Report from Dagstuhl Seminar 16112

From Theory to Practice of Algebraic Effects and Handlers

Edited by

Andrej Bauer[1], Martin Hofmann[2], Matija Pretnar[3], and Jeremy Yallop[4]

1 **University of Ljubljana, SI,** andrej.bauer@fmf.uni-lj.si
2 **LMU München, DE,** hofmann@ifi.lmu.de
3 **University of Ljubljana, SI,** matija.pretnar@fmf.uni-lj.si
4 **University of Cambridge, GB,** jeremy.yallop@cl.cam.ac.uk

Abstract

Dagstuhl Seminar 16112 was devoted to research in algebraic effects and handlers, a chapter in the principles of programming languages which addresses computational effects (such as I/O, state, exceptions, nondeterminism, and many others). The speakers and the working groups covered a range of topics, including comparisons between various control mechanisms (handlers vs. delimited control), implementation of an effect system for OCaml, compilation techniques for algebraic effects and handlers, and implementations of effects in Haskell.

Seminar March 13–18, 2016 – http://www.dagstuhl.de/16112
1998 ACM Subject Classification D.3 Programming Languages, D.3.3 Language Constructs and Features: Control structures, Polymorphism, F.3 Logics and Meanings of Programs, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages, F.3.3 Studies of Program Constructs: Control primitives, Type structure
Keywords and phrases algebraic effects, computational effects, handlers, implementation techniques, programming languages
Digital Object Identifier 10.4230/DagRep.6.3.44
Edited in cooperation with Niels F. W. Voorneveld and Philipp G. Haselwarter

1 Executive Summary

Andrej Bauer
Martin Hofmann
Matija Pretnar
Jeremy Yallop

Being no strangers to the Dagstuhl seminars we were delighted to get the opportunity to organize Seminar 16112. Our seminar was dedicated to algebraic effects and handlers, a research topic in programming languages which has received much attention in the past decade. There are strong theoretical and practical aspects of algebraic effects and handlers, so we invited people from both camps. It would have been easy to run the seminar as a series of disconnected talks that would take up most of people's schedules – we have all been to such seminars – and run the risk of disconnecting the camps as well. We decided to try a different format, and would like to share our experience in this executive summary.

On the first day we set out to identify topics of interest and organize working groups around them. This did not work, as everybody wanted to be in every group, or was at least

worried they would miss something important by choosing the wrong group. Nevertheless, we did identify topics and within them ideas began to form. At first they were very general ideas on the level of major research projects, but soon enough people started asking specific questions that could be addressed at the seminar. Around those questions small groups began to form. Out of initial confusion came self-organization.

We had talks each day in the morning, with the schedule planned two days ahead, except for the first day which started by a tutorial on algebraic effects and handlers. We left the afternoons completely free for people to work in self-organized groups, which they did. The organizers subtly made sure that everybody had a group to talk to. In the evening, just before dinner, we had a "show & tell" session in which groups reported on their progress. These sessions were the most interesting part of the day, with everyone participating: some showing what they had done so far, and others offering new ideas. Some of the sessions were accompanied by improvised short lectures.

Work continued after dinner and late at night. One of the organizers was shocked to find, on his way to bed, that the walls of a small seminar room were completely filled with type theoretic formulas, from the floor to the ceiling. He was greatly relieved to hear that the type theory was not there to stay permanently as the Dagstuhl caretakers painted the walls with a special "whiteboard" paint. They should sell the paint by the bucket as a Dagstuhl souvenir.

We are extremely happy with the outcome of the seminar and the way we organized it. An open format that gives everyone ample time outside the seminar room was significantly boosted by the unique Dagstuhl environment free of worldly distractions. We encourage future organizers to boldly try new ways of organizing meetings. There will be confusion at first, but as long as the participants are encouraged and allowed to group themselves, they will do so. If a lesson is to be taken from our seminar, it is perhaps this: let people do what they want, but also make sure they report frequently on what they are doing, preferably when they are a bit hungry.

