\mathsf{Status} := \{\mathsf{Exit} : \mathsf{Int} \twoheadrightarrow 0\}$$

The operation is parameterised by an integer value, however, an invocation of Exit can never return, because the type 0 is uninhabited. Thus Exit acts like an exception. It is convenient to abstract invocations of Exit to make it possible to invoke the operation in any context.

$$\begin{aligned} & \mathsf{exit} : \mathsf{Int} \to \alpha!\mathsf{Status} \\ & \mathsf{exit}\ n := \mathbf{absurd}\ (\mathbf{do}\ \mathsf{Exit}\ n) \end{aligned}$$

The **absurd** computation term is used to coerce the return type 0 of Exit into α. This coercion is safe, because 0 is an uninhabited type. An interpretation of Exit amounts to implementing an exception handler.

$$\begin{aligned} & \mathsf{status} : (1 \to \alpha!\mathsf{Status}) \to \mathsf{Int} \\ & \mathsf{status}\ m := \ \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with} \\ & \qquad\qquad\quad \mathbf{return}\ \_ \ \mapsto 0 \\ & \qquad\qquad\quad \langle\mathsf{Exit}\ n\rangle \quad \mapsto n \end{aligned}$$

Following the UNIX convention, the **return**-case interprets a successful completion of m as the integer 0. The operation case returns whatever payload the Exit operation was carrying. As a consequence, outside of status, an invocation of $\mathsf{Exit}\ 0$ in m is indistinguishable from m returning normally, e.g. $\mathsf{status}\,(\lambda\langle\rangle.\mathsf{exit}\ 0) = \mathsf{status}\,(\lambda\langle\rangle.\langle\rangle)$.

To illustrate status and exit in action consider the following example, where the computation gets terminated mid-way.

$$\begin{aligned} & \mathsf{basicIO}\,(\lambda\langle\rangle.\mathsf{status}\,(\lambda\langle\rangle.\mathsf{echo}\ \texttt{"dead"}; \mathsf{exit}\ 1; \mathsf{echo}\ \texttt{"code"})) \\ \rightsquigarrow^+ \ & \langle 1; \texttt{"dead"}\rangle : \langle \mathsf{Int}; \mathsf{File}\rangle \end{aligned}$$

The (delimited) continuation of $\mathsf{exit}\ 1$ is effectively dead code. Here, we have a choice as to how we compose the handlers. Swapping the order of handlers would cause the whole computation to return just $1 : \mathsf{Int}$, because the status handler discards the return value of its computation. Thus with the alternative layering of handlers the system would throw away the file state after the computation finishes. However, in this particular instance the semantics the (local) behaviour of the operations Write and Exit would be unaffected if the handlers were swapped. In general the behaviour of operations may be affected by the order of handlers. The canonical example of this phenomenon is the composition of nondeterminism and state, which we will discuss in Section 3.8.2.

3.8.4 Dynamic binding: user-specific environments

When a process is run in UNIX, the operating system makes available to the process a collection of name-value pairs called the *environment*. The name of a name-value pair is known as an *environment variable*. During execution the process may perform a system call to ask the operating system for the value of some environment variable. The value of environment variables may change throughout process execution, moreover, the value of some environment variables may vary according to which user asks the environment. For example, an environment may contain the environment variable `USER` that is bound to the name of the enquiring user.

An environment variable can be viewed as an instance of dynamic binding. It is well-known that dynamic binding can be encoded as a computational effect by using delimited control [6]. Unsurprisingly, we will use this insight to simulate user-specific environments using effect handlers.

For simplicity we fix the users of the operating system to be root, Alice, and Bob.

$$\mathsf{User} := [\mathsf{Alice}; \mathsf{Bob}; \mathsf{Root}]$$

Our environment will only support a single environment variable intended to store the name of the current user. The value of this variable can be accessed via an operation $\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}$. Using this operation we can readily implement the *whoami* utility from the GNU coreutils [14, Section 20.3], which returns the name of the current user.

$$\begin{array}{l}\mathsf{whoami} : 1 \to \mathsf{String}!\{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}\}\\ \mathsf{whoami}\ \langle\rangle := \mathbf{do}\ \mathsf{Ask}\ \langle\rangle\end{array}$$

The following handler implements the environment.

$$\begin{array}{l}\mathsf{env} : \langle \mathsf{User}; 1 \to \alpha!\{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}\}\rangle \to \alpha\\ \mathsf{env}\ \langle user; m\rangle := \ \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}\\ \qquad\qquad\qquad\quad \begin{array}{lll}\mathbf{return}\ res & \mapsto res & \\ \langle \mathsf{Ask}\ \langle\rangle \twoheadrightarrow resume\rangle & \mapsto \mathbf{case}\ user\ \{ & \mathsf{Alice} \mapsto resume\ \texttt{"alice"}\\ & & \mathsf{Bob}\ \ \mapsto resume\ \texttt{"bob"}\\ & & \mathsf{Root} \mapsto resume\ \texttt{"root"}\}\end{array}\end{array}$$

The handler takes as input the current *user* and a computation that may perform the Ask operation. When an invocation of Ask occurs the handler pattern matches on the *user* parameter and resumes with a string representation of the user. With this implementation we can interpret an application of whoami.

$$\mathsf{env}\ \langle \mathsf{Root}; \mathsf{whoami}\rangle \leadsto^{+} \texttt{"root"} : \mathsf{String}$$

It is not difficult to extend this basic environment model to support an arbitrary number of variables. This can be done by parameterising the Ask operation by some name representation (e.g. a string), which the environment handler can use to index into a list of string values. In case the name is unbound the environment, the handler can embrace the laissez-faire attitude of UNIX and resume with the empty string.

