Report from Dagstuhl Seminar 18172

Algebraic Effect Handlers go Mainstream

Edited by

Sivaramakrishnan Krishnamoorthy Chandrasekaran[1], Daan Leijen[2], Matija Pretnar[3], and Tom Schrijvers[4]

1 **University of Cambridge, GB,** `sk826@cl.cam.ac.uk`
2 **Microsoft Research – Redmond, US,** `daan@microsoft.com`
3 **University of Ljubljana, SI,** `matija.pretnar@fmf.uni-lj.si`
4 **KU Leuven, BE,** `tom.schrijvers@cs.kuleuven.be`

Abstract

Languages like C#, C++, or JavaScript support complex control flow statements like exception handling, iterators (yield), and even asynchrony (async/await) through special extensions. For exceptions, the runtime needs to be extended with exception handling stack frames. For iterators and asynchrony, the situation is more involved, as the compiler needs to turn regular code into stack restoring state machines. Furthermore, these features need to interact as expected, e.g. finally blocks must not be forgotten in the state machines for iterators. And all of this work needs to be done again for the next control flow abstraction that comes along.

Or we can use algebraic effect handlers! This single mechanism generalizes all the control flow abstractions listed above and more, composes freely, has simple operational semantics, and can be efficiently compiled, since there is just one mechanism that needs to be supported well. Handlers allow programmers to keep the code in direct-style, which is easy to reason about, and empower library writers to implement various high-level abstractions without special extensions.

The idea of algebraic effects handlers has already been experimented with in the form of small research languages and libraries in several mainstream languages, including OCaml, Haskell, Clojure, and Scala. The next step, and the aim of this seminar, is to seriously consider adoption by mainstream languages including both functional languages such as OCaml or Haskell, as well as languages like JavaScript and the JVM and .NET ecosystems.

Seminar April 22–27, 2018 – http://www.dagstuhl.de/18172

2012 ACM Subject Classification Software and its engineering → Control structures, Software and its engineering → Formal methods, Software and its engineering → Semantics

Keywords and phrases algebraic effect handlers, implementation techniques, programming abstractions, programming languages

Digital Object Identifier 10.4230/DagRep.8.4.104

Edited in cooperation with Jonathan Immanuel Brachthäuser

1 Executive Summary

Sivaramakrishnan Krishnamoorthy Chandrasekaran (University of Cambridge, GB)
Daan Leijen (Microsoft Research – Redmond, US)
Matija Pretnar (University of Ljubljana, SI)
Tom Schrijvers (KU Leuven, BE)

Algebraic effects and their handlers have been steadily gaining attention as a programming language feature for composably expressing user-defined computational effects. Algebraic effect handlers generalise many control-flow abstractions such as exception handling, iterators, async/await, or backtracking, and in turn allow them to be expressed as libraries rather than implementing them as primitives as many language implementations do. While several prototype languages that incorporate effect handlers exist, they have not yet been adopted into mainstream languages. This Dagstuhl Seminar 18172 "Algebraic Effect Handlers Go Mainstrea" touched upon various topics that hinder adoption into mainstream languages. To this end, the participants in this seminar included a healthy mix of academics who study algebraic effects and handlers, and developers of mainstream languages such as Haskell, OCaml, Scala, WebAssembly, and Hack.

This seminar follows the earlier, wildly successful Dagstuhl Seminar 16112 "From Theory to Practice of Algebraic Effects and Handlers" which was dedicated to addressing fundamental issues in the theory and practice of algebraic effect handlers. We adopted a similar structure for this seminar. We had talks each day in the morning, scheduled a few days ahead. The folks from the industry were invited to present their perspectives on some of the challenges that could potentially be address with the help of effect handlers. The afternoons were left free for working in self-organised groups and show-and-tell sessions with results from the previous days. We also had impromptu lectures on the origins of algebraic effects and handlers, which were quite well received and one of the highlights of the seminar.

Between the lectures and working-in-groups, the afternoons were rather full. Hence, a few participants offered after-dinner "cheesy talks" just after the cheese was served in the evening. The participants were treated to entertaining talks over delightful cheese and fine wine. We encourage the organisers to leave part of the day unplanned and go with what the participants feel like doing on that day. The serendipitous success are what makes Dagstuhl Seminars special.

We are delighted with the outcome of the seminar. There were interesting discussions around the problem of *encapsulation* and leaking of effects in certain higher order use cases, with several promising solutions discussed. It was identified that the problem of encapsulation and leaking effect names is analogous to the name binding in lambda calculus. Another group made significant progress in extending WebAssembly with support for effect handlers. The proposal builds on top of support for exceptions in WebAssembly. During the seminar week, the syntax extensions and operational semantics were worked out, with work begun on the reference implementation. During the seminar, Andrej Bauer pointed out that several prototype implementations that incorporate effect handlers exist, each with their own syntax and semantics. This makes it difficult to translate ideas across different research groups. Hence, Andrej proposed and initiated effects and handlers rosetta stone – a repository of examples demonstrating programming with effects and handlers in various programming languages. This repository is hosted on GitHub and has had several contributions during and after the seminar.