2 Table of Contents

3 Overview of Talks

3.1 Handlers considered harmful?

Andrzej Filinski (University of Copenhagen, DK)

At a seminar about handlers for algebraic effects – often presented as an operationally oriented alternative to more explicitly monad-based approaches for specifying and implementing computational effects – it is important not to forget about some of the considerable theoretical and practical strengths of monads. This talk outlines some areas in which effect handlers – as currently conceived in languages like Eff – may show comparative weaknesses, and is meant to inspire reflection and discussion on how those can best be addressed.

3.2 Andromeda: Type theory with Equality Reflection

Philipp G. Haselwarter (University of Ljubljana, SI)

Joint work of Andrej Bauer; Gaëtan Gilbert; Philipp G. Haselwarter; Matija Pretnar; Christopher A. Stone
Main reference http://andromedans.github.io/andromeda/
URL https://github.com/Andromedans/andromeda/releases/tag/dagstuhl-2016

We present Andromeda, a proof assistant for dependent type theory with equality reflection in the style of [1]. The design of Andromeda follows the tradition of Edinburgh LCF, in the sense that

- there is an abstract datatype of type-theoretic judgements whose values can only be constructed by a small, trusted nucleus
- the user interacts with the nucleus by writing programs in a high-level Andromeda meta-language (AML)

The type theory of Andromeda has dependent products and equality types. The rules for products are standard and include function extensionality. This flavour of dependent type theory is very expressive, as it allows one to postulate new *judgemental equalities* through the equality reflection rule. However, this comes at the expense of rendering type-checking undecidable. As there is no complete type-checking algorithm that we could implement in the nucleus, we rely on user code written in AML to prove complex equality judgements.

We demonstrate how we use effects and handlers as a mechanism for the nucleus to communicate with the user-code by asking questions about equalities. We then showcase how equality reflection can be used to introduce inductive types with their judgemental computation rules and to control opaqueness of definitions. Finally, we present how a meta-language with effects can be used to implement a memoization tactic.

References

1 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

3.3 No value restriction is needed for algebraic effects and handlers

Ohad Kammar (University of Cambridge, GB), Sean Moss, and Matija Pretnar (University of Ljubljana, SI)

We present a straightforward sound Hindley-Milner polymorphic type system for algebraic effects and handlers which allows type variable generalisation of arbitrary computations, and not just values. This result is surprising. On the one hand, the soundness of Hindley-Milner polymorphism is known to fail when not restricted in the presence of computational effects such as reference cells and continuations. On the other hand, many programming examples can be recast to use effect handlers instead of these effects. We place this result in the wider context in two ways. First, we discuss the expressive difference between reference cells and programming with algebraic effects and handlers. Second, we present a parametric set-theoretic denotational semantics that highlights the smooth interaction of algebraic effects and polymorphism.

3.4 Parameterized Extensible Effects and Session Types

Oleg Kiselyov (Tohoku University – Sendai, JP)

Parameterized monad goes beyond monads in letting us represent type-state. An effect executed by a computation may change the set of effects it may be allowed to do afterwards. We describe how to easily 'add' and 'subtract' such type-state effects. Parameterized monad is often used to implement session types. We point out that extensible type-state effects are themselves a form of session types.

3.5 Adequacy for Infinitary Algebraic Effects

Gordon Plotkin (University of Edinburgh, GB)

Moggi famously proposed a monadic account of computational effects which includes the computational λ-calculus, a core call-by-value functional programming language. One naturally then seeks a correspondingly general treatment of operational semantics. In the algebraic theory of effects, a refinement of Moggi's theory, the effects are obtained by appropriate operations, and the monad is generated from an equational theory over these operations.

