ISSN 2186-7437

NII Shonan Meeting Report

No. 203

Effect Handlers and General Purpose Languages

Jonathan Brachthäuser
Youyou Cong
Jeremy Gibbons

September 25–29, 2023

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Effect Handlers and General Purpose Languages

Organizers:
Jonathan Brachthäuser (Universität Tübingen, DE)
Youyou Cong (Tokyo Institute of Technology, JP)
Jeremy Gibbons (University of Oxford, UK)

September 25–29, 2023

1 Background and introduction

Algebraic effects and handlers are a uniform abstraction for expressing computational effects. The abstraction enables modular development of programs involving sophisticated control-flow patterns, while (optionally) guaranteeing the absence of certain undesired behavior. Effect handlers have been implemented in diverse programming languages, ranging from functional languages such as Haskell and OCaml, to imperative languages such as C and JavaScript. In particular, the next release of OCaml will be the first production-level language with built-in support for effect handlers. There are also industrial applications of effect handlers. As a representative example, the Pyro programming language of Uber has a library of effect handlers, which allows the user to easily implement inference algorithms for probabilistic programming.

In response to the growing use of effect handlers, researchers have been actively studying the theory and implementation of effect handlers. Of particular interest are the reasoning, performance, and typing of effect handlers, which were the main topics of Shonan Meeting 146 on "Programming and Reasoning with Algebraic Effects and Effect Handlers" held in 2019.

At the same time, researchers have also been working on the practice of effect handlers. In particular, the goal of a prior Dagstuhl Seminar 18172 on "Algebraic Effect Handlers Go Mainstream" was to bring effect handlers to existing languages.

We believe that it is a good time to revisit the above theoretical challenges, but from a different perspective. Specifically, in addition to sharing new results about effect handlers, we hope to transfer the knowledge from research on effect handlers back to designing languages with specific built-in effects, such as exceptions and concurrency. In particular, we investigate how recent advances in the field of algebraic effects and handlers can improve the reasoning, performance, and typing of languages, even if they do not include effect handlers themselves.

Reasoning: Handlers and equations both serve as a means to specify the behavior of algebraic effects. The former are well-suited for everyday programming, while the latter are useful for formal reasoning. Researchers have been attempting to give an equational account of effect handlers, but little progress has been made on integrating equations into existing languages with effect handlers. In this meeting, we explore practical approaches to equational reasoning of effect handlers that can be used for optimization purposes. In particular, we improve on the recent approaches that use types and fusion. We also plan to discuss how to adapt the approaches to general-purpose languages with exceptions and concurrency. In particular, we expect that many reasoning techniques for effect handlers will scale straightforwardly to exceptions, as effect handlers are a generalization of exception handlers.

Performance: There are several sources of inefficiency in running programs with algebraic effects and handlers, such as the runtime search for handlers and the process of capturing continuations. The past few years have seen various techniques for reducing inefficiency, but each of them has certain limitations in terms of applicability and performance. In this meeting, we aim to find general and effective techniques for efficiently executing algebraic effects and handlers.

To achieve this goal, we reuse ideas from various program transformations proposed recently. We also intend to discuss how to adapt such techniques to general-purpose languages with primitive effects, such as exceptions and region-based memory management. For instance, we conjecture that the similarity between handler lifetimes and object lifetimes would help us derive optimizations for memory management from those for effect handlers.

Typing: Many languages providing algebraic effects and handlers are equipped with an effect system, which prevents undesired behavior during execution of programs. There are a variety of effect systems in the literature; some of them focus on expressiveness, while others prioritize ease of use. In this meeting, we seek a sweet spot among different aspects of effect systems. Specifically, we compare type systems that enforce lexical handling with those that enforce dynamic handling, and type systems that are explicit about effect polymorphism with those that are implicit. Based on the insights gained from effect handler studies, we are also interested in developing and improving effect systems for parallel, concurrent, and probabilistic programming languages. Specifically, we believe that the discussion of lexical versus dynamic handling and explicit versus implicit polymorphism would help us increase the efficiency and user experience of these languages.

Solving these problems has two potential outcomes. First, it fosters uses of effect handlers and thus increases modularity and safety of real-world programs. Second, it leads to better implementation of effectful languages in general. We believe that having face-to-face discussions among effect handler experts is crucial to advance in this research area.

From the above problems, we draw the following specific topics for discussion at the meeting:

- effect handlers and delimited control operators (shift/reset, control/prompt, fcontrol/run);
- effect handlers and morphisms (mutumorphisms, futumorphisms, histomorphisms);
- effect handlers and parallelism;
- effect handlers and quantitative types (linear types, affine types);
- lexical handlers versus dynamic handlers;
- user experience of effect systems and effect polymorphism.

2 Overview of the meeting

The first day of the meeting was structural. The first session consisted of a round of introductions: one slide per person, collected in advance for smooth presentation; three minutes for each to speak, including saying how they got into effects and handlers. The second and third sessions consisted of an extended tutorial on algebraic effects and handlers, in theory and in practice; this served to get all participants "on the same page". The final session was spent planning the rest of the week.

