Paella: algebraic effects with parameters and their handlers

Jesse Sigal (University of Edinburgh)
joint work with
Ohad Kammar (University of Edinburgh)
Cristina Matache (University of Edinburgh)
Conor McBride (University of Strathclyde)

HOPE 12, September 2, 2024

Overview

Overview

Algebraic effects and handlers

- Why?
  - **User-defined** computational effects
  - Mathematically structured
- Examples
  - Backtracking choice
  - Global state
  - Exceptions
  - Yielding
- Implementation
  - Programs represent **computation trees**
  - Handlers **fold** over these trees

State effects

- Effects for static state:

```
read : Loc -> Bit
write : (Loc, Bit) -> ()
```

- Effects for dynamically-allocated state:

```
new : Bit -> Loc
gc : Policy -> ()
```

Problems for state effects

- To support `new` and `gc`, `Loc` needs to be "abstract" and/or "dynamic"
  - Avoid counterfeit locations
  - Change when memory cell moves
- E.g. capturing a reference in a closure

```
ExDangling = do
  loc <- new I
  let kont = \_ => write (loc, 0) -- Capture `loc` in closure
  _ <- gc Compact
  kont () -- Writing to possibly dangling pointer!
```

Overview

Algebraic effects and handlers

Ordinary computation trees

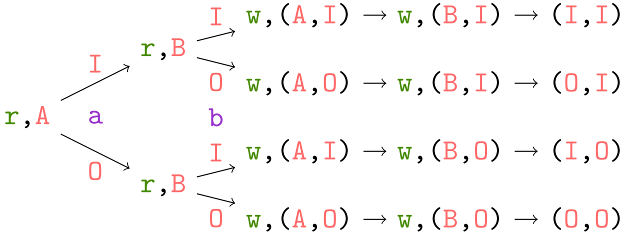
An example program

Kripke computation trees

Conclusion

Algebraic effects and handlers: computation trees

```
ExGS : (OpGS).Free (Bit, Bit)
ExGS = do -- Swaps values
  a <- read A
  b <- read B
  () <- write (A, b)
  () <- write (B, a)
  pure (b, a) -- (A,B)
```




```
data Loc = A | B
data Bit = O | I
```

Algebraic effects and handlers: computation trees

```
ExGS : (OpGS).Free (Bit, Bit)
ExGS = do -- Swaps values
  a <- read A
  b <- read B
  () <- write (A
  () <- write (B
  pure (b, a) -- (A,B)
```

```
I → r,B → I w,(A,I) → w,(B,I) → (I,I)
→ w,(B,I) → (O,I)
→ w,(B,O) → (I,O)
O w,(A,O) → w,(B,O) → (O,O)
```

```
data OpGS : AlgSignature where
  Read  : OpGS (Loc ~|> Bit)
  Write : OpGS ((Loc, Bit) ~|> ())
```

```
data Loc = A | B
data Bit = O | I
```

Algebraic effects and handlers: core types

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

AlgSignature : Type
AlgSignature = AlgOpSig -> Type

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

Algebraic effects and handlers: core types

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

AlgSignature : Type
AlgSignature = AlgOpSig -> Type

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

Algebraic effects and handlers: core types with parameters

```
record AlgOpSig where
  constructor (~|>)
  Args, Arity : Family

AlgSignature : Type
AlgSignature = AlgOpSig -> Type

data (.Free) : AlgSignature -> Family -> Family where
  Return : x -|> sig.Free x
  Op : sig opSig ->
    FamProd [< opSig.Args, opSig.Arity -% sig.Free x] -|> sig.Free x
```

Overview

Ticking: operations

Two kinds of dynamic values (Var): ticking Clocks and cooperative Tasks (cf. [Hillerström and KC(2017), Matache(2024)]).

```
data TickySig : Cell .signature where
  /// Allocate a fresh clock initialised by argument
  New  : TickySig (const Nat ~|> Var Clock)
  /// Synchronise two clocks,adding their ticks
  Sync : TickySig (FamProd [< Var Clock, Var Clock] ~|> const ())
  /// Tick a clock
  Work : TickySig (FamProd [< Var Clock, const Nat] ~|> const ())
  /// Wait until clock ticks past argument
  WaitUntil : TickySig (FamProd [< Var Clock, const Nat] ~|> const ())
  /// Threading a-la Unix interface [cf. Matache'24]
  Stop : TickySig (const () ~|> const Void)
  Fork : TickySig (const () ~|> FamSum [< Var Task, const ()])
  Wait : TickySig (Var Task ~|> const ())
```