In a previous paper with John Power, a general adequacy theorem was given for the case of monads generated by finitary operations. This covers examples such as probabilistic nondeterminism and exceptions. The idea is to evaluate terms symbolically in the absolutely free algebra with the same signature as the equational theory. Without recursion, the evaluated terms are finite; with recursion, they may be infinitely deep.

In general, however, one needs infinitary operations, for example for interactive I/O. We review the previous work and show it can be extended to include such operations by allowing infinitely wide terms. We can also define a general contextual equivalence for any monad, however an extensional characterisation is elusive. The work should be extended to cover handlers.

In most cases the natural adequacy theorem for a given effect is directly obtained from the symbolical one. An exception is state, as the symbolic operational semantics has no state component. It remains an interesting question to give a general operational semantics with a notion of state.

3.6 A tutorial on algebraic effects and handlers

Matija Pretnar (University of Ljubljana, SI)

Main reference M. Pretnar, "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper", Proc. of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI), Electr. Notes Theor. Comput. Sci., Vol. 319, pp. 19–35, 2015.
URL http://dx.doi.org/10.1016/j.entcs.2015.12.003

The seminar started with a tutorial, which had a two-fold purpose of establishing a common terminology and of introducing algebraic effects and handlers to anyone not yet familiar with them. Roughly half of the audience was familiar with algebraic effects, but everyone was well versed in functional programming and computational effects.

In the tutorial, we first looked at the basic idea of algebraic effects: every computation returns a value or performs an effect by calling an operation. Therefore, the effectful behaviour can be captured in an algebraic theory comprising a set of basic operations and equations between them. We have shown how this leads to an interpretation of computations with trees that have called operations as branching points and returned values as leaves. This furthermore results in an algebraic denotational semantics, where computations are interpreted with free models of the aforementioned algebraic theory.

Next, we have looked at how one may generalize exception handlers to handlers of other algebraic effects, and the subtleties involved in the generalisation. Using many simple examples of input & output handlers, we explored the flexibility that handlers offer in managing the control flow of programs. As a more involved example, we took a look at how one may implement many variants of backtracking with a handler for the non-deterministic choice operation. We have also revisited the algebraic semantics and seen how handlers correspond exactly to the homomorphisms, induced by the universal property of the free model.

Finally, we sketched how one may adapt a standard type system for a call-by-value language into a type & effect system, which captures the set of potentially called operations in addition to the type of returned values.

3.7 Compiling Eff to OCaml

Matija Pretnar (University of Ljubljana, SI), Amr Hany Shehata Saleh (KU Leuven, BE), and Tom Schrijvers (KU Leuven, BE)

We introduce a compilation technique for Eff, a functional language with handlers of algebraic effects. Our compiler converts an Eff program into an OCaml program that produces an element of the free monad. In order to reduce the performance overhead of the generated code, we introduce a number of optimizations.

The most crucial technique, when feasible, is to translate pure computations into direct OCaml code. For example, an Eff computation `1 + 3`, is first translated into

```
(fun x -> Return (fun y -> Return (x + y))) 1 >>= fun f ->
f 3
```

where `Return` and `>>=` are the unit and binding operation of the free monad, and `+` is native addition in OCaml. However, monadic binds are costly, so our desire is to optimize the generated code to just `Return (1 + 3)`, which we do through a series of rewriting rules.

According to our benchmarks, the optimized generated code performs at about half the speed of hand-written OCaml code. We plan to use the information provided by an effect system to further optimize the output code.