Each of the second, third, and fourth days was devoted to one aspect of effect handlers in general-purpose languages: matters of implementation, application areas, and programmer-facing issues. The middle one of these was only half a day, because of the excursion (to Hokokuji and Jomyoji) and banquet.

The second and fourth days each ended in a panel discussion, held in a fishbowl format: an initial selection of panellist fish in the bowl, but anyone else can speak, thereby becoming a new fish and evicting one of the current fish. The topics were "concurrency and distribution" and "will effect handlers replace monads?"; some notes are recorded in Section 5.

The final morning was spent in summary: discussing plans for a book, the next meeting in the series, a final presentation that had been bumped due to lack of time, and so on. The outcomes of this discussion are recorded in Section 6.

Check-in Day: September 24 (Sun)

- Welcome Banquet

Day1: September 25 (Mon)

- Session 1: Opening
- Session 2: Tutorial
- Session 3: Tutorial
- Session 4: Planning & Puzzles

Day2: September 26 (Tue)

- Session 5: Implementor-Facing Aspects
- Session 6: Implementor-Facing Aspects
- Group Photo Shooting
- Session 7: Current Status of Effect Handler Languages
- Session 8: Panel "Concurrency and Distribution"

Day3: September 27 (Wed)

- Session 9: Applications
- Session 10: Applications
- Excursion and Main Banquet

Day4: September 28 (Thu)

- Session 11: Programmer-Facing Aspects
- Session 12: Programmer-Facing Aspects
- Session 13: Programmer-Facing Aspects
- Session 14: Talk & Panel "Will Effect Handlers Replace Monads?"

Day5: September 29 (Fri)

- Session 15: Summary, Book, Next Meeting
- Session 16: Talk

3 Abstracts of talks

Introduction to Algebraic Effects and Effect Handlers

Sam Lindley, The University of Edinburgh

I'll give an introduction to algebraic effects and effect handlers as a general approach to programming and reasoning about effectful computation. I'll present the notion of a computation over an algebraic effect as a command-response tree over an effect signature quotiented by some equational theory. I'll consider how to interpret command-response trees and motivate effect handlers as the reification of such interpretations as an object language feature that provides a generic implementation strategy for algebraic effects. I'll give examples to show that it can be useful to interpret the same command-response tree using different interpretations which may not respect the same equational theory. Thus effect handlers can provide an expressive programming feature independently of any non-trivial algebraic theory.

Regions in Effekt

Philipp Schuster, University of Tübingen

We demonstrate a feature of Effekt, which is uncommon in other languages with effect handlers: regions. We start with an introduction to and discussion of the basic features and design decisions in Effekt. The complete lack of return clauses in handlers is rather controversial. We then motivate the need for mutable references to live in regions with the difference in backtracking behavior in presence of multiple resumptions. Finally, we explain how Effekt maintains region safety when users abstract over and return regions.

Operational Semantics of Effects and Handlers

Jonathan Brachthäuser, University of Tübingen

In this tutorial, we look at the different ways of giving operational semantics to effects and handlers. We start with a calculus with effects and handlers and introduce fine-grain call-by-value.

We then show a first operational semantics, sometimes referred to as *bubble semantics*: effect operations also have an additional continuation argument `do` *op e* (*x. s*). Reducing an effect operation proceeds by growing this continuation argument bottom-up ("bubbling") collecting all frames and unrelated handlers on the way. The most essential rule is that of *algebraicity* which pushes the immediate context of a let binding into the continuation.

Next, we show a dual approach which could be called *sinking semantics*. Instead of growing the continuation bottom-up, handlers are operationally pushed down and aggregate computation in their return clause. When the handlers sink to an effect operation, the continuation is available as part of the return clause of the handler.

We then look at a third semantics making use of evaluation contexts. This semantics is more coarse grained as it captures the whole context between an effect operation and a handler in one step.

Furthermore, we see how to describe the semantics of effects and handlers in terms of abstract machines. A machine state is a pair of a computation (or statement) and a stack. To capture the continuation, the machine transitions into a new state, represented by a triple, which the last component is the captured continuations. We discuss how, depending on the concrete choice of representation of the stack and the continuation, continuation capture can either be linear in the number of frames or linear in the number of effect handlers.

Finally, together with the attendees of the seminar, we collect several aspects of implementing effects and handlers, resulting in a mind map.

Search Combinators: Now with Algebraic Effects and Handlers

Tom Schrijvers, KU Leuven

In this talk I look back on my earlier work on modelling Constraint Programming and modular Search Combinators. Monadic Constraint Programming was based on a free monad to write both a model and its search tree, and to interpret it using a structurally recursive evaluation function. By means of open recursion (functional mixins) the search tree can be modified by various basic transformations that can be combined to form sophisticated search heuristics. In follow-up work we turned these transformations into a combinator language that could either be interpreted dynamically or in terms of code generators. Looking back on the work, much of the work could have been expressed more succinctly and systematically in terms of algebraic effects and handlers. I also identify an open challenge to set up the code generation approach by means of staging of handlers.

