Paella: algebraic effects with parameters and their handlers

Jesse Sigal, joint work with Ohad Kammar, Cristina Matache, and Conor McBride

ABSTRACT

We present and report about early work-in-progress developing a library for algebraic effects and handlers with resumptions structured after Kripke possible-world semantics. The resulting abstraction can express dynamic allocation effects such as dynamically allocated full ground reference cells and handlers that manipulate these references as non-dangling references on a heap. We will demonstrate our implementation in Brady's dependently-typed Idris 2, and describe its future directions and prospects to advanced mutable-to-immutable data-structure transformation, dynamically allocated thread schedulers, functional-logic programming, and constraint solving.

1 INTRODUCTION

Context. The last 40 years have seen the rise in programming with *user-defined* computational effects. Originating in early uses of expressive control-effects to define advanced control-flow abstractions [Friedman et al. 1984; Felleisen et al. 1986], this technique has gained popularity through the use of monads [Wadler 1990; Spivey 1990], and most recently algebraic effects and handlers [Cartwright and Felleisen 1994; Hancock and Setzer 2000; Plotkin and Pretnar 2009; Bauer and Pretnar 2015]. The latter two variants, monads and effect handlers, demonstrate mathematically structured programming in practice: computational counterparts to established semantic theories for effects, Moggi's use of monads [1989] and Plotkin and Power's use of universal algebra [2001; 2002].

A simple computational model for algebraic effects and handlers lends itself to a straightforward implementation. Programs define *computation trees* whose nodes express invocation of user-defined effects and whose branching represents the resumption following the invocation of effects. User-defined handlers traverse these trees, often using a fold recursion scheme. Each clause in an effect-handler explains how to visit each type of node, potentially calling the handled resumption. The connection to universal algebra manifests in the signatures identifying the resumption type of each node with an *arity* of an algebraic operation, e.g., binary backtracking choice or dereferencing of a memory bit, nullary exception raising, and unary resumptions.

The general theories of monads and algebraic effects can be formulated in general semantic settings, accounting for sophisticated computational effects. Staton [2013] extends them to account for sophisticated effects involving the dynamic allocation of names and references, including: names as used in process calculi; dynamically allocated mutable state; and logic variables. The core property setting these types of effects apart from more traditional ones is their arities, which represent concrete values that can change dynamically. In our three examples: a concrete name is chosen at runtime to ensure it is fresh compared to all other names in play; a dynamically allocated reference is chosen at runtime by the memory allocator; and two logic variables may become identical through unification which, moreover, may allocate further fresh logic variables for components of the shared value. In terms of computation trees, a resumption may be constructed in one world and invoked in a future world. Staton's theory follows a well-established line of work on *functor-category semantics* [Reynolds 1981; Oles 1986]. This tradition models varying-values by interpreting types as families of sets indexed by a set of sorted *parameters*—the *world*—representing the set of dynamically allocated values. Each family comes equipped with an operation that promotes values in one world to values at a later world, future-proofing values to further dynamic allocations.

Contribution. We report on work-in-progress developing algebraic effects and handlers for effects with dynamic allocation. We present *Paella*, a **p**arameterised **a**lgebraic **e**ffects **l**ibrary/**la**nguage implemented in Idris 2 [Brady 2021]. The main ingredient is a data-structure capable of representing resumptions/effects with arities that may vary dynamically. Resumptions thus have a Kripke possible-world structure. They take an additional argument representing the evolution of the available parameters—the world—from their lexical values to their dynamic values. We want to make this additional argument implicit, and have successfully prototyped this design in Haskell using type-classes.

We employ a mathematically structured programming approach. Traditionally, the free monads that represent computation trees are structured as set-level functions [Hancock and Setzer 2000; Swierstra 2008; Kammar et al. 2013]. We structure them using types indexed by representations of finite parameter-sets and equipped with a promotion action. This development relies on a recent decomposition of functor categories into coalgebras over indexed families [Allais et al. 2018; Fiore and Szamozvancev 2022]. Our initial results include:

- A mathematically-structured data-structure for Kripke computation trees, implemented in Idris 2.
- A suitable handler for dynamically allocated state involving pointers-to-pointers (full-ground references).

Prospects. We want to write more sophisticated handlers. First, we want to add garbage-collection steps to this heap handler. Next, we want to explore the *Tarjan-Sleator transform* [Driscoll et al. 1989]. Here the programmer declares mutable pointer data structures and a suitable handler transforms them into similar *versioned* immutable data structures with close asymptotic performance.