3.8 Effect Handlers in Scope

Tom Schrijvers (KU Leuven, BE)

Joint work of Nicolas Wu; Tom Schrijvers; Ralf Hinze
Main reference N. Wu, T. Schrijvers, R. Hinze, "Effect handlers in scope", in Proc. of the 2014 ACM SIGPLAN Symp. on Haskell (Haskell'14), pp. 1–12, ACM, 2014.
URL http://dx.doi.org/10.1145/2775050.2633358

Algebraic effect handlers are a powerful means for describing effectful computations. They provide a lightweight and orthogonal technique to define and compose the syntax and semantics of different effects. The semantics is captured by handlers, which are functions that transform syntax trees. Unfortunately, the approach does not support syntax for scoping constructs, which arise in a number of scenarios. While handlers can be used to provide a limited form of scope, we demonstrate that this approach constrains the possible interactions of effects and rules out some desired semantics. This paper presents two different ways to capture scoped constructs in syntax, and shows how to achieve different semantics by reordering handlers. The first approach expresses scopes using the existing algebraic handlers framework, but has some limitations. The problem is fully solved in the second approach where we introduce higher-order syntax.

3.9 Compositional reasoning for algebraic effects

Alex Simpson (University of Ljubljana, SI)

We obtain compositional proof systems for program verification by combining a set of generic rules, common to all language instantiations, with composition principles that must be supplied on an effect-specific basis. The proposed framework considers effects as generated by a signature of algebraic operations. The effect-specific composition principles then replace the customary equations (which are derivable from them).

3.10 Substitution, jumps and algebraic effects

Sam Staton (University of Oxford, GB)

Joint work of Marcelo Fiore; Sam Staton
Main reference M. Fiore and S. Staton, "Substitution, jumps, and algebraic effects", in Proc. of the Joint Meeting of the 23rd EACSL Annual Conf. on Computer Science Logic and the 29th Annual ACM/IEEE Symp. on Logic in Computer Science (CSL-LICS'14), 2014; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2603088.2603163
URL http://www.cs.ox.ac.uk/people/samuel.staton/papers/lics2014-substitution.pdf

I spoke about the relationship between jumps and the theory of substitution. To give an algebra for the theory of substitution is to give a first-order algebraic theory. I discussed how this explains the implementation of algebraic effects using control effects.

3.11 LiquidHaskell: Refinement Types for Haskell

Niki Vazou (University of California – San Diego, US)

Joint work of Alexander Bakst; Eric Seidel; Ranjit Jhala; Niki Vazou
Main reference N. Vazou, A. Bakst, R. Jhala, "Bounded refinement types", in Proc. of the 20th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'15), pp. 48–61, ACM, 2015; pre-print available from author's webpage.
URL http://dx.doi.org/10.1145/2784731.2784745
URL http://goto.ucsd.edu/~nvazou/icfp15/main.pdf

We saw LiquidHaskell, a decidable and highly automated verifier that uses refinement types for Haskell source code. I presented some examples (including safety of division and list sorting) that can be found in the online demo [1].

Also we saw how refinement types can be extended with bounds [2] leading to more expressive specifications that can be used to specify and verify effectual computations.

References

1 Online demo: http://goto.ucsd.edu/~nvazou/compose16/_site/01-index.html
2 Niki Vazou, Alexander Bakst, Ranjit Jhala. *Bounded refinement types*. ICFP 2015. http://goto.ucsd.edu/~nvazou/icfp15/main.pdf

4 Working groups

4.1 Towards an effect system for OCaml

Matija Pretnar (University of Ljubljana, SI), Stephen Dolan (University of Cambridge, GB), KC Sivaramakrishnan (University of Cambridge, GB), and Leo White (Jane Street – London, GB)

With the introduction of algebraic effects to OCaml[1], extending OCaml's type system into a type & effect system is a natural next step. In such a system, programs receive a type $A!\mathcal{E}$, where A is the type of returned values, and $\mathcal{E}$ is the effect annotation, whose exact form is yet to be determined. Even though there is already an existing polymorphic effect system for handlers with an inference algorithm [3], it is not obvious how to include it in OCaml due to backwards compatibility.

There are a number of properties that a feasible effect system should satisfy:

Soundness If a program e receives a type $A!\mathcal{E}$, every potential effect `E` should be captured in $\mathcal{E}$.

Usefulness An effect system that annotates each program with every possible effect there is, is obviously sound, but not very useful. Thus, an effect information should not mention an effect that is guaranteed not to happen.