References:

- Tom Schrijvers, Peter J. Stuckey, Philip Wadler, "Monadic Constraint Programming". *J. Funct. Program.* 19(6):663–697 (2009). DOI 10.1017/S0956796809990086
- Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, Peter J. Stuckey, "Search Combinators". *Constraints.* 18(2):269–305 (2013). DOI 10.1007/s10601-012-9137-8

Multiple Prompts: In Little Steps

Paul Downen, University of Massachusetts Lowell

Systems for effect handlers often use a name-based system for connecting operations with their handling code, which helps to improve compositional reasoning by prevent unwanted cross-talk between different parts of the program. However, control operators for finding the correctly-named prompt, and abstracting over the context between it and the operator, are very complex, leading to many different design proposals which are difficult to compare. Which one should the implementors of effect handlers base their language on?

To better understand the different aspects of multiple prompts, we break down the problem into a series of little steps. Starting from a well-understood foundation based on common CPS transformations, several extensions are made

to extend a language with increasingly expressive operations, capturing the control operators call/cc, shift, and shift0, before finally reaching multi-prompt delimited control. By taking little steps, we can decompose the large and complex operations into smaller ones, which compose together to express a variety of multi-prompt control operators that have been proposed in the past.

Equivalence of Expressive Power between Labeled Effectful Calculi

Kazuki Ikemori, Tokyo Institute of Technology

Effect handlers and control operators are uniform abstractions for handling computational effects. Their labeled variations of effect handlers and control operators can express different instances of the same effect, such as exceptions and multiple state. Our goal is to show the equivalence of expressive power between labeled effect handlers and labeled control operators. To show this, we define the typed macro translations between these labeled effectful calculi and prove the type and meaning preservation properties.

One-shot Algebraic Effects as Coroutines

Yukiyoshi Kameyama, University of Tsukuba

We present a direct translation of one-shot algebraic effects and handlers to asymmetric coroutines by de Moura and Ierusalimschy. Compared with existing translations for delimited-control operators and coroutines, our translation is simple and macro-expressible, does not use states or other effects. We implemented our translation as a library in Lua and Ruby, that allows one to write effectful programs in a modular way using algebraic effects and handlers.

Interleaving Effectful Computation

Philipp Schuster, University of Tübingen

How can we connect a push producer, for example a lexer that emits tokens, and a pull consumer, for example a parser that reads tokens? We have to interleave the two effectful computations. We demonstrate how this is done in Effekt. The trick is to allocate a mutable reference that contains a computation which will overwrite the mutable reference with a new computation. We then demonstrate some preliminary work on translating these kinds of programs to continuation-passing style. This leads to mutually negatively recursive types. The terms, however, are not recursive, which could make them amenable to partial evaluation.

Lexical, Bidirectional Effect Handlers: Design & Implementation

Yizhou Zhang, University of Waterloo

Pressed by the difficulty of writing asynchronous, event-driven code, mainstream languages have recently been building in support for a variety of ad-

vanced control-flow features. Meanwhile, experimental language designs have suggested effect handlers as a unifying solution to programmer-defined control effects, subsuming exceptions, generators, and async-await. However, despite these trends, complex control flow—in particular, control flow that exhibits a bidirectional pattern—remains challenging to manage. We introduce bidirectional algebraic effects, a new programming abstraction that supports bidirectional control transfer in a more natural way. Handlers of bidirectional effects can raise further effects to transfer control back to the site where the initiating effect was raised, and can use themselves to handle their own effects. We present applications of this expressive power, which falls out naturally as we push toward the unification of effectful programming with object-oriented programming. We pin down the mechanism and the unification formally using a core language that makes generalizations to effect operations and effect handlers. The usual propagation semantics of control effects such as exceptions conflicts with modular reasoning in the presence of effect polymorphism—it breaks parametricity. Bidirectionality exacerbates the problem. Hence, we set out to show the core language, which builds on the existing tunneling semantics for algebraic effects, is not only type-safe (no effects go unhandled), but also abstraction-safe (no effects are accidentally handled). We devise a step-indexed logical-relations model, and construct its parametricity and soundness proofs. These core results are fully mechanized in Coq. While a full-featured compiler is left to future work, experiments show that as a first-class language feature, bidirectional handlers can be implemented efficiently.

Definitional Interpreter for Algebraic Effects and Handlers (AEH) (and what it is good for)

Kenichi Asai, Ochanomizu University

In this talk, I show a definitional interpreter for algebraic effects and handlers (AEH). The interpreter is based on three ingredients: the standard CPS interpreter (that supports shift), meta-continuations (for shift0), and trails (for control and control0). The definitional interpreter for AEH is then given by adding handler support. Depending on whether the depth is determined by handlers or operation calls, two variations are shown, one of which requires a recursive definition for handlers. The resulting interpreter is simple, shows clear correspondence to the definitional interpreter for four delimited control operators, and thus I believe is suitable as "the" definitional interpreter for AEH.

Links and WasmFX Status Update

Sam Lindley, The University of Edinburgh

I will give a status update on recent work relating to effect handlers in Links and WasmFX.

