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)

S-REPLS 15, July 18, 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: signatures

- An operation $f : A \rightsquigarrow R$ is specified by an argument type A and arity/return type R, constructed using the infix `(~|>)`:

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

- A signature E is a family indexed by operation signatures:

```
AlgSignature : Type
AlgSignature = AlgOpSig -> Type
```

- For example, the global state signature:

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

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

Algebraic effects and handlers: computation trees

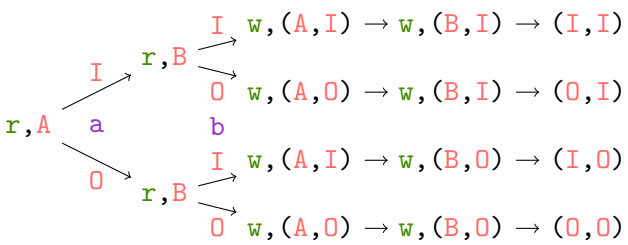
- X-**valued** E-**computation trees**
  - leaves in X
  - nodes labelled by an operator $(f : A \rightsquigarrow R) \in E$ and a $v \in A$, and are R-branching

```
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: computation trees

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

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



Algebraic effects and handlers: handlers for trees

- Handler for an X-valued E-computation tree into some B
  - E-**algebra** structure over B: for each $(f : A \rightsquigarrow R) \in E$, an operation $c_f : A \times B^R \to B$ (equivalently $c_f : B^R \to B^A$)

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

  - **valuation** $v : X \to B$
- Given an E-algebra structure and a valuation, we can fold over computation trees:

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

Algebraic effects and handlers: monad structure via free algebra

- Every signature E has an E-algebra and valuation over trees

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

valuation : {sig : AlgSignature} -> {x : Type} ->
  x -> sig.Free x
valuation x = Return x
```

- Folding with the free algebra structure gives the monad structure for `sig.Free`:

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

Overview

Ticking counters

- Suppose we have a dynamic value which can tick up:

```
Ticky : Type
Ticky = IORef Int
```

- We have three operations:
  - **grab** a new value to start ticking
  - **emit** the current value
  - **wait** a number of ticks

```
data OpTime : AlgSignature where
  Grab : OpTime (() ~|> Ticky)
  Emit : OpTime (Ticky ~|> ())
  Wait : OpTime (() ~|> ())
```

Ticking: example program

- An example program:

```
ExTime : (OpTime).Free ()
ExTime = do
  x <- grab ()  -- x = 0
  wait ()       -- wait n ticks; x = n
  y <- grab ()  -- x = n; y = 0
  emit x        -- emits n
  emit y        -- emits 0
  wait ()       -- wait m ticks; x = n + m; y = m
  emit x        -- emits n + m
  emit y        -- emits m
```

Ticking: stateful implementation for waiting

- The type to handle into, pass a list of `Ticky`s to update:

```
Clocked : Type
Clocked = List Ticky -> IO ()
```

- When we `wait`, increment all the `Ticky`s:

```
waiting : Clocked -> Clocked
waiting c = \ts => do
  putStrLn "waiting"
  getLine >>= \case
    "" => do
      _ <- for ts (\ticky => modifyIORef ticky (+ 1))
      waiting c ts
    _  => c ts
```

Ticking: handler

- We then define the algebra:

```
ClockedTime : (OpTime).AlgebraOver Clocked
ClockedTime {opSig = .(() ~|> Ticky)} Grab k x = \ts => do
  ticky <- newIORef 0
  k ticky (ticky :: ts)
ClockedTime {opSig = .(Ticky ~|> ())} Emit k x = \ts => do
  cur <- readIORef x
  printLn cur
  k () ts
ClockedTime {opSig = .(() ~|> ())}    Wait k x = \ts => do
  waiting (k ()) ts
```

- And now we can run the example:

```
RunExTime : IO ()
RunExTime = (ClockedTime).fold (\_, _ => putStrLn "done") ExTime []
```

Ticking: summary

- Ticking illustrates `Ticky` as dynamic collection of values which change based on the **world**, in this case time
- Each `wait` changes the world
- Solved with IO references and keeping them all as state
- Alternative: time as an explicit **parameter** [cf. Staton'13]
- Parameter for local state: the shape of the heap
- Enter **Paella**, a **p**arameterised **a**lgebraic **e**ffects **l**ibrary/**la**nguage

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

Overview

Kripke semantics: worlds

```
data Time = Z | S Time

World : Type
World = Time

data Le : Time -> Time -> Type
 where
  Now : Le t t
  Later : Le s t -> Le s (S t)
```

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

id : t ~> t
id = Now

infixr 9 .
(.) : t2 ~> t3 -> t1 ~> t2 -> t1 ~> t3
(.) Now le = le
(.) (Later le1) le2 = Later (le1 . le2)
```

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: 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]

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

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

Kripke semantics: 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
  Op : {opSig : OpSig} -> {f : Family} -> (op : sig opSig) ->
    FamProd [< opSig.Args, opSig.Arity -% sig.Free f] -|> sig.Free f
```