3.8.4.1 User session management

It is somewhat pointless to have multiple user-specific environments, if the system does not support some mechanism for user session handling, such as signing in as a different user. In UNIX the command *substitute user* (su) enables the invoker to impersonate another user account, provided the invoker has sufficient privileges. We will implement su as an operation $\mathsf{Su} : \mathsf{User} \twoheadrightarrow 1$ which is parameterised by the user to be impersonated. To model the security aspects of su, we will use the weakest possible security model: unconditional trust. Put differently, we will not bother with security at all to keep things relatively simple. Consequently, anyone can impersonate anyone else.

The session signature consists of two operations, Ask, which we used above, and Su, for switching user.

$$\mathsf{Session} := \{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}; \mathsf{Su} : \mathsf{User} \twoheadrightarrow 1\}$$

As usual, we define a small wrapper around invocations of Su.

```
$\mathsf{su} : \mathsf{User} \to 1!\{\mathsf{Su} : \mathsf{User} \twoheadrightarrow 1\}$
$\mathsf{su}\ \mathit{user} := \mathbf{do}\ \mathsf{Su}\ \mathit{user}$
```

The intended operational behaviour of an invocation of Su *user* is to load the environment belonging to *user* and continue the continuation under this environment. We can achieve this behaviour by defining a handler for Su that invokes the provided resumption under a fresh instance of the env handler.

```
$\mathsf{sessionmgr} : \langle \mathsf{User}; 1 \to \alpha!\mathsf{Session}\rangle \to \alpha$
$\mathsf{sessionmgr}\ \langle \mathit{user}; m\rangle := \mathsf{env}\langle \mathit{user}; (\lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}$
                                          $\mathbf{return}\ \mathit{res} \qquad\quad \mapsto \mathit{res}$
                                          $\langle\!|\mathsf{Su}\ \mathit{user}' \twoheadrightarrow \mathit{resume}|\!\rangle \mapsto \mathsf{env}\langle \mathit{user}'; \mathit{resume}\rangle)\rangle$
```

The function sessionmgr manages a user session. It takes two arguments: the initial user (*user*) and the computation (m) to run in the current session. An initial instance of env is installed with *user* as argument. The computation argument is a handler for Su enclosing the computation m. The Su-case installs a new instance of env, which is the environment belonging to user', and runs the resumption *resume* under this instance. The new instance of env shadows the initial instance, and therefore it will intercept and handle any subsequent invocations of Ask arising from running the resumption. A subsequent invocation of Su will install another environment instance, which will shadow both the previously installed instance and the initial instance.

To make this concrete, let us plug together the all components of our system we have defined thus far.

```
   basicIO (λ⟨⟩.
       sessionmgr ⟨Root; λ⟨⟩.
           status (λ⟨⟩.su Alice; echo (whoami ⟨⟩); echo " ";
                        su Bob;  echo (whoami ⟨⟩); echo " ";
                        su Root; echo (whoami ⟨⟩))⟩))
⇝⁺ ⟨0; "alice bob root"⟩ : ⟨Int; File⟩
```

The session manager (sessionmgr) is installed in between the basic IO handler (basicIO) and the process status handler (status). The initial user is Root, and thus the initial environment is the environment that belongs to the root user. Main computation signs in as Alice and writes the result of the system call whoami to the global file, and then repeats these steps for Bob and Root. Ultimately, the computation terminates successfully (as indicated by 0 in the first component of the result) with global file containing the three user names.

The above example demonstrates that we now have the basic building blocks to build a multi-user system.

3.8.5 Nondeterminism: time sharing

Time sharing is a mechanism that enables multiple processes to run concurrently, and hence, multiple users to work concurrently. Thus far in our system there is exactly one process. In UNIX there exists only a single process whilst the system is bootstrapping itself into operation. After bootstrapping is complete the system duplicates the initial process to start running user managed processes, which may duplicate themselves to create further processes. The process duplication primitive in UNIX is called *fork* [2]. The fork-invoking process is typically referred to as the parent process, whilst its clone is referred to as the child process. Following an invocation of fork, the parent process is provided with a nonzero identifier for the child process and the child process is provided with the zero identifier. This enables processes to determine their respective role in the parent-child relationship, e.g.

$$\begin{array}{l}\textbf{let}\ i \leftarrow \mathit{fork}\ \langle\rangle\ \textbf{in}\\ \textbf{if}\ i = 0\ \textbf{then}\ \ \mathit{child's\ code}\\ \textbf{else}\ \ \mathit{parent's\ code}\end{array}$$

In our system, we can model fork as an effectful operation, that returns a boolean to indicate the process role; by convention we will interpret the return value true to mean that the process assumes the role of parent.

$$\begin{array}{l}\mathsf{fork} : 1 \to \mathsf{Bool}!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}\\ \mathsf{fork}\ \langle\rangle := \textbf{do}\ \mathsf{Fork}\ \langle\rangle\end{array}$$