In conclusion, the seminar inspired discussions and brought to light the challenges in incorporating effect handlers in mainstream languages. During the previous seminar (16112), the discussions were centered around whether it was even possible to incorporate effect handlers into mainstream languages. During this seminar, the discussions were mainly on the ergonomics of effect handlers in mainstream languages. This is a testament to the success of the Dagstuhl Seminars in fostering cutting edge research.

2 Contents

3 Overview of Talks

3.1 Linking Types for Multi-Language Software

Amal Ahmed (Northeastern University – Boston, US)

License Creative Commons BY 3.0 Unported license
© Amal Ahmed
Main reference Daniel Patterson, Amal Ahmed: "Linking Types for Multi-Language Software: Have Your Cake and Eat It Too", in Proc. of the 2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA, LIPIcs, Vol. 71, pp. 12:1–12:15, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.
URL https://doi.org/10.4230/LIPIcs.SNAPL.2017.12

In the last few years, my group at Northeastern has focused on verifying compositional compiler correctness for today's world of multi-language software. Such compilers should allow compiled components to be linked with target components potentially compiled from other, very different, languages. At the same time, compilers should also be fully abstract: that is, they should ensure that if two components are equivalent in all source contexts then their compiled versions are equivalent in all target contexts. Fully abstract compilation allows programmers to reason about their code (e.g., about correctness of refactoring) by only considering interactions with other code from the same language. While this is obviously an extremely valuable property for compilers, it rules out linking with target code that has features or restrictions that can not be represented in the source language that is being compiled.

While traditionally fully abstract compilation and flexible linking have been thought to be at odds, I'll present a novel idea called Linking Types [1] that allows us to bring them together by enabling a programmer to opt into local violations of full abstraction only when she needs to link with particular code without giving up the property globally. This fine-grained mechanism enables flexible interoperation with low-level features while preserving the high-level reasoning principles that fully abstract compilation offers.

An open question is whether algebraic effects and effect handlers might be an effective way of designing linking-types extensions for existing languages.

References

1 Daniel Patterson and Amal Ahmed. *Linking Types for Multi-Language Software: Have Your Cake and Eat It Too.* In Summit on Advances in Programming Languages (SNAPL), 2017.

3.2 Idealised Algol

Robert Atkey (University of Strathclyde – Glasgow, GB)

License Creative Commons BY 3.0 Unported license
© Robert Atkey
Joint work of Robert Atkey, Michel Steuwer, Sam Lindley, Christophe Dubach
Main reference Robert Atkey, Michel Steuwer, Sam Lindley, Christophe Dubach: "Strategy Preserving Compilation for Parallel Functional Code", CoRR, Vol. abs/1710.08332, 2017.
URL https://arxiv.org/abs/1710.08332

Idealised Algol was introduced by John Reynolds in the late 1970s as the orthogonal combination of λ-calculus and imperative programming. The resulting language is an elegant combination of imperative programming (variables, while loops, etc.) and procedures (provided by λ-abstraction). A variant of Idealised Algol, called Syntactic Control of

Interference, provides a way to banish aliasing within the language, and hence to provide a way to express race free parallelism.

In this talk, I discussed the similarities between Idealised Algol's expressive handling of variables and mutable state, in particular Reynolds' conception of variables as“objects” consisting of a getter and a setter, and handlers for algebraic effects. Idealised Algol appears to naturally include a particularly efficient subset of handlers: linear handlers (the continuation must always be used), that are tail recursive. By sticking to this subset, it is possible to generate efficient code without expensive stack manipulation.

Some of this work is joint with Sam Lindley, Michel Steuwer and Christophe Dubach, where we have applied Idealised Algol with interference control to the problem of generating code from functional specifications for execution on parallel computing hardware, such as multicore processors and GPUs.

3.3 What is algebraic about algebraic effects and handlers?

Andrej Bauer (University of Ljubljana, SI)

In this tutorial we reviewed the classic treatment of algebraic theories and their models in the category of sets. We then drew an uninterrupted line of thought from the classical theory to algebraic effects and handlers. First we generalized operations with integral arities to parameterized operations with arbitrary arities, as these are needed for modeling computational effects. The free models of theories with generalized operations can be used as denotations of effectful programs. The universal property of a free models can be used to derive the notion of handlers. At the level of types, the value types correspond to sets of generators and the computation types to the free models. The naive set-theoretic treatment presented in the tutorial should be replaced with a domain-theoretic one if we wanted adequate denotational semantics of a realistic programming language with general recursion. The contents of the tutorial has been written up an extended in [1].

References

1 Andrej Bauer. What is algebraic about algebraic effects and handlers? *ArXiv e-print 1807.05923*, 2018. https://arxiv.org/abs/1807.05923.

3.4 Event Correlation with Algebraic Effects

Oliver Bracevac (TU Darmstadt, DE)

Joint work of Oliver Bracevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, Mira Mezini
Main reference Oliver Bracevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, Mira Mezini, “Versatile Event Correlation with Algebraic Effects”, Proc. ACM Program. Lang., Vol. 2, pp. 67:1–67:31, ACM, 2018.
URL http://dx.doi.org/10.1145/3236762

This talk presents a language design on top of algebraic effects and handlers for defining n-way joins over asynchronous event sequences. The design enables mix-and-match-style compositions of join variants from different domains, e.g., stream-relational algebra, event processing, reactive and concurrent programming, where joins are defined in direct style pattern notation. Their matching behavior is programmable via dedicated control abstractions for coordinating and aligning asynchronous streams. Our insight is that semantic variants of joins are definable as cartesian product computations with side effects influencing how the computation unravels. Based on this insight, we can afford working with a naive enumeration procedure of the cartesian product and turn it into efficient variants, by injection of appropriate effect handlers.