Ticking: example program

- Consider the example program:

```
ExTicks =                    (\w, [<                ] =>
  new  _ 0            ) >>== (\w, [< c1             ] =>
  new  _ 1            ) >>== (\w, [< c1, c2         ] =>
  new  _ 3            ) >>== (\w, [< c1, c2, c3     ] =>
  work _ [< c1, 50])  >>>>   (\w, [< c1, c2, c3     ] =>
  work _ [< c2, 60])  >>>>   (
  forkOff             $      (\w, [< c1, c2, c3     ] =>
    waitUntil
       _ [< c3, 40])  >>>>   (\w, [< c1, c2, c3     ] =>
    work
       _ [< c1, 10])
  )                   >>==   (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c3])  >>>>   (\w, [< c1, c2, c3, tid] =>
  wait _ tid          ) >>>> (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c2])
```

Ticking: example program

- Consider the example program:

```
ExTicks =                      (\w, [<                    ] =>
  new  _ 0           ) >>== (\w, [< c1                ] =>
  new  _ 1           ) >>== (\w, [< c1, c2            ] =>
  new  _ 3           ) >>== (\w, [< c1, c2, c3        ] =>
  work _ [< c1, 50]) >>>> (\w, [< c1, c2,
  work _ [< c2, 60]) >>>> (
  forkOff            $      (\w, [< c1, c2,
    waitUntil
      _ [< c3, 40]) >>>> (\w, [< c1, c2,
    work
      _ [< c1, 10])
  )                    >>== (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c3]) >>>> (\w, [< c1, c2, c3, tid] =>
  wait _ tid         ) >>>> (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c2])
```

$$\frac{\Gamma \vdash t : MA \qquad \Gamma, x : A \vdash u : MB}{\Gamma \vdash \text{let } x = t \text{ in } u : MB}$$

Ticking: example program

- Consider the example program:

```
ExTicks =                    (\w, [<                ] =>
  new  _ 0         ) >>== (\w, [< c1             ] =>
  new  _ 1         ) >>== (\w, [< c1, c2         ] =>
  new  _ 3         ) >>== (\w, [< c1, c2, c3     ] =>
  work _ [< c1, 50]) >>>> (\w, [< c1, c2, c3     ] =>
  work _ [< c2, 60]) >>>> (
  forkOff          $      (\w, [< c1, c2, c3     ] =>
    waitUntil
       _ [< c3, 40]) >>>> (\w, [< c1, c2, c3     ] =>
    work
       _ [< c1, 10])
  )                  >>== (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c3]) >>>> (\w, [< c1, c2, c3, tid] =>
  wait _ tid       ) >>>> (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c2])
```

Ticking: example program

- Consider the example program:

```
ExTicks =                      (\w, [<
  new  _ 0          ) >>== (\w, [< c1
  new  _ 1          ) >>== (\w, [< c1
  new  _ 3          ) >>== (\w, [< c1
  work _ [< c1, 50]) >>>> (\w, [< c1
  work _ [< c2, 60]) >>>> (
  forkOff            $      (\w, [< c1, c2, c3     ] =>
    waitUntil
      _ [< c3, 40]) >>>> (\w, [< c1, c2, c3     ] =>
    work
      _ [< c1, 10])
  )                    >>== (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c3]) >>>> (\w, [< c1, c2, c3, tid] =>
  wait _ tid          ) >>>> (\w, [< c1, c2, c3, tid] =>
  sync _ [< c1, c2])
```

Time aware continuations; when work ticks and sync merges clocks, the rest of the program is updated

Ticking: takeaways

- `Var Clock` has dynamic values which change based on the **world**, in this which clocks and tasks are at large
- Each `new` and `fork` change the world
- Can't be solved by naive IO references due to clock synchronisation (requires another level of indirection)
- `Clock` and `Task` are example **parameters** [Staton(2013)]
- Parameter for local state: the shape of the heap
- Enter **Paella**, a **pa**rameterised **al**gebraic **e**ffects **l**ibrary/**la**nguage

Idea: do the same, but with world-aware types, i.e. Kripke semantics

Overview

Algebraic effects and handlers

Ordinary computation trees

An example program

Kripke computation trees

Conclusion

Kripke semantics: worlds

```
World : Type

infixr 1 ~>
(~>) : (src, tgt : World) -> Type

id : t ~> t

infixr 9 .
(.) : (t2 ~> t3) -> (t1 ~> t2) -> (t1 ~> t3)
```