In UNIX the parent process *continues* execution after the fork point, and the child process *begins* its execution after the fork point. Thus, operationally, we may understand fork as returning twice to its invocation site. We can implement this behaviour by invoking the resumption arising from an invocation of Fork twice: first with true to continue the parent process, and subsequently with false to start the child process (or the other way around if we feel inclined). The following handler implements this behaviour.

$$\begin{array}{l}\mathsf{nondet} : (1 \to \alpha!\{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}) \to \mathsf{List}\ \alpha\\ \mathsf{nondet}\ m := \textbf{handle}\ m\ \langle\rangle\ \textbf{with}\\ \qquad\qquad \begin{array}{ll}\textbf{return}\ \mathit{res} & \mapsto [\mathit{res}]\\ \langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow \mathit{resume}\rangle & \mapsto \mathit{resume}\ \mathsf{true} \mathbin{+\!\!+} \mathit{resume}\ \mathsf{false}\end{array}\end{array}$$

The **return**-case returns a singleton list containing a result of running m. The Fork-case invokes the provided resumption *resume* twice. Each invocation of *resume* effectively copies m and runs each copy to completion. Each copy returns through the **return**-case, hence each invocation of *resume* returns a list of the possible results obtained by interpreting Fork first as true and subsequently as false. The results are joined by list concatenation ($\mathbin{+\!\!+}$). Thus the handler returns a list of all the possible results of m. (Remark: this handler is an instance of the standard backtracking nondeterminism handler from the literature, which has been used in related work to show that effect handlers endow their host language with additional asymptotic computational efficiency [13].)

Let us consider nondet together with the previously defined handlers. But first, let us define two computations.

$$\mathsf{ritchie}, \mathsf{hamlet} : 1 \to 1!\{\mathsf{Write} : \langle \mathsf{FileDescr}; \mathsf{String}\rangle \twoheadrightarrow 1\}$$

```
ritchie ⟨⟩ :=  echo "UNIX is basically ";
               echo "a simple operating system, ";
               echo "but ";
               echo "you have to be a genius
                     to understand the simplicity.\n"
hamlet ⟨⟩ := echo "To be, or not to be, ";
              echo "that is the question:\n";
              echo "Whether ’tis nobler in the mind to suffer\n"
```

The computation ritchie writes a quote by Dennis Ritchie to the file, whilst the computation hamlet writes a few lines of William Shakespeare's *The Tragedy of Hamlet, Prince of Denmark*, Act III, Scene I [1] to the file. Using nondet and fork together with the previously defined infrastructure, we can fork the initial process such that both of the above computations are run concurrently.

```
    basicIO (λ⟨⟩.
        nondet (λ⟨⟩.
            sessionmgr ⟨Root; λ⟨⟩.
                status (λ⟨⟩.if fork ⟨⟩ then su Alice; ritchie ⟨⟩
                                 else su Bob; hamlet ⟨⟩))))
⇝+ ⟨[0, 0]; "UNIX is basically a simple operating system, but
          you have to be a genius to understand the simplicity.\n
          To be, or not to be, that is the question:\n
          Whether 'tis nobler in the mind to suffer\n"⟩ : ⟨List Int; File⟩
```

The computation running under the status handler immediately performs an invocation of fork, causing nondet to explore both the **then**-branch and the **else**-branch. In the former, Alice signs in and quotes Ritchie, whilst in the latter Bob signs in and quotes a Hamlet. Looking at the output there is supposedly no interleaving of computation, since the individual writes have not been interleaved. From the stack of handlers, we *know* that there has been no interleaving of computation, because no handler in the stack handles interleaving. Thus, our system only supports time sharing in the extreme sense: we know from the nondet handler that every effect of the parent process will be performed and handled before the child process gets to run. In order to be able to share time properly amongst processes, we must be able to interrupt them.

3.8.5.1 Interleaving computation

We need an operation for interruptions and corresponding handler to handle interrupts in order for the system to support interleaving of processes.

```
interrupt : 1 → 1!{Interrupt : 1 ↠ 1}
interrupt ⟨⟩ := do Interrupt ⟨⟩
```

The intended behaviour of an invocation of Interrupt is to suspend the invoking computation in order to yield time for another computation to run. We can achieve this behaviour by reifying the process state. For the purpose of interleaving processes via interruptions it suffices to view a process as being in either of two states: 1) it is done, that is it has run to completion, or 2) it is paused, meaning it has yielded to provide room for another process to run. We can model the state using a recursive variant type parameterised by some return value α and a set of effects ε that the process may perform.

```
Pstate α ε θ := [Done : α;
                 Paused : 1 → Pstate α ε θ!{Interrupt : θ; ε}]
```

This data type definition is an instance of the *resumption monad* [3]. The Done-tag simply carries the return value of type α. The Paused-tag carries a suspended computation, which returns another instance of Pstate, and may or may not perform any further invocations of Interrupt. Payload type of Paused is precisely the type of a resumption originating from a handler that handles only the operation Interrupt such as the following handler.

```
reifyProcess : (1 → α!{Interrupt : 1 ↠ 1; ε}) → Pstate α ε
reifyProcess m := handle m ⟨⟩ with
                      return res              ↦ Done res
                      ⟨Interrupt ⟨⟩ ↠ resume⟩ ↦ Paused resume
```