Links is a programming language for the web which has been used as a playground for research for the last 18 years. It included support for continuations as well as a row-based effect type system before effect handlers were added, so it was a natural environment in which to experiment with adding effect handlers. Links also supports session types which rely on a linear type system. At

a previous Shonan meeting in 2019, we reported a soundness bug resulting from a bad interaction between linear resources and multishot continuations. I will briefly outline how we have now fixed this soundness bug in Links by way of a new notion of control flow linearity.

WebAssembly (Wasm) is a universal intermediate language supported by all of the main web browsers. WasmFX is an extension of Wasm with support for effect handlers, aimed at making it much easier for language implementors to support concurrency features such as async/await, generators, and lightweight threads. I will briefly outline the current status of WasmFX. There is a full formal specification, an implementation in the Wasm reference interpreter, and current work is focusing on experimenting with an implementation as an extension of the standalone Wasm runtime Wasmtime.

Granule Status Report: Past, Present, and a Possible Future

Dominic Orchard, University of Kent and University of Cambridge

Granule is a functional programming language that incorporates linear, graded, and indexed types into a single typed language. The implementation was inspired by a string of theoretical papers between 2013-16 on coeffect and effect systems, which I then built a language prototype from in 2017. This talk revisits the origins of Granule and gives a demonstration of using Granule for fine-grained reasoning about functional programs, leveraging graded modal types in concert with indexed and linear types. In recent years, my group has used Granule as a research vehicle for (1) exploring grade-and-type directed program synthesis (2) type-based renderings of ownership, uniqueness, and borrowing, and (3) effect handlers. I will briefly touch on all these aspects and their implementation in Granule.

The State of Eff

Matija Pretnar, University of Ljubljana

The talk describes the history and the current state of Eff, the first programming language with native support of algebraic effect handlers. The first version of Eff was developed by Andrej Bauer and myself and appeared in 2010. It was untyped and had a syntax similar to Python. In the next version in 2011, we introduced the more commonly known ML-like syntax and dynamic generation of effect instances that worked analogously to references in OCaml. After that, the versions are a bit harder to reconstruct, but the main improvement was a subtyping-based effect system.

Building on the effect system, the next goal for Eff was an optimizing compiler, on which I worked together with Tom Schrijvers and his students. The compiler would produce efficient code by inlining handlers as much as possible, and monadically embed the rest in OCaml. It soon became obvious that the effect system has to be more robust, which was achieved by using coercions as explicit witnesses of subtyping. Currently, Filip Koprivec, my PhD student, focuses on simplifying those witnesses in the polymorphic setting, as otherwise each coercion needs to be passed around as an additional parameter at runtime.

In parallel to the optimizing compiler project, Žiga Lukšič developed EEff, a variant of Eff that tracks algebraic equations between operations that have to be respected by the handlers. Currently, EEff only prints out the obligations, though one could envision sending them to an SMT solver or producing specifications for a proof assistant.

Finally, the talk mentions Millet, a language one can use as a basis when developing a prototype language exploring an interesting novel idea. Millet implements all the boring and expected components of a programming language (parsing, algebraic datatypes, simple type-checking, interactive toplevel, compilation pipeline, ...), and any fork just needs to extend all the components with the support for the additional features. Then, any later change to Millet (right now, a module system, a Wasm backend, and LSP support are in the works) should be orthogonal and can hopefully be merged into forks with not too much additional work.

Effekt Status Update

Jonathan Brachthäuser, University of Tübingen

In this talk, I give an overview over the (short) history and motivation behind the Effekt language. It is positioned as a *functional imperative* programming language.

I present the compiler pipeline and illustrate it with a simple example. Each step in the pipeline comes with a proof of typability preservation and most parts also with a proof of semantics preservation. The compiler first makes handling explicit by converting into capability-passing style. It then makes delimited control explicit by performing a lift inference. Finally, we translate into iterated continuation-passing style.

After the overview, I also briefly discuss ongoing and future work, such as just-in-time compilation for effect handlers and improving error messages specific to effects.

Effect Handlers for Choice-based Learning

Ningning Xie, University of Toronto

Machine learning has achieved many successes during the past decades, spanning domains of game-playing, protein folding, competitive programming, and many others. However, while there have been major efforts in building programming techniques and frameworks for machine learning programming, there has been very little study of general language design for machine learning programming.

We pursue such a study in this talk, focusing on choice-based learning, particularly where choices are driven by optimizations. This includes widely-used decision-making models and techniques (e.g., Markov decision processes or gradient descent) which provide frameworks for describing systems in terms of choices (e.g., actions or parameters) and their resulting feedback as losses (dually, rewards).

We propose and give evidence for the following thesis: languages for choice-based learning can be obtained by combining two paradigms, algebraic effects and handlers with the selection monad. We provide a prototype implementation

as a Haskell library and present a variety of programming examples for choice-based learning: stochastic gradient descent, hyperparameter tuning, generative adversarial networks, and reinforcement learning.