Kripke semantics: computation trees

```
recor
  con
  Arg
  Ari

Signa
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
```

Inlining the abstractions:

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

Kripke semantics: algebras for trees

```
(.AlgebraOver) : Signature -> Family -> Type
sig.AlgebraOver g = {opSig : OpSig} -> (op : sig opSig) ->
  (opSig.Arity -% g) -|> (opSig.Args -% g)

TermAlgebra : {sig : Signature} ->
  (f : Family) -> BoxCoalg f -> sig.AlgebraOver (sig.Free f)

pure : {sig : Signature} -> {f : Family} -> f -|> sig.Free f
pure = Return
```

Kripke semantics: folds for trees, monad structure

```
(.fold) : {sig : Signature} -> {f, g : Family} ->
  sig.AlgebraOver g -> (f -|> g) -> (sig.Free f -|> g)

(.extend) :  {sig : Signature} -> {f,g : Family} ->
  BoxCoalg g -> (f -|> sig.Free g) -> (sig.Free f -|> sig.Free g)
coalg.extend alpha = (TermAlgebra g coalg).fold alpha
```

Overview

Ticking with Kripke trees: a new ticky

- We can now define a time aware `Ticky`:

```
data Ticky : World -> Type where
  Clock : Int -> Ticky t

BoxCoalgTicky : BoxCoalg Ticky
BoxCoalgTicky = MkBoxCoalg $ \t1, x, t2, ticks =>
  case ticks of
    Now => x
    Later ticks' =>
      case (BoxCoalgTicky .map ticks' x) of
        Clock i => Clock (i + 1)
```

- Note how we no longer need IO references, state is now pure!
- When brought to a future world, the clock will be updated

Ticking with Kripke trees: operations

- Our operations are analogous:

```
data OpTime : Signature where
  Grab : OpTime (const () ~|> Ticky)
  Emit : OpTime (Ticky ~|> const ())
  Wait : OpTime (const () ~|> const ())
```

Ticking with Kripke trees: example program again

- Our program is analogous:

```
ExTime : FamProd [<] -|>  (OpTime).Free (const ())
ExTime =              (\t, [<]  =>
  grab _ () ) >>== (\t, [< x] =>
  wait _ () ) >>>> (\t, [< x] =>
  grab _ () ) >>== (\t, [< x, y] =>
  emit _ x  ) >>>> (\t, [< x, y] =>
  emit _ y  ) >>>> (\t, [< x, y] =>
  wait _ () ) >>>> (\t, [< x, y] =>
  emit _ x  ) >>>> (\t, [< x, y] =>
  emit _ y  )
```

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

- Time aware continuations; when `wait` advances time the rest of the program is updated

Ticking with Kripke trees: IO only for interaction

- Our algebra type now ignores the time:

```
Clocked : Time -> Type
Clocked t = IO ()

BoxCoalgClocked : BoxCoalg Clocked
BoxCoalgClocked = MkBoxCoalg $ \t1, m, t2, ticks => m
```

- We only use IO for input and output in order to choose how long to wait
- Importantly, we will not use it for state

Ticking with Kripke trees: waiting

- Waiting now advances the world:

```
waiting : (const () -% Clocked) -|> (const () -% Clocked)
waiting t cont = \t', [< ticks, ()] => do
  let cont' = BoxCoalgExp .map ticks cont
  putStrLn "waiting"
  getLine >>= \case
    "" =>
      let cont'' = BoxCoalgExp .map (Later Now) cont'
      in eval (S t') [< waiting (S t') cont'', ()]
    _  => eval t' [< cont', ()]
```

- We first bring the continuation into the future, if we don't, type error
- If we wait, then we step one more time, bringing the continuation into the future again by one more step

Ticking with Kripke trees: handler

- Our algebra is also analogous:

```
grabOp : FamProd [< Ticky -% Clocked, const ()] -|> Clocked
grabOp t [< cont, ()] = eval t [< cont, Clock 0]

emitOp : FamProd [< const () -% Clocked, Ticky] -|> Clocked
emitOp t [< cont, Clock i] = printLn i >> eval t [< cont, ()]

waitOp : FamProd [< const () -% Clocked, const ()] -|> Clocked
waitOp t [< cont, ()] = eval t [< waiting t cont, ()]

ClockedAlgebra : (OpTime).AlgebraOver Clocked
ClockedAlgebra = MkAlgebraOver {sig = OpTime} $ \case
  Grab => grabOp
  Emit => emitOp
  Wait => waitOp
```

Ticking with Kripke trees: running

- And finally we can run it starting at time 0:

```
RunExTime : IO ()
RunExTime =
  (ClockedAlgebra).fold (\_, _ => putStrLn "done") Z (ExTime Z [<])
```

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)

Bibliography


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