This handler tags and returns values with Done. It also tags and returns the resumption provided by the Interrupt-case with Paused. This particular implementation is amounts to a handler-based variation of Harrison's [5] non-reactive resumption monad. If we compose this handler with the nondeterminism handler, then we obtain a term with the following type.

$$\mathsf{nondet}\,(\lambda\langle\rangle.\mathsf{reifyProcess}\ m) : \mathsf{List}\ (\mathsf{Pstate}\ \alpha\ \{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}; \varepsilon\})$$

for some $m : 1 \to \{\mathsf{Proc}; \varepsilon\}$ where $\mathsf{Proc} := \{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}; \mathsf{Interrupt} : 1 \twoheadrightarrow 1\}$. The composition yields a list of process states, some of which may be in suspended state. In particular, the suspended computations may have unhandled instances of Fork as signified by it being present in the effect row. The reason for this is that in the above composition when reifyProcess produces a Paused-tagged resumption, it immediately returns through the **return**-case of nondet, meaning that the resumption escapes the nondet. Recall that a resumption is a delimited continuation that captures the extent from the operation invocation up to and including the nearest enclosing suitable handler. In this particular instance, it means that the nondet handler is part of the extent. We ultimately want to return just a list of αs to ensure every process has run to completion. To achieve this, we need a function that keeps track of the state of every process, and in particular it must run each Paused-tagged computation under the nondet handler to produce another list of process state, which must be handled recursively.

```
schedule : List (Pstate $\alpha$ {Fork : Bool; $\varepsilon$} $\theta$) $\to$ List $\alpha!\varepsilon$
schedule $ps$ := let $run$ $\leftarrow$ rec $sched\,\langle ps; done\rangle$.
                          case $ps$ {                    [] $\mapsto$ $done$
                                      (Done $res$) :: $ps'$ $\mapsto$ $sched\,\langle ps'; res :: done\rangle$
                                     (Paused $m$) :: $ps'$ $\mapsto$ $sched\,\langle ps' +\!\!+ (\mathsf{nondet}\ m); done\rangle$}
                      in $run\,\langle ps; []\rangle$
```

The function schedule implements a process scheduler. It takes as input a list of process states, where Paused-tagged computations may perform the Fork operation. Locally it defines a recursive function *sched* which carries a list of active processes *ps* and the results of completed processes *done*. The function inspects the process list *ps* to test whether it is empty or nonempty. If it is empty it returns the list of results *done*. Otherwise, if the head is Done-tagged value, then the function is recursively invoked with tail of processes ps' and the list *done* augmented with the value *res*. If the head is a Paused-tagged computation m, then *sched* is recursively invoked with the process list ps' concatenated with the result of running m under the nondet handler.

Using the above machinery, we can define a function which adds time-sharing capabilities to the system.

```
timeshare : (1 $\to$ $\alpha$!Proc) $\to$ List $\alpha$
timeshare $m$ := schedule [Paused ($\lambda\langle\rangle$.reifyProcess $m$)]
```

The function timeshare handles the invocations of Fork and Interrupt in some computation m by starting it in suspended state under the reifyProcess handler. The schedule actually starts the computation, when it runs the computation under the nondet handler.

The question remains how to inject invocations of Interrupt such that computation gets interleaved. The interested reader may consult my dissertation for a discussion of different ways to inject interrupts, and for a more complete development of Tiny UNIX with file I/O, process synchronisation, programmable shell environment via shallow handlers, and more, as well as a discussion of ways to realise effect handlers, and hence, the operating system using canonical implementation techniques [8, 9, 11, 12, 15].

References

1 William Shakespeare. *The Tragedy of Hamlet, Prince of Denmark*. 1564-1616
2 Dennis Ritchie and Ken Thompson. *The UNIX Time-Sharing System*. Commun. ACM, 17, 1974
3 Nikolaos S. Papspyrou. *A resumption monad transformer and its applications in the semantics of concurrency* Proceedings of the 3rd Panhellenic Logic Symposium, Anogia, Greece, 2001
4 Eric Steven Raymond. *The Art of UNIX Programming*. ISBN 0131429019. Pearson Education, 2003
5 William L. Harrison. *The Essence of Multitasking*. AMAST, LNCS, 2006
6 Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. *Delimited dynamic binding*. ICFP, Portland, Oregon, USA, 2006
7 Ohad Kammar, Sam Lindley, and Nicolas Oury. *Handlers in action*. ICFP, Boston, Massachusetts, USA, 2013
8 Daniel Hillerström and Sam Lindley. *Liberating Effects with Rows and Handlers*. TyDe@ICFP, Nara, Japan, 2016
9 Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. *Continuation Passing Style for Effect Handlers*. FSCD, Oxford, UK, 2017
10 Daan Leijen. *Implementing Algebraic Effects in C – "Monads for Free in C"*. APLAS, Suzhou, China, 2017
11 Daniel Hillerström and Sam Lindley. *Shallow Effect Handlers*. APLAS, New Zealand, 2018
12 Daniel Hillerström, Sam Lindley, and Robert Atkey. *Effect Handlers via Generalised Continuations*. JFP (special issue on algebraic effects and handlers) 30:e5, 2020
13 Daniel Hillerström, Sam Lindley, and John Longley. *Effects for Efficiency: Asymptotic Speedup with First-Class Control*. ICFP, New Jersey, USA, 2020
14 David MacKenzie and others. *GNU Coreutils (for version 8.32)*. Free Software Foundation, 2020
15 Daniel Hillerström. *Foundations for Programming and Implementing Effect Handlers*. PhD thesis, The University of Edinburgh, UK, 2021