Backwards compatibility We want each program that was typable before introducing effect annotations, to remain typable. Furthermore, the effect system should play along nicely with OCaml's module system, thus whole-program analysis is out of the question.

To see what the above properties imply, take a program

```
if X then perform E1 else perform E2
```

The effect information of `perform E1` must mention `E1` for the sake of soundness, but omit `E2` for the sake of usefulness. Conversely, the effect information of `perform E2` should mention `E2` but not `E1`. But the whole program must remain typable due to backwards compatibility, and its type should mention both `E1` and `E2` due to soundness. From this, it follows that the effect system needs to provide a way of enlarging effect information. There are two established ways of providing this flexibility: subtyping [4] or row polymorphism [1]. Both are difficult to apply directly to OCaml, due to already-existing language features:

Monomorphic types The ML type system makes a distinction between monomorphic and polymorphic types, and in certain contexts only monomorphic types are permitted. Many existing programs are typeable only because, say, `int` → `int` is monomorphic, and would break if it became a polymorphic type.

Signature matching Comparing a module implementation against its interface requires not only inferring polymorphic types, but checking whether a given polymorphic type is more polymorphic than another.

Invariant contexts While OCaml supports (explicit) subtyping, not all type parameters are either co- or contra-variant. For instance, the type parameters to `ref`, the indices of GADTs, and unannotated abstract types are neither co- nor contra-variant.

Subtyping makes type inference difficult by breaking unification, so the usual approach is to infer *constrained types* of the form $A|\mathcal{C}$, where $\mathcal{C}$ is the set of constraints between

[1] https://github.com/ocamllabs/ocaml-effects

type (and later also effect) parameters in A [2]. However, there are a number of practical problems. First, it is hard to determine when a constrained type $A|\mathcal{C}$ is an instance of $A'|\mathcal{C}'$, causing problems for compatibility with the module system. Next, constraint generation in the inference algorithm needs to be directed in order to keep track of covariance and contravariance. This causes problems with the current inference algorithm of OCaml, which mostly works with equations and is undirected. Finally, constraints are cumbersome to write and difficult to read, decreasing chances of adoption in the programming community.

A possible solution for subtyping is to encode constraints in types, potentially dropping some of them, which results in types that satisfy a weak form of principality: the inferred type is unique and captures most of possible typings of the given program, but not all of them.

For row polymorphism, typability of existing programs poses a problem. These programs, which may cause any effect provided by OCaml (input/output, references, ...), should receive an annotation, say `IO`, that distinguishes them from pure programs. Furthermore, existing monomorphic types should remain monomorphic. For example, a function `old_fun` that used to have a type `unit` → `unit` should get a type `unit` → (`unit!IO`). However, one then cannot type the program `if X then old_fun () else perform E`, as the type of the left branch does not contain a row variable and cannot be expanded to mention `E`.

A possible solution for this issue is to give monomorphic types to existing monomorphic programs, but allow a limited form of subeffecting, which weakens the effect annotation during application. Then, for example, `old_fun` would have a type `unit` → (`unit!IO`), but its application `old_fun ()` would get the type `unit!`[`IO`|ρ].

References

1 Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, volume 153 of *EPTCS*, pages 100–126, 2014.

2 François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report RR-3483, INRIA, 1998.

3 Matija Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

4 Keith Wansbrough and Simon L. Peyton Jones. Once upon a polymorphic type. In *POPL*, pages 15–28. ACM, 1999.

5 Open problems

5.1 Are all functions continuous and how to prove it?

Andrej Bauer (University of Ljubljana, SI)

5.1.1 Mathematical background

Brouwer's statement *"all functions are continuous"* can be formulated without reference to topology as follows. A functional $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is *continuous* at $a : \mathbb{N} \to \mathbb{N}$ when there exists $m : \mathbb{N}$ such that, for all $b : \mathbb{N} \to \mathbb{N}$, if $\forall k < m, ak = bk$ then $fa = fb$. This says that the value of fa depends only on the initial segment $a\,0$, $a\,1$, ..., $a\,(m-1)$.