3.5 Effect Handlers for the Masses

Jonathan Immanuel Brachthäuser (Universität Tübingen, DE)

Joint work of Jonathan Immanuel Brachthäuser, Philipp Schuster, Klaus Ostermann
Main reference Jonathan Immanuel Brachthäuser, Philipp Schuster, Klaus Ostermann, "Effect Handlers for the Masses", under submission at the Conference on Object Oriented Programming Systems Languages & Applications, Boston, USA, 2018.

Algebraic effect handlers are a program structuring paradigm with rising popularity in the functional programming language community. Effect handlers are less wide-spread in the context of imperative, object oriented languages. We present library implementations of algebraic effects in Scala and Java. Both libraries are centered around the concept of handler/capability passing and a shallow embedding of effect handlers. While the Scala library is based on a monad for multi-prompt delimited continuations, the Java library performs a CPS translation as bytecode instrumentation. We discuss design decisions and implications on extensibility and performance.

References

1 Jonathan Immanuel Brachthäuser, Philipp Schuster. *Effekt: Extensible Algebraic Effects in Scala*. Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, Vancouver, Canada, 2017
2 Jonathan Immanuel Brachthäuser, Philipp Schuster. *Effect Handlers for the Masses*. Under submission at the Conference on Object Oriented Programming Systems Languages & Applications, Boston, USA, 2018

3.6 Combining Algebraic Theories

Jeremy Gibbons (University of Oxford, GB)

Joint work of Kwok Cheung, Jeremy Gibbons

I summarized results due to Hyland, Plotkin, and Power [5, 6] on combining algebraic theories, as explored in the doctoral thesis [2] of my student Kwok Cheung. Specifically, the *sum* $S + T$ of algebraic theories S and T has all the operations of S and T, and all the equations, and no other equations. The *commutative tensor* $S \otimes T$ adds equations of the form

$$f(g(x_1, x_2), g(y_1, y_2), g(z_1, z_2)) = g(f(x_1, y_1, z_1), f(x_2, y_2, z_2))$$

for each operation f of one theory and g of the other. The *distributive tensor* $S \triangleright T$ adds to the sum equations of the form

$$\begin{array}{rcl} f(g(x_1,x_2),y,z) & = & g(f(x_1,y,z),f(x_2,y,z)) \\ f(x,g(y_1,y_2),z) & = & g(f(x,y_1,z),f(x,y_2,z)) \\ f(x,y,g(z_1,z_2)) & = & g(f(x,y,z_1),f(x,y,z_2)) \end{array}$$

for each operation f of S and g of T.

I also showed some examples:

- *global state* $S \to (1 + -) \times S$ arises as the sum of the theories *State* and *Failure*;
- *local state* $S \to 1 + (- \times S)$ arises as the commutative tensor *State* $\otimes$ *Failure*;
- probability and nondeterminism interact via probabilistic choice ${}_w\oplus$ distributing over nondeterministic choice $\Box$, but not vice versa, so arises as the distributive tensor *Prob* $\triangleright$ *Nondet*;
- two-player games have both angelic choice $\sqcup$ and demonic choice $\sqcap$, each of which distributes over the other, so arises as the two-way distributive tensor *Nondet* $\triangleleft\triangleright$*Nondet*

and some non-examples:

- the *list monad* does not arise as any of these combinations of the theories *Nondet* (i.e., just binary choice, with associativity) and *Failure*;
- the *list transformer done right* [3] applied to the monad arising from some theory T does not arise as any of these combinations of the monoidal theory of the list monad with T;
- symmetric bidirectional transformations [4] maintain two complementary data sources A and B in synchronization; they have the signature of two copies of the theory of *State*, but the two implementations are *entangled* [1]—the ‘get’ operations of A and B commute with each other, but the ‘set’ operation on one side does not commute with any operation on the the other.

I conjecture there are more such well-behaved and useful combinators on algebraic theories to be found, which might explain the above counter-examples and others like them.

References

1 Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 187–214. Springer, 2015.

2 Kwok Cheung. *Distributive Interaction of Algebraic Effects.* Dphil thesis, University of Oxford, 2018.

3 Yitzhak Gale. ListT done right. https://wiki.haskell.org/ListT_done_right, 2007.

4 Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. Symmetric lenses. In Thomas Ball and Mooly Sagiv, editors, *Principles of Programming Languages*, pages 371–384. ACM, 2011.

5 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer. Science*, 357(1-3):70–99, 2006.

6 Martin Hyland and John Power. Discrete Lawvere theories and computational effects. *Theoretical Computer Science*, 366(1-2):144–162, 2006.