ChiRo: A Causal Probabilistic Programming Language

Eli Bingham, Basis & Broad Institute of MIT and Harvard

Despite remarkable progress over the last two decades in reducing causal inference to statistical practice, the "causal revolution" proclaimed by Judea Pearl and other pioneers remains incomplete, with a sprawling and fragmented technical literature that is still inaccessible to non-experts and isolated from the cutting-edge computational methods and software tools being developed within mainstream machine learning research. Probabilistic programming languages are promising substrates for bridging this gap thanks to the close correspondence between their operational semantics and most standard mathematical formalisms for causal inference, especially that of structural causal models.

This talk will introduce ChiRho, a new causal probabilistic programming language embedded in Python. ChiRho extends an existing probabilistic programming language (Pyro) that is built on algebraic effects and handlers with new algebraic operations for expressing interventions and counterfactuals on causal models represented as probabilistic programs, and new effect handlers for automatically reducing causal inference computations over these models to ordinary probabilistic inference computations on transformed probabilistic programs. I will also illustrate ChiRho's design with a representative example application from single-cell biology: estimating the causal effects of drug treatments on cancer cells' gene expression directly from experimental data.

Quantum Computing as an Effect

Amr Sabry, Indiana University

Free categorical constructions characterise quantum computing as the combination of two copies of a reversible classical model, glued by the complementarity equations of classical structures. This recipe effectively constructs a computationally universal quantum programming language from two copies of Pi, the internal language of rig groupoids. The construction consists of Hughes' arrows. Thus answer positively the question whether a computational effect exists that turns reversible classical computation into quantum computation: the quantum effect.

A Functional Account of Probabilistic Programming with Possible Worlds

Tom Schrijvers, KU Leuven

While there has been much cross-fertilization between functional and logic programming—e.g., leading to functional models of many Prolog features—this appears to be much less the case regarding probabilistic programming, even though this is an area of mutual interest. Whereas functional programming

often focuses on modeling probabilistic processes, logic programming typically focuses on modeling possible worlds. These worlds are made up of facts that each carry a probability and together give rise to a distribution semantics. The latter approach appears to be little-known in the functional programming community. This talk aims to remedy this situation by presenting a functional account of the distribution semantics of probabilistic logic programming that is based on possible worlds. We present a term monad for the monadic syntax of queries together with a natural interpretation in terms of boolean algebras. Then we explain that, because probabilities do not form a boolean algebra, they—and other interpretations in terms of commutative semirings—can only be computed after query normalisation to deterministic, decomposable negation normal form (d-DNNF). While computing the possible worlds readily gives such a normal form, it suffers from exponential blow-up. Using heuristic algorithms yields much better results in practice.

- Birthe van den Berg, Tom Schrijvers, "A Functional Account of Probabilistic Programming with Possible Worlds" (Declarative Pearl). FLOPS 2022:186–204. DOI 10.1007/978-3-030-99461-7_11

Effectful Software Contracts

Cameron Moy, Northeastern University

Contracts enable programmers to describe sophisticated properties of their components. Effects are sometimes needed to monitor such properties. Unrestricted effects in contracts, though, can violate desirable reasoning principles. This talk proposes effect handlers as a means to unify the existing landscape of effectful contracts, while guaranteeing that contracts remain well behaved.

Answer-Refinement Modification: A Refinement Type System for Algebraic Effect Handlers

Taro Sekiyama, National Institute of Informatics

In this talk, I will introduce a refinement type system for algebraic effects and handlers. The expressivity and usefulness of algebraic effects and handlers come from their ability to manipulate delimited continuations, but delimited continuations also complicate programs' control flow and make their verification harder. To address the complexity, the proposed refinement type system is empowered with a novel concept that we call answer refinement modification (ARM for short). ARM allows the refinement type system to precisely track what effects occur and in what order when a program is executed by reflecting such information about effects as modifications to the refinements in the types of delimited continuations. I will demonstrate the usefulness of the refinement type system with ARM by reasoning about program examples exploiting algebraic effects and handlers.

Lessons from Pyro

Eli Bingham, Basis & Broad Institute of MIT and Harvard

Pyro is a probabilistic programming language embedded in Python that focuses

on scalable, gradient-based algorithmic approaches to approximate Bayesian inference. Like Stan, PyMC and many other more established PPLs, Pyro has been used by computational scientists, engineers and statisticians to solve a diverse array of real-world probabilistic machine learning problems. Unlike those other PPLs, Pyro is built on a foundation of algebraic effects and handlers, a design choice that has provided a high level of flexibility and extensibility to its users and developers. Moreover, as a result of the many strong constraints imposed by the needs of users, the structure of inference algorithms, and the choice of host language, Pyro's implementation of effect handlers differs somewhat from standard presentations of handlers in other non-probabilistic languages.

In this talk, I will give a brief, opinionated tour of Pyro's operational semantics, with a focus on a small set of core features that differentiate Pyro and its handlers from other languages. For each feature, I will summarize the circumstances that motivated its design, describe the feature as it exists in Pyro today, warts and all, and speculate on possible implications for the design of a hypothetical future effect handling system intended as a foundation for a Pyro successor or other advanced machine learning systems.