3.9 ParaFuzz: Fuzzing Multicore OCaml Programs

Sivaramakrishnan Krishnamoorthy Chandrasekaran (Indian Institute of Techology, IN)

Joint work of Sivaramakrishnan Krishnamoorthy Chandrasekaran, Sumit Padhiyar, Adharsh Kamath

Parallel programs are notoriously hard to test due to the particular combination of input and scheduling non-determinsm. Techniques such as property-based testing and fuzz testing are extremely effective for handling input non-determinsm. Crowbar is a tool for OCaml which combines property-based testing and fuzz testing for OCaml programs. Can we extend this to capture scheduling non-determinism? The answer is yes, and ParaFuzz shows how. A key challenge is getting control over the thread scheduling decisions. We should how effect handlers can help with this.

3.10 Retrofitting Effect Handlers onto OCaml

Sivaramakrishnan Krishnamoorthy Chandrasekaran (Indian Institute of Techology, IN)

Joint work of Krishnamoorthy Chandrasekaran Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, Anil Madhavapeddy
Main reference K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, Anil Madhavapeddy: "Retrofitting effect handlers onto OCaml", in Proc. of the PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211, pp. 206–221, ACM, 2021.
URL https://doi.org/10.1145/3453483.3454039

Multicore OCaml extends OCaml with support for effect handlers in order to express concurrency natively in direct-style. Given that we're extending an industrial-strength language with millions of lines of existing code, none of which is written with non-local control-flow in mind, our primary concern is backwards compatibility. Specifically, (a) not breaking legacy code, (b) retaining the performance profile of legacy code, and (c) debugging and profiling tool compatibility. In this talk, I shall discuss the backwards compatibility challenges and our solutions in Multicore OCaml.

3.11 Koka update: Compilation to C via generalized evidence passing and Perceus reference counting.

Daan Leijen (Microsoft Research – Redmond, US)

Koka can now compile effect handlers to standard C code; it uses a generalized evidence passing in combination with a multi-prompt delimited control monad to compile effect handlers (ICFP21). Moreover, it uses compile-time optimized reference counting (PLDI21) to manage memory without needing a GC or runtime system. I will show some of the new Koka language features, highlight the interesting parts of the compilation phases, and show various benchmarks.

3.12 Handler Calculus

Sam Lindley (University of Edinburgh, GB)

We present handler calculus, a core calculus of effect handlers. Inspired by the Frank programming language, handler calculus does not have primitive functions, just handlers. Functions, products, sums, and inductive types, are all encodable in handler calculus. We extend handler calculus with recursive effects, which we use to encode recursive data types. We extend handler calculus with parametric operations, which we use to encode existential data types. We then briefly outline how one can encode universal data types by composing a CPS translation for parametric handler calculus into System F with Fujita's CPS translation of System F into minimal existential logic.

3.13 Efficient Compilation of Algebraic Effect Handlers

Matija Pretnar (University of Ljubljana, SI)

Joint work of Georgios Karachalias, Filip Koprivec, Matija Pretnar, Tom Schrijvers
Main reference Georgios Karachalias, Filip Koprivec, Matija Pretnar, Tom Schrijvers: "Efficient Compilation of Algebraic Effect Handlers", Proc. ACM Program. Lang., Vol. 5(OOPSLA), Association for Computing Machinery, 2021.
URL https://doi.org/10.1145/3485479

The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code.

In the talk, I have presented our OOPSLA submission, which shows that the performance gap can be drastically narrowed (in some cases even closed) by means of type-and-effect directed optimising compilation. Our approach consists of source-to-source transformations in two phases of the compilation pipeline. Firstly, elementary rewrites, aided by judicious function specialisation, exploit the explicit type and effect information of the compiler's core language to aggressively reduce handler applications. Secondly, after erasing the effect information further rewrites in the backend of the compiler emit tight code.

This work comes with a practical implementation: an optimising compiler from Eff, an ML style language with algebraic effect handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks, our approach eliminates much of the overhead of handlers, outperforms capability-passing style compilation and yields competitive performance compared to hand-written OCaml code as well Multicore OCaml's dedicated runtime support.

3.14 Programming and Proving with Indexed effects in F*

Aseem Rastogi (Microsoft Research India – Bangalore, IN) and Nikhil Swamy (Microsoft Research – Redmond, US)

Joint work of Guido Martínez, Aymeric Fromherz, Tahina Ramananandro
Main reference Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, Nikhil Swamy: "Programming and Proving with Indexed Effects"
URL https://www.fstar-lang.org/papers/indexedeffects/

F* now supports a feature that allows programmers to define monadic effects with an arbitrary indexing structure. We have been using this to program and prove a variety of systems, using custom effect-typing disciplines, combining various prior approaches in novel ways. For example, we've been developing graded parameterized monads, or parameterized Dijkstra monads, graded Dijkstra monads, and parameterized-monad-indexed monads, and other seemingly exotic but very useful constructions. We've applied them to settings ranging from information flow control, to parsers, to separation logic, and to algebraic effects. The talk is intended to tell people about these structures, point out how one can program with them in F*, get feedback about it, and hopefully interest folks to develop new such structures, to use them in practice, and to study their semantics.

