Effects

IO, exceptions, states, concurrency, backtracking, etc.

Problem: ad-hoc implementation, unsound combinations

What are effect handlers? A **composable** and **structured** control-flow abstraction.

But why not monads? **Algebraic effects** are more *composable*, becuase they **abstract the details** on how effects are handled **away**, while to reason on **monads** one need to use *transformers* to actually look into the details.

Of course, monads are more expressive because they don't need to be algebraic.

Definition

An **algebraic effect** is:

- operations (effect constructors) with *signatures*
- *axioms*

E.g. *one boolean location*

Signature: `put_t: 1, put_f: 1, get: 2`

- $\text{put}_b(m)$: put b (t/f) then continue with m
- get (m, n): get the value then continue with m if true, n if false

Axioms:

- `get(m, m) = m`
- `get(get(m, m'), get(n, n')) = get(m, n')` (load elim)
- `put_b(put_b'(m)) = put_b'(m)` (store elim)
- `get(put_t(m), put_f(n)) = get(m, n)` (load store elim)
- `put_b(get(m_t, m_f)) = put_b(m_b)` (store load elim)

NOTE When one tries to implement an algebraic effect, one usually comes up with axioms, then tries to implement the effect in a way that respects the axioms, instead of the other way around.

Interpretation (semantics)

$$T_{b(X)} = B \to (X \times B)$$

$$\begin{aligned} [\![c]\!] &= \lambda s.(c, s) \\ [\![\text{get}(m, n)]\!] &= \lambda s.\ \text{if}\ s\ \text{then}\ [\![m]\!]s\ \text{else}\ [\![n]\!]s \\ [\![\text{put}_{b(m)}]\!] &= \lambda s.[\![m]\!]b \end{aligned}$$

Theorem: the interpretation is *sound* and *complete* with respect to the *axioms*.

$$m = n \Leftrightarrow [\![m]\!] = [\![n]\!]$$

Proof:

- $\Rightarrow$: by direct case analysis on axioms and expanding the interpretation.
- $\Leftarrow$: by induction on the structure of axioms.

NOTE P.S. It can be proven that the `get-get` equation is redundant.

Interpretation B

This time, we keep track of all history states.

$$T_{\text{log}}(X) = B \to (X \times \text{list } B)$$

$$[\![c]\!] = \lambda s.(c, [s])$$

$$[\![\text{get}(m, n)]\!] = \lambda s.\ \text{if } s \text{ then } [\![m]\!]s \text{ else } [\![n]\!]s$$

$$[\![\text{put}_{b(m)}]\!] = \lambda s'.\ \text{let } (n, s) \leftarrow [\![m]\!] \text{ in } (n, b :: s)$$

Theorem: the interpretation is *complete* with respect to the *axioms*.

$$m = n \Leftarrow [\![m]\!] = [\![n]\!]$$

Proof: same as before.

Theorem: the interpretation is **unsound** with respect to the *axioms*.

$$m = n \nRightarrow [\![m]\!] = [\![n]\!]$$

Proof: consider $\text{put}_b(\text{put}_{b'}(m)) \neq \text{put}_{b'}(m)$.

Interpretation C

Let's throw away the state completely. One can see it's *sound*, of course, but it's *incomplete* because it equates all programs as if there's no state so it recognizes much more programs than the axioms allow.

e.g. exception

Signature `raise_e: 0 (e ∈ E)`

Axioms None

Interpretation `T_e(X) = X + e`. Return is `inl` and `raise` is `inr`.

e.g. non-determinism

NOTE ... skiped for brevity

But how to tell if a set of axioms is good enough?

- **equationally inconsistent**: $\forall x, y, x = y$. Explosion! This is what we want to avoid: unsound.
- **Hilbert-Post complete**: adding any *unprovable* equation makes it *equationally inconsistent*. This means our set of axioms is very complete.

How are algebraic effects *algebraic*?

~~TODO~~ ~~Insert fancy commute diagram here~~[1]

Instead of a categorical definition, in a program sense, *algebraic* means:

Assume the notion of *evaluation context* (the $\square \mid E\ n \mid (\lambda x.m)E$ thing),
for any `op: n`, $E[\text{op}(m_1, ..., m_n)] = \text{op}(E[m_1], ..., E[m_n])$

NOTE i.e. E and op commute

Computational Trees

Effectful programs can be represented as *computational trees*, trees whose *leaves* are *values* and *internal nodes* are *operations*.

TODO put the second example figure here

[1]Plotkin, Gordon, and John Power. 2001. 'Semantics for Algebraic Operations'. Electronic Notes in Theoretical Computer Science 45: 332–45. [PDF]

i.e. `get(or(raise, t), put_t(f))`

NOTE *Interaction tree* is a coinductive version of computational tree.

As free monads

```haskell
data Free f a = Pure a | Free (f (Free f a))
```

- `Pure a` (*triangle*) - pure value
- `Free (f (Free f a))` (*rect*) - an op that produces another `Free f a` computation

```haskell
return c >>= r = r c
op(m1, ..., mn) >>= r = op(m1 >>= r, ..., mn >>= r)
```

NOTE One can see that the binding operation behaves (not by coincidence) super similar to the very definition of *algebraic* effects.