3.7 Handlers.Js: A Comparative Study of Implementation Strategies for Effect Handlers on the Web

Daniel Hillerström (University of Edinburgh, GB)

Joint work of Sam Lindley, Robert Atkey, KC Sivaramakrishnan, Jeremy Yallop

Handlers for algebraic effects have steadily been gaining traction since their inception. This traction can be partly attributed to their wide application space which includes diverse programming disciplines such as concurrent programming, probabilistic programming, etc. They have been implemented in a variety of programming languages as either a native primitive or embedded via existing abstraction facilities.

The many different implementations have contributed to mapping out the implementation space for effect handlers. While the picture for how to implement effect handlers in native code is pretty clear, the picture for implementing effect handlers via embedding in high-level programming languages is more blurry. Embedding in JavaScript is currently the only viable option for implementing effect handlers on the Web.

In this talk, I will discuss and compare five viable different compilation strategies for effect handlers to JavaScript. Two of the five strategies are based on novel encodings via generators/iterators and generalised stack inspection, respectively. Although, I will discuss these compilation strategies in the context of JavaScript, they are not confined to JavaScript. The strategies are also viable in other high-level languages such as, say, Python.

3.8 First Class Dynamic Effect Handlers and Deep Finally Handling

Daan Leijen (Microsoft Research – Redmond, US)

Main reference Daan Leijen, “First Class Dynamic Effect Handlers”, in TyDe’18, St. Louis, US, Sep. 2018.
URL https://www.microsoft.com/en-us/research/publication/first-class-dynamic-effect-handlers

We first show how “inject” combined with higher-ranked types can encode first-class polymorphic references. However, it is cumbersome to program this way as you need to “inject” carefully to select the right handler by position. To remedy this, we extend basic algebraic effect handlers with *first class dynamic effects* to refer to a handler directly by name. Dynamic effects add a lot more expressiveness but surprisingly only need minimal changes to the original semantics. As such, dynamic effects are a powerful abstraction but can still be understood and reasoned about as regular effect handlers. We illustrate the expressiveness of dynamic effects with first class event streams in CorrL and also model full polymorphic heap references without requiring any further primitives. Following this, we add “finally” and “initially” clauses to handlers to robustly deal with external resources. We show you generally need a form of “deep” finally handling to reliably invoke all outstanding “finally” clauses.

Note: the original title of the talk was “Algebraic Effects with Resources and Deep Finalization”; However, it was decided afterwards this naming caused too much confusion: “Resources” was already used for external resources in Matija’s thesis, and “Finalization” reminded of finalization in the object oriented world which is invoked by the GC and non-deterministic.

3.9 Encapsulating effects

Sam Lindley (University of Edinburgh, GB)

3.9.1 Leaking effects

Naively composing effect handlers that produce and consume an intermediate effect leads to that effect leaking such that external instances are accidentally captured. I illustrate the problem in Frank and then show how Biernacki et al.'s lift operator resolves the issue. I then briefly discuss other possible solutions.

For the following, I assume basic familiarity with the Frank programming language [6]. Let us begin by defining in Frank a maybe data type, reader and abort interfaces along with effect handlers for them.

```
data Maybe X = nothing | just X

interface Reader X = ask : X
interface Abort = abort X : X

reads : {List S -> <Reader S>X -> [Abort]X}
reads []       <ask -> k> = abort!
reads (s :: ss) <ask -> k> = reads ss (k s)
reads _        x          = x

maybe : {<Abort>X -> Maybe X}
maybe <abort -> _> = nothing
maybe x            = just x
```

The `reads` handler interprets the `ask` command by reading the next value, if there is one, from the supplied list; if the list is empty then it raises the `abort` command. The `maybe` handler interprets `abort` using the maybe data type.

It seems natural to want to precompose `maybe` with `reads`. A naive attempt yields the following Frank code.

```
bad : {List S -> <Reader S, Abort>X -> Maybe X}
bad ss <m> = maybe (reads ss m!)
```

The `bad` handler handles `ask` as expected, yielding `just v` for some value `v` if input is not exhausted

```
bad [1,2,3] (ask! + ask! + ask!) == just 6
```

and `nothing` if input is exhausted:

```
bad [1,2] (ask! + ask! + ask!) == nothing
```

Alas, as indicated by its type, `bad` also exhibits additional behaviour. As well as handling any `abort` command raised by the `reads` handler, it also captures uses of `abort` from the ambient context:

```
bad [1,2,3] (ask! + ask! + abort!) == nothing
```

One might think that we could simply supply a different type signature to `bad` in order to suppress `Abort`. But the underlying problem is not with the types; it is with the dynamic semantics. Without some way of hiding the `Abort` effect there is no way of preventing the `maybe` handler from accidentally capturing any `abort` command raised by the ambient context.

The current version of Frank (April 2018) provides a solution to the effect encapsulation problem: Biernacki et al.'s `lift` operator [3], with which we can precompose `maybe` with `reads` as follows.

```
good : {List S -> <Reader S>X -> Maybe X}
good ss <m> = maybe (reads ss (lift <Abort> m!))
```

An effect (*ability* in Frank parlance), in Frank (much like Koka [5]) denotes a total map from interface names to finite lists of instantiations. Interfaces that are not present are denoted by empty lists. The head of a list denotes the active instantiation of an interface. The `lift` operator adds a dummy instantiation onto the head of the list associated with a given interface. Thus, the invocation `lift <Abort> m!` adds a dummy instantiation of `Abort` onto the ability associated with the computation `m!`. The dummy instantiation ensures that the `maybe` handler cannot accidentally capture `abort` commands raised by the ambient context. So

```
good [1,2,3] (ask! + ask! + abort!) : [Abort]Maybe Int
```

and:

```
maybe (good [1,2,3] (ask! + ask! + abort!)) == nothing
```

The `lift` operator provides a means for hiding effects reminiscent of de Bruijn representations for bound names. It generates a fresh instantiation of an interface by shifting all of the existing instances along by one.