3.15 Low-level effect handlers for Wasm

Andreas Rossberg (Dfinity – Zürich, CH)

Joint work of Andreas Rossberg, Daniel Hillerström, Sam Lindley, KC Sivaramakrishnan, Matija Pretnar, Daan Leijen
URL https://github.com/effect-handlers/wasm-spec

I presented the ongoing work on a proposal for adding low-level effect handlers to Wasm.

3.16 Back to Direct Style 3

Philipp Schuster (Universität Tübingen, DE)

Joint work of Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, Klaus Ostermann

Programs in continuation-passing style are good to optimize but bad to run. We present a program transformation that goes from continuation-passing style back to direct style. It is a continuation of the "Back to Direct Style" line of work by Danvy and Lawall. We present a language with a type-and-effect system where it is possible to have multiple levels of control. Just like we can iterate the CPS transformation to make more and more levels of control explicit, we can iterate the direct-style transformation, to make more and more levels of control implicit. What we present is "work in progress" and we would like to discuss the approach in general, possible applications, and a logical interpretation with the audience.

3.17 CPS Transformation with Affine Types for Call-By-Value Implicit Polymorphism

Taro Sekiyama (National Institute of Informatics – Tokyo, JP)

Joint work of Taro Sekiyama, Tsukada, Takeshi
Main reference Taro Sekiyama, Takeshi Tsukada: "CPS transformation with affine types for call-by-value implicit polymorphism", Proc. ACM Program. Lang., Vol. 5(ICFP), pp. 1–30, 2021.
URL https://doi.org/10.1145/3473600

Transformation of programs into continuation-passing style (CPS) reveals the notion of continuations, enabling many applications such as control operators and intermediate representations in compilers. Although type preservation makes CPS transformation more beneficial, achieving type-preserving CPS transformation for implicit polymorphism with call-by-value (CBV) semantics is known to be challenging. We identify the difficulty in the problem that we call scope intrusion. To address this problem, we propose a new CPS target language Λ^{open} that supports two additional constructs for polymorphism: one only binds and the other only generalizes type variables. Unfortunately, their unrestricted use makes Λ^{open} unsafe due to undesired generalization of type variables. We thus equip Λ^{open} with affine types to allow only the type-safe generalization. We then define a CPS transformation from Curry-style CBV System F to type-safe Λ^{open} and prove that the transformation is meaning and type preserving. We also study parametricity of Λ^{open} as it is a fundamental property of polymorphic languages and plays a key role in applications of CPS transformation. To establish parametricity, we construct a parametric, step-indexed Kripke logical relation for λ^{open} and prove that it satisfies the Fundamental Property as well as soundness with respect to contextual equivalence.

3.18 Effects with Shifted Names in OCaml

Antal Spector-Zabusky (Jane Street – London, GB)

We are currently designing an effect system for OCaml consisting of algebraic effects with a fused "resume a continuation inside a handler" operation. We use shifted names to name effects, allowing operations that abstract over names to avoid shadowing names in the surrounding environment via renaming. This talk presents the design of the runtime semantics of this language as they currently stand.

3.19 Effects, Interface Types and async APIs

Luke Wagner (Fastly – San Francisco, US)

One focus of WASI right now is on HTTP APIs and supporting efficient request chaining via simple module linking/composition. Due to the streaming async nature of HTTP request handling, effects/coroutines are a natural fit. Expressing async APIs in a cross-language-compositional manner is challenging, though, when most of the constituent languages don't directly support algebraic effects. This talk discusses an idea we're working on for how to reconcile these constraints by building in a fixed 'async' effect to Interface Types that can be thought of as a specialized use of algebraic effects. When bound to JavaScript, Interface-Typed async functions would naturally bind to JavaScript async (i.e., Promise-returning) functions in a manner similar to the current wasm stack-switching JS API proposal.

4 Working groups

4.1 Control Operators Breakout Session

Jonathan Immanuel Brachthäuser (EPFL – Lausanne, CH), Youyou Cong (Tokyo Institute of Technology, JP), Sam Lindley (University of Edinburgh, GB), and Taro Sekiyama (National Institute of Informatics – Tokyo, JP)

In this breakout session, we explored the correspondence between effect handlers and delimited control operators from different perspectives. One question we discussed is what is the effect-handler-counterpart of shift/reset and control/prompt. These control operators keep the surrounding delimiter upon capture of a continuation, while effect handlers remove the surrounding handler upon a call of an operation. Sam suggested that such effect handlers may be useful for implementing the fork and yield operations, where we need a form of recursion, but we found that the control operators do not have enough expressive power.

Inspired by Sam's idea, we had a discussion on the correspondence between effect handlers and recursion schemes. Jonathan drafted a version of effect handlers that could correspond to histo-morphisms. In this sketch it appears histo-morphic effect handlers support a combination of the usual (deep) resumptions and shallow resumptions at each effect call.