The statement *"all functionals are continuous everywhere"* is valid in various models of intuitionistic mathematics, such as Kleene's number realizability and Kleene's function

realizability. We can ask whether the statement is realized in any given functional programming language. Such a realizer is called a *modulus of continuity* and is a functional $\mu : ((\mathbb{N} \to \mathbb{N}) \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ such that, for all $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ and $a, b : \mathbb{N} \to \mathbb{N}$, if $\forall k < \mu f a \,.\, a\, k = b\, k$ then $f a = f b$. Essentially, $\mu f a$ computes how much of a is needed to compute $f a$.

5.1.2 Implementing the modulus of continuity

It is impossible to implement μ in PCF and Haskell. Briefly, every hereditarily total functional definable in PCF is extensional (one can use Ulrich Berger's theory of totality [2] to establish this), while a result by Troelstra [3, §9.6.10–9.6.11] shows that an extensional modulus of continuity violates choice principles that are realized by PCF.

Therefore, we necessarily need additional computational features that let μ inspect the workings of f. Here are a few attempts, where we pretend that the type of integers is the type of natural numbers (we ignore negative values).

ML with references

Consider ML with references (and no other features). Then a possible μ is

```
let mu_ref f a =
  let k = ref 0 in
  let a' n = (k := max !k n; a n) in
    f a' ; !k
```

However:
1. Can f use its own local references? If it can use them in an unrestricted way then it can break `mu_ref`. How do we reasonably restrict the use of local references by f?
2. More generally, how do we formulate the exact preconditions on f and a?
3. What is the theorem that needs to be proved, and how is it proved?

ML with exceptions

With exceptions (and no other features) we can do it as follows:

```
exception Abort

let mu_exc f a =
  let rec search k =
    try
      let a' n = (if n < k then a n else raise Abort) in
        f a' ; k
    with Abort -> search (k+1)
  in
    search 0
```

However:
1. What if f catches `Abort`? May it do so? What is the exact precondition on f?
2. Would local exceptions help? If so, can f use its own local exceptions?

Other setups

1. In Haskell we could do everything inside a fixed monad. This is still not entirely easy, even if we figure out what it means for f to be "pure".
2. Moving to a total language is probably helpful. However, keep in mind that μ does not exist in pure λ-calculus, so straight Agda or some such system is out of the question.
3. Other effects can be used to implement a candidate μ, but it seems like they should be *local* (local references, local exceptions, delimited control) or else f has access to them.

5.1.3 Open problem

At first sight it seems that the above implementations of μ work, but as soon as we try to formulate exactly what it is that we want to prove, it becomes clear that not everything is clear, so the first problem is:

> *Explain what it means to realize "all functions are continuous" in a realizability model based on a programming language with computational effects.*

One has to find a good notion of a realizer that uses effects in a "benign way". For instance, asking for purity in the sense of [1] seems too restrictive. Once it is clear what problem we are trying to solve, we may attempt to prove that the modulus is really there:

> *Identify computational effects which allow realization of the modulus of continuity, and prove* rigorously *that the realizer works.*

Attacking the problem ought to improve our ability to argue about higher-type computation in the presence of computational effects.

References

1 Andrej Bauer, Martin Hofmann and Aleksandr Karbyshev. On Monadic Parametricity of Second-Order Functionals. *Foundations of Software Science and Computation Structures – 16th International Conference, FOSSACS 2013*, 225–240, 2013.

2 Ulrich Berger. Computability and Totality in Domains. *Mathematical Structures in Computer Science* 12(3), 281–294, 2002.