Parametrized

Parametrized Operations

Motivation: Consider if we want to generalize the single location boolean effect to location indexed by countable *loc* and also we want to store nat instead. It's infeasible to define infinite operations and axioms.

e.g.

```haskell
update: loc x nat ~> 1
lookup: loc ~> nat
```

Parametrized Arguments

Problem previously, we have `get(m, n)` where `m` is for `true` and `n` is for `false`. However, now we are trying to store a nat, which means we essentially need to provide infinite branches!

Solution: we can use *parametrized arguments*.

e.g.

```haskell
lookup(l, λx. nat. m)
```

Q: *how are we going to define algebraic in this case?*

$E[\mathtt{op}\ (p, \lambda x : n.m)] = \mathtt{op}\ (p, \lambda x : n.E[m])$

NOTE E is not squashed into p in this case. TODO why?

e.g. $\text{lookup}(l, \lambda x.m)n = \text{lookup}(l, \lambda x.mn)$

Generic Effects

Motivation TODO

$$\frac{m : \text{loc} \times \text{nat}}{\text{gen\_update}\ m : 1} \qquad \frac{n : \text{loc}}{\text{gen\_lookup}\ n : \text{nat}}$$

`lookup(l, λ x: nat. m)  vs  gen_lookup(l): nat`

NOTE Intuitively, they are just a let binding away.

1. `gen_update(l, 42) ≡ update(l, 42), λx. x)`
2. `update((l, 42). λx. m) ≡ let x = gen_update((l, 42)) in m`

Example Calculus

Syntax

Imagine *STLC* with bools and `if` statements, formated in a contextual semantics way (eval ctx), but with `op  e` in terms.

NOTE `op` is not a value.

We'll extend it with handlers in the next subsection.

e.g.

```haskell
choose: () ~> bool
fail: () ~> a

drunkToss () =
  if choose () then
    if choose () then Heads else Tails
  else fail ()
```

Effect Handler

Syntax

```haskell
  handle { op ↦ λx k. e₁, return ↦ λx. e₂ } e
```

- `x` is the operation argument
- `k` is the *delimited continuation*
- `return` is for when the omputation returns a pure value

Handlers are *terms*, but not *values*.

E.g.

```haskell
maybeFail = {
  fail ↦ λx k. Nothing,   -- if fail, return Nothing. We are changing the type of the computation to Maybe a
  return ↦ λx. Just x     -- if return, return Just x
}

trueChoice = {
  choose ↦ λx k. k true,  -- we resume the computation with `true`
  return ↦ λx. x           -- we just return the value.
}

allChoices = {
  choose ↦ λx k. k true ++ k false,  -- we resume the computation twice with either `true` or `false`
```

```
    return ↦ λx. [x]  -- we return a list of values. Note how we changed the type of the computation to List a
}
```

TODO maybe it would be better to directly give the non-det choose example so we can explain k and type change at the same time

Delimited Continuation

Assume one understands what's a continuation,

a *delimited continuation* is a continuation that captures the control flow up to **a certain point**, i.e. it **does not capture the whole program** but only the part that is relevant to the current effect.

In the setting of effect handlers, this delimited continuation captures from where the operation is called to where the handler is in the eval ctx.

E.g.

```haskell
handle h E [op v]      -- where op is fresh in E
```

- `x`: `v`
- `k`: `λx. handle h E[x]`

This continuation delimits to the handler's scope. It does not handle any operation outside of the handler term.

Handler composition

When we need to compose two handlers, the behavior can be different depending on the order of composition.

```haskell
handle allChoices (handle maybeFail (drunkToss ()))
  -> [Just Heads, Just Tails, Nothing]
```

, but

```haskell
handle maybeFail (handle allChoices (drunkToss ()))
  -> Nothing
```

One can make sense of this by looking into the return type of the handlers:

- **maybeFail** changes the return type from `a` to `Maybe a`
- **allChoices** changes the return type from `a` to `List a`

So,

- **allChoices** ∘ **maybeFail** changes the return type from `a` to `List (Maybe a)`
- **maybeFail** ∘ **allChoices** changes the return type from `a` to `Maybe (List a)`

Dynamics

$$\text{handle } h\ v \longrightarrow f\ v \quad \text{where } \texttt{return} \mapsto f \in h$$

$$\text{handle } h\ E[\texttt{op}\ v] \longrightarrow f\ v(\lambda x.\ \text{handle } hE[x]) \quad \text{where } \texttt{op} \mapsto f \in h, \texttt{op}\ \#E$$

NOTE $\texttt{op}\ \#E$ means nothing inside E is capturing `op`.

E.g. cooperative threads

Alternatives

Shallow

$$\text{handle } h\ E[\texttt{op}\ v] \rightarrow f\ v(\lambda x.E[x]) \quad \text{where } \texttt{op} \mapsto f \in h, \texttt{op} \ \# E$$

NOTE handler is not re-installed

e.g. Unix pipeline

```haskell
pipe(p, c) = handle { await ↦ λx k. copipe(k, p) } (c ())
copipe(k, p) = handle { yield ↦ λx k. pipe (k, λ_. c x) } (p ())
```

In this case, we are using shallow handlers to alternate between two handlers.