3.9.2 Concurrency

In his Master's dissertation, Lukas Convent identified the effect encapsulation problem in the context of a previous version of Frank [4] that did not provide support for `lift`. He presents a number of examples of trying to compose effect handlers together in order to implement various forms of concurrency. The resulting types expose many of the internal implementation details illustrating how important the problem is to solve.

To illustrate how `lift` helps to solve these problems I include an adaptation of code from Convent's thesis to implement Erlang-style concurrency based on an actor abstraction in Frank by composing together a number of different handlers. The adapted code takes advantage of `lift`.

```
include prelude

--------------------------------------------------------------------------------
-- Queue interface and FIFO implementation using a zipper
--------------------------------------------------------------------------------

interface Queue S = enqueue : S -> Unit
                  | dequeue : Maybe S

-- zipper queue
data ZipQ S = zipq (List S) (List S)
```

```
emptyZipQ : {ZipQ S}
emptyZipQ! = zipq [] []

-- FIFO queue implementation using a zipper
-- (returns the remaining queue alongside the final value)
runFifo : {ZipQ S -> <Queue S>X -> Pair X (ZipQ S)}
runFifo (zipq front back)       <enqueue x -> k> = runFifo (zipq front (x :: back)) (k unit)
runFifo (zipq []     [])        <dequeue -> k>   = runFifo emptyZipQ! (k nothing)
runFifo (zipq []     back)      <dequeue -> k>   = runFifo (zipq (rev back) []) (k dequeue!)
runFifo (zipq (x :: front) back) <dequeue -> k>  = runFifo (zipq front back) (k (just x))
runFifo queue                   x                = pair x queue

-- discard the queue
evalFifo : {<Queue S>X -> ZipQ S -> X}
evalFifo <t> q = case (runFifo q t!) { (pair x _) -> x }

-- start with an empty queue
fifo : {<Queue S>X -> X}
fifo <m> = evalFifo m! (emptyZipQ!)

-- discard the value
execFifo: {<Queue S>X -> ZipQ S -> ZipQ S}
execFifo <t> q = case (runFifo q t!) { (pair _ q) -> q }

--------------------------------------------------------------------------------
-- Definitions of interfaces, data types
--------------------------------------------------------------------------------

interface Co = fork  : {[Co]Unit} -> Unit
             | yield : Unit

data Mailbox X = mbox (Ref (ZipQ X))

interface Actor X = self    : Mailbox X
                  | spawn Y : {[Actor Y]Unit} -> Mailbox Y
                  | recv    : X
                  | send Y  : Y -> Mailbox Y -> Unit

data WithSender X Y = withSender (Mailbox X) Y

--------------------------------------------------------------------------------
-- Example actor
--------------------------------------------------------------------------------

spawnMany : {Mailbox Int -> Int -> [Actor Int [Console], Console]Unit}
spawnMany p 0 = send 42 p
spawnMany p n = spawnMany (spawn {let x = recv! in print "."; send x p}) (n-1)

chain : {[Actor Int [Console], Console]Unit}
chain! = spawnMany self! 640; recv!; print "\n"

------------------------------------------------------------------------------
-- Implement an actor computation as a stateful concurrent computation
------------------------------------------------------------------------------

-- Our syntactic sugar assumes that all instances of the implicit
-- effect variable are instantiated to be the same but they needn't be
-- the same as the ambient effects.
--
-- For liftBody we need exactly that all but the ambient effects be
-- the same.
liftBody : {{[Actor X]Unit} -> [E |]{[Actor X, Co [RefState], RefState]Unit}}
liftBody m = {lift <RefState, Co> m!}


act : {Mailbox X -> <Actor X>Unit -> [Co [RefState], RefState]Unit}
act mine     <self -> k> = act mine (k mine)
act mine     <spawn you -> k> = let yours = mbox (new (emptyZipQ!)) in
                                fork {act yours (liftBody you)!};
                                act mine (k yours)
act (mbox m) <recv -> k> = case (runFifo (read m) dequeue!)
                                { (pair nothing _)  -> yield!;
                                                       act (mbox m) (k recv!)
                                | (pair (just x) q) -> write m q;
                                                       act (mbox m) (k x) }
act mine     <send x (mbox m) -> k> = let q = execFifo (enqueue x) (read m) in
                                      write m q;
                                      act mine (k unit)
act mine     unit = unit

runActor : {<Actor X>Unit -> [RefState]Unit}
runActor <m> = bfFifo (act (mbox (new emptyZipQ!)) (lift <Co> m!))

bfFifo : {<Co>Unit -> Unit}
bfFifo <m> = fifo (scheduleBF (lift <Queue> m!))

------------------------------------------------------------------------------
-- Scheduling
------------------------------------------------------------------------------

data Proc = proc {[Queue Proc]Unit}

enqProc : {[Queue Proc]Unit} -> [Queue Proc]Unit
enqProc p = enqueue (proc p)

runNext : {[Queue Proc]Unit}
runNext! = case dequeue! { (just (proc x)) -> x!
                         | nothing         -> unit }

-- defer forked processes (without effect pollution)
scheduleBF : {<Co>Unit -> [Queue Proc]Unit}
scheduleBF <yield -> k>  = enqProc {scheduleBF (k unit)};
                           runNext!
scheduleBF <fork p -> k> = enqProc {scheduleBF (lift <Queue> p!)};
                           scheduleBF (k unit)
scheduleBF unit          = runNext!

-- eagerly run forked processes
scheduleDF : {<Co>Unit -> [Queue Proc]Unit}
scheduleDF <yield -> k>  = enqProc {scheduleDF (k unit)};
                           runNext!
scheduleDF <fork p -> k> = enqProc {scheduleDF (k unit)};
                           scheduleDF (lift <Queue> p!)
scheduleDF unit          = runNext!

main : {[Console, RefState]Unit}
main! = runActor (lift <RefState> chain!)
```