3 Anne Troelstra and Dirk van Dalen. Constructivism in mathematics, volume 2. Elsevier, 1988.

5.2 Capturing algebraic equations in an effect system

Matija Pretnar (University of Ljubljana, SI)

Equational theories

The main premise of algebraic effects is that effects can be described with an equational theory consisting of a set of operations and equations between them [7]. For example, non-determinism can be described by an operation `choose` and three equations stating its idempotency, commutativity and associativity. Computations returning values from X are then interpreted as elements of the *free model* of such a theory.

Issues with interpreting handlers

Handlers of algebraic effects, which assign a handling term for each operation, can be interpreted as homomorphisms from the free model to some other (not necessary free) model of the same theory [8]. However, there are computationally interesting handlers that do not respect all of the expected equations. One is a handler that collects all possible results of a non-deterministic computation in a list. This respects the associativity, but not the idempotency or commutativity of `choose`. Similarly, a state handler that logs all memory updates handles a computation that sequentially writes two values differently than one that writes only the second one, even though these two computations are often considered equivalent [6].

Since handlers that do not respect the equations cannot receive an algebraic interpretation [8], some recent work [1, 4] assumes no non-trivial equations to hold, giving up most of the existing results on combining algebraic theories [3] and optimizations [5].

Extending types with equations

A possible way of resolving this issue is to capture the subset of valid equations in types. An algebraic approach already has a natural effect system, in which computations receive a type $A!\mathcal{O}$, where A is the type of returned values, and $\mathcal{O}$ is the set of operations that may get called [1, 4]. For example, a non-deterministic computation returning integers would be given the type $\texttt{int}!\{\texttt{choose}\}$, while a pure computation would have the type $\texttt{int}!\emptyset$.

This description can be extended to one of the form $A!\mathcal{O}\&\mathcal{E}$, where $\mathcal{E}$ is a now the subset of equations we assume to hold between operations $\mathcal{O}$. This type may be interpreted as the free model of the theory with the same operations, but with equations only from $\mathcal{E}$. For example, `if choose () then 1 else 2` and `if choose () then 2 else 1` can be considered as equivalent computations of type $\texttt{int}!\{\texttt{choose}\}\&\{\texttt{comm}\}$, but not of type $\texttt{int}!\{\texttt{choose}\}\&\{\texttt{assoc}\}$. This generalizes both the traditional approach to algebraic effects, if one considers $\mathcal{E}$ to be the set of all equations in the theory, or the approach with no equations, if $\mathcal{E} = \emptyset$.

Similar interpretation applies to handlers of type $A_1!\mathcal{O}_1\&\mathcal{E}_1 \Rightarrow A_2!\mathcal{O}_2\&\mathcal{E}_2$, where $\mathcal{E}_1$ is now the set of equations the handler must respect. For example, the handler

```
let choose_left = handler
  | choose () k -> k true
```

which makes `choose` constantly yield `true` in the handled computation, can be given the type $A!\{\texttt{choose}\}\&\{\texttt{assoc}, \texttt{idem}\} \Rightarrow A!\emptyset\&\emptyset$. Next, the handler

```
let choose_all = handler
  | choose () k -> (k true) @ (k false)
  | val x -> [x]
```

which returns the list of all possible results of the handled computation, can be given the type $A!\{\texttt{choose}\}\&\{\texttt{assoc}\} \Rightarrow A\,\text{list}!\emptyset$. Finally, the handler

```
let choose_sum = handler
  | choose () k -> (k true) + (k false)
```

which returns the sum of all possible results of the handled computation, can be given the type $\texttt{int}!\{\texttt{choose}\}\&\{\texttt{assoc}, \texttt{comm}, \texttt{idem}\} \Rightarrow \texttt{int}!\emptyset$.

The equations expected for the domain of the handler can also depend on the ones holding for the codomain. For example, one expects the handler

```
let choose_opposite = handler
  | choose () k -> if choose () then (k false) else (k true)
```

to have the type $A!\{\texttt{choose}\}\&\mathcal{E} \Rightarrow A!\{\texttt{choose}\}\&\mathcal{E}$
for any set of equations $\mathcal{E} \subseteq \{\texttt{assoc}, \texttt{comm}, \texttt{idem}\}$.