We also talked about what is the control-operator counterpart of multi-handlers. Multi-handlers can handle multiple computations at once, which is difficult to express using shift/reset-style control operators. Daniel suggested that fcontrol/run may be easier to work with, and Youyou successfully implemented a "bi-handler" (handlers that can handle two computations) using these operators.

4.2 UX of Effect Systems Breakout Session

Jonathan Immanuel Brachthäuser (EPFL – Lausanne, CH), Youyou Cong (Tokyo Institute of Technology, JP), Paulo Emílio de Vilhena (INRIA – Paris, FR), and Filip Koprivec (University of Ljubljana, SI)

In this breakout session, we had a discussion on teaching effect systems. As a scenario where effect systems can be useful, Conor suggested building an OS, and April suggested developing GUIs (especially Web applications). As a tool for helping students understand effects, Youyou introduced an algebraic stepper developed at Ochanomizu University, and Matija introduced a similar tool supported in the aeff language. After the seminar, Nick, Jonathan, and Youyou had a meeting on the curriculum design of an effect handler course. There is also a plan to write a textbook called "How to Design Effectful Programs", which defines a series of design recipes for effect constructs.

4.3 Effect Handlers Benchmark Suite

Daniel Hillerström (University of Edinburgh, GB)

At the moment, a lot of work is about efficient runtime systems, or compilation, for effect handlers. However, as identified by this working group there is no standard benchmark suite for effect handler oriented programs. The literature makes use of a varying collection of ad-hoc benchmarks. This working group has begun the effort to create a community-maintained standardised benchmark suite for effect handler oriented programs. By standardised, we mean that the suite will contain a set of benchmarks intended to measure different aspects of effect handlers, e.g. single-shot, multi-shot, tail-resumptive handlers, etc, and each benchmark will have a description of its objective, how it should be realised (e.g. common implementation), and its parameters.

The benchmark suite is being actively developed on GitHub on the following repository.
`https://github.com/effect-handlers/effect-handlers-bench`

4.4 Wasm breakout session

Andreas Rossberg (Dfinity – Zürich, CH), Sam Lindley (University of Edinburgh, GB), and Luke Wagner (Fastly – San Francisco, US)

4.4.1 Introduction

The main purpose for adding effect handlers to WebAssembly is to provide a well-behaved mechanism for "stack switching". The proposal uses the asymmetric suspend/resume pair of primitives that is characteristic of handlers. This has been criticised for lacking a symmetric way of switching to another continuation directly, without going through a handler, and there is some concern that the double hop through a handler might involve unnecessary overhead for use cases like lightweight threading.

We discussed an idea, originally brought up by Luke Wagner, for extending the proposal with a more symmetric `switch_to` primitive. In fact, this can be broken down into two independent mechanisms:

1. Naming individual handlers, as a way of targeting them directly with a suspend, and thereby avoiding the linear search for a handler (somewhat similar to multi-prompt continuations).
2. A special built-in effect that switches to another continuation and is implicitly handled by every handler (or can be declared to be).

In addition, we discussed the possibility of first-class effect tags.

4.4.2 Named handlers

The idea here is to introduce a new reference type `(handler t*)`, which essentially is a unique prompt created by executing a variant of the resume instruction and is passed to the continuation:

```
cont.resume_from (event $tag $handler)* : [ t1* (cont $ft) ] -> [ t2* ]
where:
 -- $ft = [ (handler t2*) t1* ] -> [ t2* ]
```

The handler reference is similar to a prompt in a system of multi-prompt continuations. However, since its created fresh for each handler, multiple activations of the same prompt cannot exist by construction.

This instruction is complemented by an instruction for suspending to a specific handler:

```
cont.suspend_to $tag : [ t1* (handler t3*) ] -> [ t2* ]
where:
-- $tag : [ t1* ] -> [ t2* ]
```

If the handler is not currently active, e.g., because an outer handler has been suspended, then this instruction would trap.

We briefly pondered over the possibility of also an additional instruction to terminate a handler:

```
cont.return_to : [ t3* t1* (handler t1*) ] -> [ t2* ]
```

However, this would be like a throw, but without the ability to catch it. IT would therefore introduce yet another form of control flow transfer, whose interaction with other control operators (e.g., finally) would have to be considered. We concluded that it is preferable for the time being not to go there.

4.4.3 Direct switching

Given named handlers, it is possible to introduce a slightly more magic instruction for switching directly to another continuation:

```
cont.switch_to : [ t1* (cont $ft1) (handler t3*) ] -> [ t2* ]
where:
 -- $ft1 = [ (handler t3*) (cont $ft2) t1* ] -> [ t3* ]
 -- $ft2 = [ t2* ] -> [ t3* ]
```

This behaves as if there was a built-in tag

```
(tag Switch (param t1* (cont $ft1)) (result t3*))
```

with which the computation `suspends_to` the handler, and the handler implicitly handles this by `resuming_to` the continuation argument, thereby effectively switching to it in one step. Like `suspend_to`, this would trap if the handler wasn't currently active.

The fact that the handler implicitly resumes_to, passing itself as a handler to the target continuation, makes this construct behave like a deep handler, which is slightly add odds with the rest of the proposal.

In addition to the handler, `switch_to` also passed the new continuation to the target, which would allow the target to switch_to back to it in a symmetric fashion. Notably, in such a use case, `$ft1` and `$ft2` would be the same type (and hence recursive).