This code implements actors using a coroutining concurrency interface, which in turn is implemented using an interface for queues of processes, which are implemented using a simple zipper data structure. The example `chain` spawns a collection of processes and passes a message through the entire collection.

The crucial point is that the type of `runActor` mentions only the `Actor` interface that is being handled and the `RefState` interface that is being used to implement it. Convent's original code leaks out the `Queue` interface and the `Co` interface. Moreover, the type becomes particularly hard to read because some of the interfaces are parameterised by several effects.

3.9.3 Discussion

The designers of the Eff programming language [2] have explored several different designs for effect handlers and effect type systems for effect handlers. Some versions of Eff provide features that address the effect encapsulation problem to some degree. The earliest version of Eff [1] resolved the encapsulation problem by supporting the generation of fresh instances of an effect. This was relatively straightforward as at the time Eff did not provide an effect type system. A later version of Eff included a somewhat complicated effect type system with support for instances that used a region-like effect type system [7]. The latest version of Eff at the time of writing [8] does not appear to offer a solution to the effect encapsulation problem. It provides an effect type system with subtyping, without duplicate effect interfaces and no support for effect instances.

For effect systems that allow only one copy of each effect interface we might solve the effect encapsulation problem by adding a primitive for introducing a fresh copy of an interface. For instance, to hide the intermediate `Abort` interface we could generate a fresh copy of `Abort` – call it `Abort'` – which we could then use to rename any `abort` commands in the ambient context to `abort'` before renaming them back after running the `reads` and `maybe` handlers.

An as yet unimplemented Frank feature that is related to effect encapsulation is negative adjustments [6]. Currently an adjustment in Frank always adds interfaces to the ambient ability, and specifies which interfaces must be handled. Negative adjustments would allow interfaces to be removed, enabling the programmer to specify that a computation being handled does not support some of the interfaces in the ambient. Roughly, the `lift` operation is the inverse of a negative adjustment. It remains to be seen what the relative pros and cons of negative adjustments and `lift` are in practice.

More generally, there is a broad design space for effect type systems that support effect encapsulation and it is worth considering the full range of options.

References

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

2 Andrej Bauer and Matija Pretnar. Eff, 2018.

3 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL*, 2(POPL):8:1–8:30, 2018.

4 Lukas Convent. Enhancing a modular effectful programming language. Master's thesis, The University of Edinburgh, Scotland, 2017.

5 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *POPL*, pages 486–499. ACM, 2017.

6 Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *POPL*, pages 500–514. ACM, 2017.

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

8 Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. Explicit effect subtyping. In *ESOP*, volume 10801 of *Lecture Notes in Computer Science*, pages 327–354. Springer, 2018.

3.10 Experiences with structuring effectful code in Haskell

Andres Löh (Well-Typed LLP, DE)

Effectful Haskell code that is written using monad transformers can easily become difficult to maintain. However, it is unclear whether algebraic effects doenot suffer from the same problem. Both approaches seem to encourage specifying a minimal amount of effects for each code fragment, leading to effect constraints being propagated in a bottom-up fashion throughout the program, often without much thought for control. In this talk, I argue that sometimes, it is better to be less general, by identifying just a few meaningful interfaces in a program, corresponding to different sets of available effects, and keeping testing in mind. These interfaces are then pushed down, and code is merely checked to not use effects that are outside of the allowed subset.

3.11 Make Equations Great Again!

Matija Pretnar (University of Ljubljana, SI)

Joint work of Žiga Lukšič, Matija Pretnar

Algebraic effects have originally been presented with equational theories, i.e. a set of operations and a set of equations they satisfy. Since a significant portion of computationally interesting handlers overrides the effectful behaviour in a way that invalidates the equations, most approaches nowadays assume an empty set of equations.

At the Dagstuhl Seminar 16112, I presented an idea in which the equations are represented locally in computation types [1]. In this way, handlers that do not respect all equations are not rejected but receive a weaker type. In the talk, I presented the progress made and questions that remain open.