We also want to explore other effects: threads and logic variables. Logic variables are particularly exciting because they form the basis for unification-based type-checking. There, the notion of progress between worlds represents solutions of unification constraints. We are writing from our experience developing Epigram 1 [McBride and McKinna 2004]. There, the elaborator did not have a built-in ability to transport resumptions. That is, when the elaborator tried to resume a suspended problem, the values stored in the resumption's closure were out-of-date with respect to the progress made in the meantime, leading to subtle bugs[1]. As a consequence, solvable

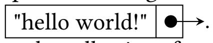
HOPE'24, Sept 2024, Milan, Italy
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

elaboration problems did not progress sufficiently, leading to false-positive type-checking errors. The Epigram 1 elaborator solved this issue only by the heavy-handed reification of resumptions as syntax open to substitution. Paella offers a more elegant approach in which we express elaboration problems as computation trees, and the strategy for solving them as appropriate handlers.

2 EFFECTS WITH DYNAMIC ALLOCATION

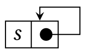
Algebraic effects centre around an algebraic signature, which traditionally associates to each *operator* an *argument*[2] and *arity* types. For example, we represent a store of globally-known locations $\ell \in \mathbf{Loc}$ storing bits $b \in \mathbf{Bit} := \{0, 1\}$, using two operators:
•read : **Loc⇝Bit:** representing the binary operation that dereferences a given argument location and returns the bit it stores. It is a binary operator as its resumption expects one of two possible results for the bit stored in this location.
•write : **Loc × Bit⇝Unit:** representing the unary operation that mutates the argument location to store the argument value. It is a unary operator as the resumption expects no information in return.
We include a short review of algebraic effects in Appendix A and a simple computation-tree-based implementation in Idris 2, with a state-passing handler for global state with $\mathbf{Loc} = \{0, 1, \dots, 9\}$.

In this example, the collection of memory addresses is fixed globally. In particular, it does not enjoy the dynamic allocation and abstraction properties we expect from dynamically allocated references in a high-level programming language such as OCaml that supports such reference cells. To support such dynamic allocation, we need to move outside the realm of plain collection.

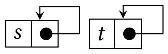
Algebraically, our operators can involve sorted parameters $p : c$. As a running example, we will use the parameter ptr that represent a reference to a *string cons-cell*, a pair of a string and a ptr, which we can represent graphically as: 

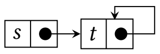
. We need to move to indexed collections because the collection of possible values changes dynamically depending on the allocated references. We demonstrate this fact with three types of values: possible references of type ptr; possible well-formed cons-cells; and heaps:

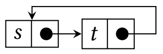
- If none have been allocated, then the collection of references is empty; the collection of cons-cell is empty since each cell requires a reference; and the only possible heap is empty.
- With one ptr-reference, the collection of ptr-references has exactly one value—the unique reference; we have countably many possible cons-cells, each pointing at the unique reference; and only heaps with one self-pointing cell:



- With two references, we have three kinds of heaps:



; 

; and 

So we need to use sets/types that vary depending on the allocated references. More generally, they will vary depending on the allocated parameters.

[1]Conor dubs this The Goodbye-Lenin Problem. In the movie of that name, the protagonist's mother wakes up from a coma and she does not know that East Germany is no longer as it was.
[2]Plotkin and Power [2001] call the argument type the *parameter* type. As this notion of parameter is different to Staton's, we use a different term.

3 KRIPKE COMPUTATION TREES

Let A be a set of *sorts*, such as ptr. A *world* w consists of a finite set var w, whose element are called *parameters* together with an assignment of sorts $s : \mathrm{var}\, w \longrightarrow A$. The collections of references, storable values in references, and heaps are all example of world-indexed families. Putting these into Idris, and reusing the standard library wherever possible, we define worlds as snoc-lists of sorts:

```
World : Type             | data Var : A -> Family where
World = SnocList A       |  Here  : Var a (w :< a)
Family : Type            |  There : Var a w ->
Family = World -> Type   |          Var a (w :< b)
```

Families are closed under many operations. For example, the product of a collection of families is a family: $(\prod_i F_i)w := \prod_i (F_i w)$, which in Idris 2 we implement with the following product type:

```
data ForAll : SnocList a -> (a -> Type) -> Type where
 Lin  : ForAll xs p
 (:<) : ForAll xs p -> p x -> ForAll (xs :< x) p
FamProd : SnocList Family -> Family
FamProd fs w = ForAll fs (\f => f w)
```