One observation we made is that symmetric switching is not necessarily tied to named handlers, since there could also be an indirect version with dynamic handler lookup:

```
cont.switch : [ t1* (cont $ft1) ] -> [ t2* ]
where:
-- $ft1 = [ (cont $ft2) t1* ] -> [ t3* ]
-- $ft2 = [ t2* ] -> [ t3* ]
```

Finally, it seems undesirable that every handler implicitly handles the built-in Switch tag, so this should be opt-in by a mode flag on the resume instruction(s).

4.4.4 First-class effect tags

We also discussed the possibility of having first-class effect tags. This would address a different but overlapping set of use cases compared to named handlers.

It would take the introduction of a new form of structured type, a tag type, and an instruction to generate fresh tags of such a type:

```
(type $tagtype (tag ...))

tag.new $tagtype : [] -> [(ref $tagtype)]
```

To be useful, though, this would require a new variant of resume instruction, whose handler table is created dynamically from its tag operands:

```
cont.resume (event $handler)* : [ (ref $tt)* t1* (cont $ft) ] -> [ t2* ]
where:
 -- ($tt = tag ...)*
 -- $ft = [ t1* ] -> [ t2* ]
```

Since the dispatch table has to be created dynamically at each execution of this instruction, it might be quite expensive in practice, especially since the handlers in the proposal behave like shallow handlers, i.e., must be recreated for every resumption. Also, this cannot easily be circumvented by adding first-class handlers, since the latter are made difficult because of the local nature of the branch labels handlers depend on. More investigation is needed.

4.5 Dependent types breakout session

Wouter Swierstra (Utrecht University, NL) and Robert Atkey (University of Strathclyde – Glasgow, GB)

In this session we discussed the various approaches to modelling effects and handlers accurately using rich types, typically involving some variation of monads such as parametrised monads, indexed monads, and graded monads. These can often be embedded in an existing programming language – but languages such as F^* add native support for collecting and resolving the proof obligations associated with certain effectful computations.

5 Open problems

5.1 Efficient stack layout for multishot handlers

Filip Koprivec (University of Ljubljana, SI)

Joint work of Filip Koprivec, Matija Pretnar

Much has been done on optimizing the performance of effect handlers, from an optimized runtime and evidence translation in Koka to a specialized stack structure in Multicore OCaml. We proposed an efficient stack management technique based on heap-allocated fibres by Sivaramakrishnan and others. We present a "work in progress" idea for a stack structure specialized for multiple resumptions.

The program stack is stored as a sequence of fibres corresponding either to computation or a handled effect. Once the computation is handled by an enclosing handler, the whole part of the stack corresponding to that computation is "frozen" and specifically marked for a copy on reuse when invoking the continuation. When the continuation is resumed before the frozen computation gets popped from the stack, there is no need to allocate any heap storage for the environment of continued computation as frozen fibre gets copied from the stack to the top of the stack directly.

This decreases the pressure on both allocator and garbage collector while reusing already allocated stack memory. Reuse of stack saved computations is faster than allocation on the heap and this especially improves performance when reusing the same continuation multiple times. The improvement an optimized stack structure offers is heavily dependent on handler usage and memory allocator performance.

Participants

- Danel Ahman
University of Ljubljana, SI
- Amal Ahmed
Northeastern University – Boston, US
- Robert Atkey
University of Strathclyde – Glasgow, GB
- Andrej Bauer
University of Ljubljana, SI
- Oliver Bracevac
Purdue University – West Lafayette, US
- Jonathan Immanuel Brachthäuser
EPFL – Lausanne, CH
- Youyou Cong
Tokyo Institute of Technology, JP
- Paulo Emílio de Vilhena
INRIA – Paris, FR
- Stephen Dolan
Jane Street – London, GB
- Ronald Garcia
University of British Columbia – Vancouver, CA
- April Gonçalves
Heliax – Glasgow, GB
- Maria Gorinova
University of Edinburgh, GB
- Daniel Hillerström
University of Edinburgh, GB
- Mauro Jaskelioff
National University of Rosario, AR
- Ohad Kammar
University of Edinburgh, GB
- Oleg Kiselyov
Tohoku University – Sendai, JP
- Filip Koprivec
University of Ljubljana, SI
- Sivaramakrishnan Krishnamoorthy Chandrasekaran
Indian Institute of Techology, IN
- Daan Leijen
Microsoft Research – Redmond, US
- Sam Lindley
University of Edinburgh, GB
- Conor McBride
University of Strathclyde – Glasgow, GB
- Daniel Patterson
Northeastern University – Boston, US
- Maciej Piróg
University of Wroclaw, PL
- Gordon Plotkin
Google – Mountain View, US
- Matija Pretnar
University of Ljubljana, SI
- Aseem Rastogi
Microsoft Research India – Bangalore, IN
- Andreas Rossberg
Dfinity – Zürich, CH
- Philipp Schuster
Universität Tübingen, DE
- Taro Sekiyama
National Institute of Informatics – Tokyo, JP
- Antal Spector-Zabusky
Jane Street – London, GB
- Nikhil Swamy
Microsoft Research – Redmond, US
- Wouter Swierstra
Utrecht University, NL
- Luke Wagner
Fastly – San Francisco, US
- Leo White
Jane Street – London, GB
- Nicolas Wu
Imperial College London, GB