References

1 Matija Pretnar. Capturing algebraic equations in an effect system. In *Dagstuhl Seminar 16112*, pages 55–57. 2016. DOI: 10.4230/DagRep.6.3.44

3.12 Quirky handlers

Matija Pretnar (University of Ljubljana, SI) and Žiga Lukšič (University of Ljubljana, SI)

Programming language terms are usually represented with an inductive type that lists all their possible constructors. It turns out that most functions on such a type are routine. For example, the set of free variables in a given arithmetic expression is almost always the union of free variables in subterms (except if the expression itself is a variable). Still, we must treat every single case in the function definition, and this quickly becomes annoying.

There are many ways in which this problem can be avoided, for example using open recursion or type classes. In the talk, we will see how to use handlers to define such functions with as little boilerplate as possible, yet ensure that the compiler forces us to revisit each part of the code when the type definition changes.

3.13 What is coalgebraic about algebraic effects and handlers?

Matija Pretnar (University of Ljubljana, SI)

In this tutorial we reviewed the work of Gordon Plotkin and John Power [1] in which they proposed comodels of algebraic theories and tensoring of comodels and models as a mathematical model for the interaction of an effectful program with its external environment. A comodel of a theory T in a category $\mathbf{C}$ is a model of T in the opposite category $\mathbf{C}^{\mathrm{op}}$, and the category of comodels in $\mathbf{C}$ is the opposite of the category of models in $\mathbf{C}^{\mathrm{op}}$. We may define the tensor $M \otimes W$ of a comodel W and a model M, which is a certain quotient of the product $M \times W$. A pair $(p, w) \in M \times W$ may be viewed as a program p running in the external environment w. The comodel W provides resources needed for execution of algebraic operations in M. In the tutorial we emphasized the fact that comodels and tensoring are a more appropriate model of top-level behavior of effectful program than various notions of “top-level” or “default” handlers. A handler has access to the continuation, but at the top level this is not the case, or else programs would be able to control the external world. It has to be the other way around.

References

1 Gordon D. Plotkin and John Power. Tensors of comodels and models for operational semantics. *Electronic Notes in Theoretical Computer Science*, 218:295–311, 2008.

3.14 Effect Handlers for WebAssembly (Show and Tell)

Andreas Rossberg (Dfinity Foundation, CH)

Integrating effects into Wasm involves a number of complications that do not exist in more high-level (and purer) languages. For example, the presence of branches interacts in interesting ways with try blocks, handlers, and continuations. It necessitates a stricter distinction between exceptions and effects. That in turn complicates the semantics and its formalisation.

We worked out a the semantics for handlers in Wasm as an extension to the existing proposal for exception handling. The next, far more challenging step will be to implement them in a production engine in order to validate their practical feasibility and evaluate their real-world performance.

3.15 Neither Web Nor Assembly

Andreas Rossberg (Dfinity Foundation, CH)

WebAssembly (or "Wasm") [1] is a portable high-performance code format that has been designed not just for the Web but for a broad range of embedding environments. It is standardised and fully formalised using well-established techniques from programming language theory. Version 1 of WebAssembly, which is currently available, was deliberately limited in scope to encompass low-level programming languages such as C++. For the next stage, more support for high-level languages will be added.

One central requirement is the ability to express the large variety of control abstractions that appear in high-level languages, such as coroutines, light-weight threads, generators, and asynchronous computations. At the lowest level, their commonality is the need to "switch stacks". However, ad-hoc mechanisms for doing so are inadequate for WebAssembly, due to its nature of a code format that must be sufficiently high-level to guarantee safety, and due to the desire to maintain a high-assurance formal specification. That asks for a primitive with strong semantic foundations.

Effect handlers would fit the bill perfectly. But it is an open question how exactly they can be designed in the context of a low-level stack machine and whether they can be implemented efficiently under the constraints imposed on existing WebAssembly implementations.

References

1 A. Haas, A. Rossberg, D. Schuff, B. Titzer, D. Gohman, L. Wagner, A. Zakai, J.F. Bastien, M. Holman. *Bringing the Web up to Speed with WebAssembly*. PLDI 2017.

3.16 Efficient Compilation of Algebraic Effects and Handlers

Tom Schrijvers (KU Leuven, BE)

Joint work of Amr Hany Saleh, Axel Faes, Georgios Karachalias, Matija Pretnar, Tom Schrijvers

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 this paper we show 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 two stages. Firstly, we combine elementary source-to-source transformations with judicious function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for pure computations.

This work comes with a practical implementation: an optimizing 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 and yields competitive performance with hand-written OCaml code.

3.17 Algebraic effects – specification and refinement

Wouter Swierstra (Utrecht University, NL)

How should we reason about programs written with algebraic effects? As the meaning of a program is determined by its handlers, we need a way to specify the intended behaviour of handlers. In this talk, I sketched an approach based on predicate transformers that enables the calculation of effectful programs from their specification.

3.18 Adding an effect system to OCaml

Leo White (Jane Street – London, GB)

Joint work of Stephen Dolan, Matija Pretnar, Sivaramakrishnan Krishnamoorthy Chandrasekaran, Daniel Hillerström

Type systems designed to track the side-effects of expressions have been around for many years but they have yet to breakthrough into more mainstream programming languages. This talk focused on on-going work to add an effect system to OCaml.