Graded Algebraic Effects and Handlers

Dominic Orchard, University of Kent and University of Cambridge

The graded type system of Granule provides an opportunity to present algebraic effects and handlers with explicit requirements on how continuations are used: zero-shot (discardable), one-shot and multi-shot continuations can be specified precisely and enforced via graded types. Granule provides an implementation of algebraic effects via an embedding of the free graded monad, against which graded signatures can be written and graded algebras for handlers. This talk demonstrates state and non-determinism in the graded algebraic effects and handlers style, where in particular, the non-deterministic case can require that handlers use the continuation for 'failure' 0 times, and for branching choice exactly 2 times.

Control As Proofs: And the Route to Composition, in Three Parts

Paul Downen, University of Massachusetts Lowell

The Curry-Howard Correspondence has created an industry for linking important insights and techniques that are shared between formal logic and programming languages. For decades, it was commonly believed that this correspondence could not include classical logic, until the 1990s when it was discovered that classical reasoning was connected to first-class continuations. This formed a computational interpretation of classical logic where the programmer can effect change on the control flow in a program using operators like call/cc.

However, on the side of programming, this style of control effect is much weaker than an alternative — delimited control — which allows for a more composable approach to control-flow manipulation. From the starting point of classical proofs as programs, there are two different methods of composing continuations, either through a dynamically-rebindable "top level" continuation,

or by simply using the same programs in new contexts. While inspired by the Curry-Howard Correspondence for classical logic, neither of these approaches (yet) is tied to their own form of logic. Possibly, the logical interpretation of delimited control might be achieved by expressing some of its important aspects in a more structured form by using exotic logical connectives which are not yet found in programming languages.

Coexponentials

Amr Sabry, Indiana University

Continuations have traditionally been viewed as "dual" to values. Despite many attempts to formalize this intuition, there remains some ad hoc corners. We propose that coexponentials provide an improved interface to continuations using cofunctions as the natural dual of functions.

Effect Handlers and Multi-Stage Programming

Jeremy Yallop, University of Cambridge

Effect handlers interact with quotation in multi-stage programming languages both virtuously and viciously. In this talk I illustrate how effect handlers are convenient for implementing the let- and if-insertion operations that are often needed in multi-stage programs, and how unrestricted use of effect handlers can give rise to scope extrusion errors.

Type-safe Effectful Staging

Yukiyoshi Kameyama, University of Tsukuba

Staging allows a programmer to write domain-specific, custom code generators. Ideally, a programming language for staging provides all necessary features for staging, and at the same time, gives static guarantee for the safety properties of generated code including well typedness and well scopedness. We address this classic problem for the language with control operators, which enables code optimizations in a modular and compact way. In the work with Oishi, we designed a type-safe two-stage language with the delimited-control operators shift0 and reset0, which allows us to express multi-layer let-insertion. In the work with Yokoyama, we extended the result to a two-stage language with algebraic effects and handlers.

Higher-Order Effects

Tom Schrijvers, KU Leuven

In this talk I explain how algebraicity is a property that can be constructively added to effectful operations by means of the Cayley construction. I also identify higher-order effects as those where the operation's signature depends on the computation's monad because it takes (non-continuation) computations as parameter. By means of a higher-order free monad and accompanying structural recursion scheme these computation parameters can be handled uniformly

with all other computations without the programmer having to explicitly write recursive effect handlers.

- Birthe van den Berg, Tom Schrijvers, "A Framework for Higher-Order Effects & Handlers". CoRR abs/2302.01415 (2023). DOI 10.48550/arXiv.2302.01415

Control-in Dependent Types

William J. Bowman & Paulette Koronkevich, University of British Columbia

In this talk, we'll walk through what goes wrong when you want to transform dependently typed programs to make control or data flow explicit. Such transformations, or their image, have many uses: designing and implementing compilers, optimizations, implementing control effects, or structuring (or modeling) imperative computation. We'll look in particular at CPS and ANF transformations, what goes wrong, how to make them work for dependent types, and what general lessons we might take away from this.

Effective Equality: Overcoming Obstacles with Beta and Eta

Paul Downen, University of Massachusetts Lowell

Effects make the art of equational reasoning — determining whether two expression always give equivalent results in the context of any program — more difficult. Or do they? Many of the complications introduced by side effects like mutable state already appear in general purpose functional language that avoid them. In contrast, the side effect of first-class control gives us a tool *within* the programming language which helps express and reason about the equality of programs — with or without side effect — in a more principled way.

This talk gives a survey of lessons learned while building and using equational theories for programming languages that use a variety of effects. The general philosophy revolves around pairs of simple syntactic rewrites for the operational and extensional facts of each programming language feature — based on the beta and eta laws of the lambda calculus — combined with a certain equality related to control flow in abstract machines. This gives a significant approximation of observational equality and derives a large number of other specialized reasoning principles used in practice. The use of labeled control flow also gives us a syntactic method for more complex forms of equational reasoning, such as coinduction, that is valid in both eager and lazy languages, both with and without effects. Finally, the simple beta- and eta-based equational reasoning is used in a setting of more general effects, including delimited control and ultimately effect handlers, where several embeddings of effect handlers are explored.