Sheep

$$\text{handle } h\ E[\texttt{op}\ v] \rightarrow f\ v\ (\lambda h'.\lambda x.\ \text{handler } h'\ E[x]) \quad \text{where } \texttt{op} \mapsto f \in h, \texttt{op} \ \# E$$

NOTE *Users* can choose to install a new handler `h'`. Of course, one can always choose to install the same handler `h` again, or install nothing at all.

It's what's implemented in *WASM* lol.

Parametrized

$$\text{handle } h\ s\ E[\texttt{op}\ v] \rightarrow f\ v(\lambda s'.\lambda x.\ \text{handler } h\ s'\ E[x]) \quad \text{where } \texttt{op} \mapsto f \in h, \texttt{op} \ \# E$$

NOTE It has the same expressiveness as deep handlers.

Masking By `lift[op] e` one could specify to skip one handler when handling `op` in `e`.

Named A name is attached to both the handler and the operation, so one can specify which handler to use for a particular operation term.

Effect Type System

Syntax

$$\begin{aligned} &\text{effect labels } \ell \\ &\text{effects } \varepsilon \quad := \langle\rangle \mid \langle \ell \mid \varepsilon \rangle \\ &\text{types } A := \text{Int} \mid \text{Bool} \mid A \xrightarrow{\varepsilon} B \end{aligned}$$

NOTE We have multiple choices with regard to the model of effects. It can be sets, simple rows, scoped rows, or some other stuff.

$A \xrightarrow{\varepsilon} B$: a function taking an A that **may perform effects** ε and returns a value of type B.

Judgment

$\boxed{\Gamma \vdash e : A \mid \varepsilon}$ a term e of type A in context Γ that may perform effects ε.

$$\frac{}{\Gamma \vdash \text{True} : \text{Bool} \mid \varepsilon}\ \text{T} \qquad \frac{\Gamma, x : A \vdash e : B \mid \varepsilon}{\Gamma \vdash \lambda x.e : A \xrightarrow{\varepsilon} B \mid \varepsilon'}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : A \xrightarrow{\varepsilon} B \mid \varepsilon \qquad \Gamma \vdash e_2 : A \mid \varepsilon}{\Gamma \vdash e_1\ e_2 : B \mid \varepsilon}\ \textsc{App}$$

For T (consts), we allow arbitrary effects so that we can "pretend" it performs ε. This phenomenon is called *effect pollution*.

For Abs, we allow arbitrary effects $\mid \varepsilon'$, i.e. it doesn't need to be the same as ε, because a function itself does not perform any effects. For App we require the effects of the function and its argument to match.

The reason why we model it as rows is because we can actually deal with effects with *row unification* only, instead of complex set reasoning.

TODO Op, Handle, and Handle judgment

Theorem (*Type Preservation*). If $\Gamma \vdash e : A \mid \varepsilon$, and $e \to e'$, then $\Gamma \vdash e' : A \mid \varepsilon$.

Conjecture (*Progress?*). If $\vdash e : A \mid \varepsilon$, then either e is a value or there exists e' such that $e \to e'$.

WARN What if some effect in ε is not handled?

Theorem (*Progress with effects*). If $\vdash e : A \mid \varepsilon$, then either e is a value or there exists e' such that $e \to e'$, or $e = E[\mathrm{op}\ v]$ where $\mathrm{op} \in \varepsilon$ and $\mathrm{op}\ \#E$, i.e. E does not handle op so the computation is stuck.

Collary (*Progress*). If $\vdash e : A \mid \langle\rangle$, then either e is a value or there exists e' such that $e \to e'$.

Runtime Implementation

Problem: the naive implementation is slow.

$$\text{handle } h\ E[\texttt{op}\ v] \to f\ v(\lambda x.\ \text{handle } hE[x]) \quad \text{where } \texttt{op} \mapsto f \in h, \texttt{op}\ \#E$$

One need to first *capture* the continuation and then *search* the handler, both of which are expensive.

Solutions:

- **CPS** (*Lean*) - Using closures to capture the continuation, but still one needs to pay for closure allocation.
- **Segmented stacks** (*OCaml*) - Very efficient handling of one-shot resumption.

  Stacks are segmented into *fibers* with *handlers* as dividers. Once an action is met, the continuation is stuffed into the deepest fiber.
- **Capability-passing style** (*Effekt, Scala*) - Efficient lexically scoped handlers, and has a slightly different semantics. Handlers are decided lexically, thus efficient, but care must be taken to ensure handlers does not escape its scope at runtime.
- **Rewriting** (*Eff*) - Source-to-source transformation
- **Evidence-passing semantics** (*Koka*) - Pushing down handlers to the action call-site instead of searching for them. *Tail-resumption* (*tail-call* for handlers) is handled ad-hoc to allow in-place tail resumption, eliminating continuation capturing.