Open questions

Exact typing rules. When a computation may receive an enriched type remains to be determined. One may expect rules such as

$$\frac{\Gamma \vdash c : A!\mathcal{O}\&\mathcal{E} \qquad \mathcal{E} \subseteq \mathcal{E}'}{\Gamma \vdash c : A!\mathcal{O}\&\mathcal{E}'}$$

as we may always consider additional equivalences between programs to hold. The most involved rule seems to be one for assigning a type $A_1!\mathcal{O}_1\&\mathcal{E}_1 \Rightarrow A_2!\mathcal{O}_2\&\mathcal{E}_2$ to a handler. Here, we must check that the given handler respects all the equations $\mathcal{E}_1$, probably in a similar way as checking whether a handler is correct [8]. Since the equations describe the properties of effects on the level of algebraic theories, we can expect the resulting type system to be simpler than one involving dependent types or refinement types, however one must bear in mind that determining whether a handler respects a given set of equations is undecidable [8].

Applications

Handlers provide a very powerful control mechanism, which can dynamically change the context in which programs are run. One potential application of the described approach is to at least partially convey information about this behaviour through equations. The equations could also be used for enforcing behaviour. Even though determining their validity is undecidable, one could take a tool such as QuickCheck [2], which verifies properties of pure values by generating random tests, and extend it to testing impure computations.

Another prospective application is modular reasoning about handlers. For example, one can show that the usual monadic state handler satisfies certain properties [1], but the exact proof works only for the particular handler and needs to be redone for a different implementation. With equations in types, one could split the reasoning into two parts: (1) showing that a handler respects certain equations and has a given type, and (2) showing that any handler with that type satisfies a given property.

References

1 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.
2 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279. ACM, 2000.
3 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.
4 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, pages 145–158. ACM, 2013.
5 Ohad Kammar and Gordon D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*, pages 349–360. ACM, 2012.
6 Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FoSSaCS*, volume 2303 of *LNCS*, pages 342–356. Springer, 2002.
7 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
8 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

Participants

- Sandra Alves
University of Porto, PT
- Kenichi Asai
Ochanomizu Univ. – Tokyo, JP
- Robert Atkey
University of Strathclyde – Glasgow, GB
- Clément Aubert
Appalachian State University – Boone, US
- Andrej Bauer
University of Ljubljana, SI
- Edwin Brady
University of St. Andrews, GB
- Xavier Clerc
Apimka – Paris, FR
- Stephen Dolan
University of Cambridge, GB
- Andrzej Filinski
University of Copenhagen, DK
- Philipp Haselwarter
University of Ljubljana, SI
- Martin Hofmann
LMU München, DE
- Patricia Johann
Appalachian State University – Boone, US
- Yukiyoshi Kameyama
University of Tsukuba, JP
- Ohad Kammar
University of Cambridge, GB
- Oleg Kiselyov
Tohoku University – Sendai, JP
- Daan Leijen
Microsoft Res. – Redmond, US
- Sam Lindley
University of Edinburgh, GB
- Conor McBride
University of Strathclyde – Glasgow, GB
- Gordon Plotkin
University of Edinburgh, GB
- Matija Pretnar
University of Ljubljana, SI
- Amr Hany Shehata Saleh
KU Leuven, BE
- Gabriel Scherer
Northeastern University – Boston, US
- Tom Schrijvers
KU Leuven, BE
- Alex Simpson
University of Ljubljana, SI
- KC Sivaramakrishnan
University of Cambridge, GB
- Sam Staton
University of Oxford, GB
- Niki Vazou
University of California – San Diego, US
- Niels Voorneveld
University of Ljubljana, SI
- Leo White
Jane Street – London, GB
- Jeremy Yallop
University of Cambridge, GB