Designing a Language for Learning Continuations

Youyou Cong, Tokyo Institute of Technology

Continuations are being introduced into various programming languages. This trend creates a need for a language that provides support for understanding the

concept of continuations. In this talk, I explore with the audience the design space of such a language.

What's Lexical about Lexical Handlers?

Jonathan Immanuel Brachthäuser, University of Tübingen, Germany

Operationally, there are two important aspects to effects and handlers: dynamic scoping and delimited continuations. Effect handlers are *dynamically scoped*, that is, effects will always be handled by the dynamically closest handler surrounding the call to the effect operation. To handle an effect, the *delimited continuation*, which represents the remainder of the program from the operation call up to and including the handler, is captured and made available in the handler.

This dynamic behavior can lead to accidental capture and preventing it can result in a problem referred to as "effect encapsulation problem". As an alternative solution to prevent accidental capture, *lexical handlers* have been introduced in various forms (Biernacki et al. 2019, Zhang and Myers 2019, Brachthäuser et al. 2020). The terminology "lexical handlers" has ever since caused some confusion in the community since in realistic implementations, such as the Effekt language, handlers are not entirely lexically scoped.

This talk presented the result of discussions led at the Shonan meeting. We attempt to distill the essence of lexical handlers (denoted $\mathsf{handle}_{\#}$ in the following) and contrast it with dynamic handlers (denoted handle) by means of a single reduction rule:

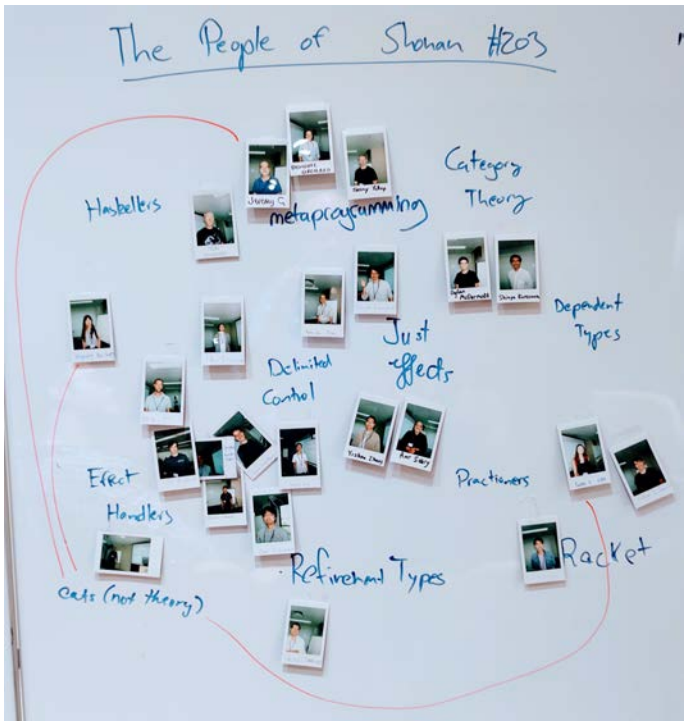
$$\begin{array}{l} \mathsf{handle}_{\#}\ s\ \mathsf{with}\ \{\ \overline{op\ (x,\ k) \to s}\ \uplus\ \mathsf{return}\ x \to s_1\ \} \\ \quad \longrightarrow \\ \mathsf{handle}\ s[\overline{op \mapsto op_l}]\ \mathsf{with}\ \{\ \overline{op_l\ (x,\ k) \to s}\ \uplus\ \mathsf{return}\ x \to s_1\ \} \\ \quad \text{where}\ \ l\ \ \text{fresh} \end{array}$$

The rule establishes lexical scoping by renaming operations that are in the lexical scope of the handler and then falls back to the standard semantics of dynamic handlers. Importantly, handlers (dynamic and lexical) are considered binding occurrences for operations.

- Biernacki, D., Piróg, M., Polesiuk, P., & Sieczkowski, F. (2019). "Binders by day, labels by night: effect instances via lexically scoped handlers". Proceedings of the ACM on Programming Languages, 4(POPL), 1-29. DOI 10.1145/3371116
- Brachthäuser, J. I., Schuster, P., & Ostermann, K. (2020). Effects as capabilities: effect handlers and lightweight effect polymorphism. Proceedings of the ACM on Programming Languages, 4(OOPSLA), 1-30. DOI 10.1145/3428194
- Zhang, Y., & Myers, A. C. (2019). Abstraction-safe effect handlers via tunneling. Proceedings of the ACM on Programming Languages, 3(POPL), 1-29. DOI 10.1145/3290318

4 List of participants

- Kenichi Asai, Ochanomizu University
- Eli Bingham, Basis & Broad Institute of MIT and Harvard
- William J. Bowman, University of British Columbia
- Paul Downen, University of Massachusetts Lowell
- Kazuki Ikemori, Tokyo Institute of Technology
- Yukiyoshi Kameyama, Univeristy of Tsukuba
- Shin-ya Katsumata, National Institute of Informatics
- Paulette Koronkevich, University of British Columbia
- Sam Lindley, The University of Edinburgh
- Dylan McDermott, Reykjavik University
- Cameron Moy, Northeastern University
- Dominic Orchard, University of Kent and University of Cambridge
- Amr Sabry, Indiana University
- Tom Schrijvers, KU Leuven
- Philipp Schuster, University of Tübingen
- Taro Sekiyama, National Institute of Informatics
- Tachio Terauchi, Waseda University
- Ningning Xie, University of Toronto
- Jeremy Yallop, University of Cambridge
- Yizhou Zhang, University of Waterloo



5 Summary of discussions

Two discussion sessions were held: one on Tuesday with the topic "Concurrency and Distribution", and one on Thursday with the topic "Will Effects and Handlers Replace Monads?". Each was run as a fishbowl conversation, with a rotating panel.