For example, a cons-cell is the product family of the constantly string family and the ptr-parameter family:

```
ConsCell : Family
ConsCell = FamProd [< const String, Var Ptr]
```

A *renaming* $\rho : w \rightsquigarrow v$ is a sort-preserving function between the underlying sets of parameters $\rho : \mathrm{var}\, w \longrightarrow \mathrm{var}\, v$. The renamings out of a given world are a family:

```
(~>) : World -> World -> Type  | Env : World ->
w ~> u = (a : A) ->            |   Family
  Var a w -> Var a u           | Env w = (w ~>)
```

These renamings act on the families `Var A` and `ConsCell` making them into functors. In our setting, we do not enforce any functoriality laws yet, and so overapproximate the collection of natural transformations between two functors by the collection of transformations between them $(F \Longrightarrow G) := \prod_w (Fw \longrightarrow Gw)$. So we define the Kripke exponential as $(F \twoheadrightarrow G)w := ((\mathbf{Env}\, w \times F) \Longrightarrow G)$:

```
(-%) : Family -> Family -> Family
(f -% g)w = FamProd [< Env w, f] -|> g
```

Putting these together, we implement signatures and computation trees almost verbatim using the Kripke concepts:

```
 record OpSig where       | Signature : Type
  constructor (~|>)       | Signature =
  Args,Arity : Family     |    OpSig -> Type
partial
data (.Free) : Signature -> Family -> Family where
 Return : f -|> sig.Free f
 Op : {opSig : OpSig} -> {f : Family} ->
  (op : sig opSig ) ->
  FamProd [< opSig.Args , opSig.Arity -% sig.Free f]
   -|> sig.Free f
```

As this implementation is not strictly positive, we need to declare it as **`partial`**. We similarly define algebras and handlers, and implement a dynamically allocated heap.

REFERENCES

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proc. ACM Program. Lang.* 2, ICFP (2018), 90:1–90:30. https://doi.org/10.1145/3236785

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/J.JLAMP.2014.02.001

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPICS.ECOOP.2021.9

Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 789)*, Masami Hagiya and John C. Mitchell (Eds.). Springer, 244–272. https://doi.org/10.1007/3-540-57887-0_99

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making data structures persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124. https://doi.org/10.1016/0022-0000(89)90034-2

Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1986. Reasoning with Continuations. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986.* IEEE Computer Society, 131–141.

Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498715

Daniel P. Friedman, Christopher T. Haynes, and Eugene Kohlbecker. 1984. Programming with Continuations. In *Program Transformation and Programming Environments*, Peter Pepper (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–274.

Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic*, Peter G. Clote and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 317–331.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *J. Funct. Program.* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989.* IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

F J Oles. 1986. *Type algebras, functor categories and block structure.* Cambridge University Press, USA, 543–573.

Gordon Plotkin and John Power. 2001. Semantics for Algebraic Operations. *Electronic Notes in Theoretical Computer Science* 45 (2001), 332–345. https://doi.org/10.1016/S1571-0661(04)80970-8 MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics.

Gordon D. Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2303)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

John C. Reynolds. 1981. The Essence of Algol. In *Algorithmic Languages; Proceedings of the International Symposium on Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet (Eds.). Elsevier Science Inc., USA.

Mike Spivey. 1990. A functional theory of exceptions. *Science of Computer Programming* 14, 1 (1990), 25–42. https://doi.org/10.1016/0167-6423(90)90056-J

Sam Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science.* 519–519. https://doi.org/10.1109/LICS.2013.58

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 61–78. https://doi.org/10.1145/91556.91592

A ALGEBRAIC EFFECTS IN IDRIS 2

In dependently-typed languages like Idris 2, we represent signatures as families of operations (below, right) indexed over the argument and arity types, which we construct using the infix data constructor `(~|>)` (below, left):

```
record AlgOpSig where       | AlgSignature : Type
 constructor (~|>)          | AlgSignature =
 Args, Arity : Type         |       AlgOpSig -> Type
```

For example, take the signature for global state below (right), using a fixed type of globally known addresses storing a bit (below, left).

```
Loc : Type            | data OpGS : AlgSignature where
Loc = Fin 10          |  Read  : OpGS (Loc ~|> Bit)
data Bit = O | I      |  Write : OpGS ((Loc, Bit) ~|> Unit)
```

The X-*valued* E-*computation trees* for such a signature E are then those trees whose leaves are in X and whose R-branching nodes are labelled by an operator $(f : A \rightsquigarrow R) \in E$ and an argument $v \in A$. We use a free monad [Swierstra 2008] in postfix notation for them:

```
data (.Free) : AlgSignature -> Type -> Type where
 Return : x -> sig.Free x
 Op : sig opSig ->
 (opSig.Args,opSig.Arity -> sig.Free x)-> sig.Free x
```

As we will shortly see, the free monad has a monad structure, and we can use monadic **do**-notation to define computation trees. Idris 2 also include that bang (`!`) notation which prepends its argument before the current monadic context.