This effect system is primarily motivated by the desire to keep track of algebraic effects in the OCaml type system. Through the Multicore OCaml project, support for algebraic effects is likely to be included in OCaml in the near future. However, the effect system also allows for tracking side-effects more generally. It distinguishes impure functions, which perform side-effects, from pure functions, which do not. It also includes a tracked form of exception to support safe and efficient error handling.

This talk gave an overview of the effect system and demonstrated a prototype implementation on some practical examples.

3.19 Multi-Stage Programming with Algebraic Effects

Jeremy Yallop (University of Cambridge, GB)

I showed how algebraic effects and handlers are useful in *multi-stage* programming (a kind of programmer-directed, annotation-driven form of partial evaluation). There is a long tradition of using continuations and continuation-passing style to improve the results of partial evaluation and multi-stage programming [4, 3]. However, algebraic effects and handlers lead to a particularly elegant formulation of various transformations of the code generated by multi-stage programs.

The running example in the talk was the staging of a standard functional program – typed `printf/scanf` [1] — using BER MetaOCaml's staging facilities [3] and Multicore OCaml's implementation of effects [2]. While naively staging the program produces some performance improvements, the generated code is still sub-optimal. Several transformations, conveniently expressed using algebraic effects, significantly improve the output:

- `let` insertion untangles nested computations
- normalization of destructuring `let` bindings avoids repeated tupling and detupling
- insertion of branches into the generated code exposes information about future-stage values in each branch (such as whether a boolean variable will have the value `true` or `false` in a particular region of the program), enabling further optimizations. This last transformation makes essential use of *multi-shot continuations.*

Some of these techniques are described in more detail in recent work [5].

References

1 O. Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, Nov. 1998.

2 S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. OCaml Users and Developers Workshop 2015, September 2015.

3 O. Kiselyov. The design and implementation of BER metaocaml – system description. In *Functional and Logic Programming – 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 86–102, 2014.

4 J. L. Lawall and O. Danvy. Continuation-based partial evaluation. *SIGPLAN Lisp Pointers*, VII(3):227–238, July 1994.

5 J. Yallop. Staged generic programming. *Proc. ACM Program. Lang.*, 1(ICFP):29:1–29:29, Aug. 2017.

4 Working groups

4.1 Denotational Semantics for Dynamically Generated Effects

Robert Atkey (University of Strathclyde – Glasgow, GB)

We discussed a possible worlds / functor category semantics for a simple language with dynamically generated effect names and handlers. In this semantics, types are indexed by effect signatures, describing the set of possible effect names in scope, and computations are given a semantics in terms of a monad that supports 'free' operations from the current effect signature world, and the possible generation of new effect names in the style of Stark's monads for name generation. Some interesting program equivalences were also discussed, and it was noted that they depend on whether or not effect names can "leak" out of their scope, usually via higher-order state. The encoding of ML-style references using handlers was also discussed. This requires that the "arities" of effects can also include effect names.

4.2 Reasoning with Effects

Jeremy Gibbons (University of Oxford, GB)

We discussed a number of topics around the equations of algebraic effects and handlers:

- general concerns about the equations of an algebraic theory often being ignored
- should the equations be thought of as being attached to the operations of a theory or to the handler(s) for that theory?
- where do the equations come from? the programmer's intentions? QuickCheck exploration of the properties of a handler?

Participants

- Amal Ahmed
Northeastern University – Boston, US
- Robert Atkey
University of Strathclyde – Glasgow, GB
- Lennart Augustsson
X Inc – Mountain View, US
- Andrej Bauer
University of Ljubljana, SI
- Oliver Bracevac
TU Darmstadt, DE
- Jonathan Immanuel Brachthäuser
Universität Tübingen, DE
- Edwin Brady
University of St Andrews, GB
- Stephen Dolan
University of Cambridge, GB
- Jeremy Gibbons
University of Oxford, GB
- Daniel Hillerström
University of Edinburgh, GB
- Mauro Jaskelioff
National University of Rosario, AR
- Ohad Kammar
University of Oxford, GB
- Andrew Kennedy
Facebook – London, GB
- Oleg Kiselyov
Tohoku University – Sendai, JP
- Sivaramakrishnan Krishnamoorthy Chandrasekaran
University of Cambridge, GB
- Daan Leijen
Microsoft Research – Redmond, US
- Sam Lindley
University of Edinburgh, GB
- Andres Löh
Well-Typed LLP, DE
- Žiga Lukšič
University of Ljubljana, SI
- Anil Madhavapeddy
Docker Inc. – Cambridge, GB
- Conor McBride
University of Strathclyde – Glasgow, GB
- Adriaan Moors
Lightbend Inc. – Lausanne, CH
- Matija Pretnar
University of Ljubljana, SI
- Andreas Rossberg
Dfinity Foundation, CH
- Tom Schrijvers
KU Leuven, BE
- Perdita Stevens
University of Edinburgh, GB
- Wouter Swierstra
Utrecht University, NL
- Leo White
Jane Street – London, GB
- Nicolas Wu
University of Bristol, GB
- Jeremy Yallop
University of Cambridge, GB