Fishbowl Discussion: Concurrency and Distribution

All participants; initial panel: Ningning Xie, Sam Lindley, Taro Sekiyama, Philipp Schuster

We began the discussion with a general clarification of the terms *concurrency*, *parallelism*, and *distribution*. The three are different. A distinguishing feature of distribution is that failure is pervasive. Is concurrency only for making things happen simultaneously or (particularly with speculative concurrency) faster, or is it also a modularity tool? We then discussed *preemptive* versus *collaborative* concurrency. Some practical systems simulate preemption by forcing collaboration, i.e. by inserting yields; this was done for example in the context of the Frank language. Can all use-cases of preemptive concurrency be expressed like this? We briefly discussed existing work on the combination of effect handlers and concurrency. In Æff, preemptive concurrency is a separate concept from effect handlers. Does it have to be? The equation $\mathsf{fork}(\mathsf{yield}(z_1), z_2) = \mathsf{fork}(z_2, z_1)$ is hard to prove given the standard implementation of a scheduler with effect handlers. Why is it so hard to come up with an equational theory for concurrency? It was proposed to start from process algebra, go via monads, and then arrive at effect handlers for concurrency. Perhaps parametrized algebraic theories would emerge? On the more practical side, the importance of speculation for performance was emphasized. This led to the topic of transactions, and more specifically transactional memory. Given that in languages with effect handlers we usually already track effects, does that make transactional memory work better? We then switched to parallelism. How can we use effect handlers for or with parallelism? Is parallelism a separate effect external to handlers? We noticed that there is a class of *commutative* effect operations that can safely be executed concurrently, such as incrementing a counter or appending to a log (assuming that log ordering is not significant). This has been observed in work on parallel execution of for-loops. What other useful commutative effects are there?

Fishbowl Discussion: Will Effects and Handlers Replace Monads?

All participants; initial panel: Shin-ya Katsumata, Eli Bingham, Yukiyoshi Kameyama, Cameron Moy

The discussion began by trying to clarify the role of a monad. It was noted that monads are a mathematical concept, much like associativity of operations. Associativity is also not a programming language feature, but a tool for programmers and language designers to reason about programs. Similarly, monads are a mathematical concept that helps modeling programs. It was conjectured that, while new programming language features might replace the usage of monads as a programming language feature, they will very likely still be relevant as a mathematical tool for reasoning. The discussion then shifted to the contrasting paradigms of monads and effect handlers as programming devices. It was noted that monads put "semantics first", such as starting from lists and then discovering operations for non-determinism. In contrast, effect handlers follow the notion of "syntax first", describing the algebraic operations and programming against this interface, to then later choose an appropriate semantics. It was conjectured that putting syntax first might be easier for programmers, since they are used to working with interfaces. It also might lead to software which is more modular. A controversial take was brought up, suggesting that monads are not commonly used, sparking a reflection on the utility of these abstractions for the average programmer. Some also noted that as programming language designers it should be our priority to support average programmers in understanding and managing effects in programs. The discussion advanced with a point that any abstraction aiding in clarifying effects in types is valuable, be it monads or effect handlers, with Java exceptions cited as a useful example. Finally, it was discussed how the conceptual proximity of effect handlers to exceptions could help in educating programmers about them. It might be easier to explain effect handlers, starting from exceptions and then explaining how to recover from them, then teaching monads.

6 Future directions

Repositories: The effect-bench repository for benchmarking effect handlers has had a change of editors: Filip Koprivec is stepping down, to be replaced by Jesse Sigal, who will join Philipp Schuster. We also reminded ourselves of the Effects and Handlers Rosetta Stone, "a collection of examples demonstrating programming with effects and handlers in various programming languages".

Book: We discussed the invitation from NII to produce a book on the subject of this meeting. On the one hand, this was considered to be a valuable project—it would be very helpful to have a focussed and curated collection of chapters that one could give to a new student in the field, whether of new or republished material. We envisioned a division into several parts: tutorials; relevant theory; programming aspects; implementation techniques; application areas; research directions. On the other hand, the *Communications of NII Shonan Meetings* series published by Springer is not open access, and the books are not cheap. There does not seem to be any point in the endeavour if the content would not be easily accessible.

Next meeting: We will submit a proposal for a Dagstuhl Seminar to follow this meeting. Proposed organizers are Jonathan Brachthäuser, Daniel Hillerström, Yukiyoshi Kameyama, and Ningning Xie; a possible theme is "beyond algebraic effects".