Kripke semantics: families

```
Family : Type
Family = World -> Type

infixr 1 -|>
(-|>) : (f, g : Family) -> Type
f -|> g = (w : World) -> f w -> g w

id : {f : Family} -> f -|> f
id w x = x

infixr 9 .
(.) : {f, g, h : Family} -> (g -|> h) -> (f -|> g) -> (f -|> h)
(beta . alpha) w = beta w . alpha w
```

Kripke semantics: signatures and computation trees

```
record OpSig where
  constructor (~|>)
  Args   : Family
  Arity : Family

Signature : Type
Signature = OpSig -> Type

data (.Free) : Signature -> Family -> Family where
  Return : f -|> sig.Free f -- (w : World) -> f w -> sig.Free f w
  Op : {opSig : OpSig} -> {f : Family} -> (op : sig opSig) ->
    FamProd [< opSig.Args, opSig.Arity -% sig.Free f] -|> sig.Free f
```

Kripke semantics: families with actions (presheaves)

```
ActionOver : Family -> Type
ActionOver f = {w, w' : World} -> (rho : w ~> w') -> (f w -> f w')

Box : Family -> Family
Box f w = (w' : World) -> (w ~> w') -> f w'

record BoxCoalg (f : Family) where
  constructor MkBoxCoalg
  next : f -|> Box f
  -- (w : World) -> f w -> (w' : World) -> (w ~> w') -> f w'

(.map) : {f : Family} -> BoxCoalg f -> ActionOver f
coalg.map {w,w'} rho x = coalg.next w x w' rho
```

See [Allais et al.(2018), Fiore and Szamozvancev(2022)] for this approach

Kripke semantics: basic families with actions

```
BoxCoalgConst : {t : Type} -> BoxCoalg (const t)
BoxCoalgConst = MkBoxCoalg $ \_, x, _, _ => x

Env : World -> Family
Env w = (w ~>)

BoxCoalgEnv : {w0 : World} -> BoxCoalg (Env w0)
BoxCoalgEnv = MkBoxCoalg $ \w, rho, w', rho' => rho' . rho
-- rho  : w0 ~> w
-- rho' : w  ~> w'
```

Kripke semantics: product of families with actions

```
data ForAll : SnocList a -> (a -> Type) -> Type where
  Lin  : ForAll sx p
  (:<) : ForAll sx p -> p x -> ForAll (sx :< x) p

FamProd : SnocList Family -> Family
FamProd sf w = ForAll sf (\f => f w)

BoxCoalgProd : {sf : SnocList Family} ->
  ForAll sf BoxCoalg -> BoxCoalg $ FamProd sf
```

Kripke semantics: exponential of families with actions

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

eval : FamProd [< f -% g, f] -|> g
eval w [< alpha, x] = alpha w [< id, x]

(.curry) : {h : Family} -> (coalg : BoxCoalg h) ->
  (FamProd [< h, f] -|> g) -> (h -|> (f -% g))

BoxCoalgExp : BoxCoalg (f -% g)
BoxCoalgExp = MkBoxCoalg $ \w, alpha, w', rho =>
  \w'', [< rho', x] => alpha w'' [< rho' . rho, x]
-- rho  : w  ~> w'
-- rho' : w' ~> w''
```

Kripke semantics: signatures and computation trees again

```
record OpSig where
  constructor (~|>)
  Args, Arity  : Family

Signature : Type
Signature = OpSig -> Type

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
```

Kripke semantics: signatures and computation trees again

```
recor
   con
   Arg

Signa
Signa
```

Inlining the abstractions:

```
(w : World) ->
  ( opSig.Args w,
    (w' : World) -> (w ~> w') -> opSig.Arity w' -> sig.Free f w'
  )
-> sig.Free f w
```

```
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
```

Overview

Recap of Kripke semantics

- We defined a type of worlds and families over such worlds
- We defined families with actions (presheaves), as well as their products and exponentials
- We defined new computation trees with branching that supports any future world
- These trees have an action and a folding operation
- They also form a monad, and so we can create computations which are updatable!

Prospects

- Applications
  - Full ground local state (i.e. ground values and pointers) and the Tarjan-Sleator transform (WIP in Idris 2)
  - Elaboration and constraint solving with meta-variables (already in Haskell)
  - Threads (WIP in Idris 2)
- Improved ergonomics
  - Semantic reflection for Idris 2
  - Type classes and local instances (already in Haskell)

Repository: `https://github.com/ohad/paella`

Bibliography I

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

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 Hillerström and KC. 2017.
Concurrent Programming with Effect Handlers.

Bibliography II

Cristina Matache. 2024.
An algebraic theory of named threads.

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