For example, the following computation tree swaps the contents of the two at locations 4 and 5 and returns them:

```
Ex1 : (OpGS).Free (Bit, Bit)
Ex1 = do
  tmp <- read 4
  write (4, !(read 5))
  write (5, tmp)
  Return (!(read 4), !(read 5))
```

To define a handler for an X-valued E-computation tree into some B, we require two structures. First, an E-*algebra* structure over B, interpreting every oprator $(f : A \rightsquigarrow R) \in E$ as an R-ary A-parameterised operation $A \times B^R \longrightarrow B$, curried:

```
0 (.AlgebraOver) : AlgSignature -> Type -> Type
sig.AlgebraOver b = {0 opSig : AlgOpSig} ->
  (op : sig opSig) ->
  (opSig.Arity -> b) -> (opSig.Args -> b)
```

The keyword `0` is a *quantity* annotation, indicating that (a) the function `(.AlgebraOver)` and and (b) the argument `opSig` must be erased at runtime.

Every signature supports the free algebra structure:

```
(.FreeAlgOver) : (0 sig : AlgSignature) ->
                 (0 x : Type) ->
               sig.AlgebraOver (sig.Free x)
sig.FreeAlgOver x op k args = Op op (args, k)
```

The second structure we need to define for handlers is a *valuation* $v : X \longrightarrow B$. The free algebra over X also supports the valuation `Return : x -> sig.Free x`.

Given an E-algebra structure and a valuation, we can fold over computation trees inductively:

```
(.fold) : sig.AlgebraOver b -> (x -> b) ->
          sig.Free x -> b
a.fold val (Return y) = val y
a.fold val (Op op (args, k)) =
   a op (a.fold val . k) args
```

In particular, folding using the free algebra structure provides the monad structure for the free monad:

```
(>>=) : sig.Free x -> (x -> sig.Free y) -> sig.Free y
t >>= k = (sig.FreeAlgOver y).fold k t
```

The free monad also supports *generic effects* [Plotkin and Power 2001] for each operator, which creates a node with the corresponding effects:

```
0 genericOpType :
  (sig : AlgSignature) -> (opSig : AlgOpSig) -> Type
genericOpType sig opSig =
  opSig.Args -> sig.Free opSig.Arity

genericOp : (op : sig opSig) -> genericOpType sig opSig
genericOp op arg = Op op (arg, \x => Return x)
```

For example, the two generic effects for global state are reading and writing to it:

```
read : genericOpType OpGS (Loc ~|> Bit)
read = genericOp BG.Read
```

```
write : genericOpType OpGS ((Loc, Bit) ~|> Unit)
write = genericOp BG.Write
```

As an end-to-end example, we recall the dynamically-scoped global-state handler [Kammar and Pretnar 2017] that has the following algebra structure. In this setting, a *store* is a mapping from locations to bits **Store** := **Loc** $\longrightarrow$ **Bit**, which we represent as a 10-element vector (below, left):

```
Store : Type          | InitStore : Store
Store = Vect 10 Bit   | InitStore = replicate 5 O
                      |          ++ replicate 5 I
```

We will use the example initial store (above, right) whose first half is unset/zeroed and whose second half is set.

The global state handler returning results in B is then the following algebra structure over **Store** $\longrightarrow B$:

```
DynScopeGS : (OpGS).AlgebraOver (Store -> b)
DynScopeGS {opSig = .(Loc        ~|> Bit )}
 Read  k loc s = k (index loc s) s
DynScopeGS {opSig = .((Loc, Bit) ~|> Unit)}
 Write k (loc, v) s = k () (replaceAt s loc v)
 where
 replaceAt : Vect n a -> Fin n -> a -> Vect n a
 replaceAt (x :: xs) FZ y = y :: xs
 replaceAt (z :: xs) (FS pos) y =
   z :: replaceAt xs pos y
```

E.g., `(DynScopeGS).fold (\x,s => x) Ex1 InitStore` evaluates to `(I, O)`.