Notes on "What is Algebraic About Algebraic Effects and Handlers"

Lemon

May 3, 2025

Contents

1 Algebraic Effects and Continuations

This note is taking while reading Andrej Bauer's writing (??, ????) Currently, this note only limits to the first 3 section, and only limit to the algebraic computation part

1.1 What is a "Free Tree"

Given a set of operations $\{\mathrm{op_i}\}$ with general arity $\{\mathrm{A_i}\}$ and parameter $\{\mathrm{P_i}\}$, The definition states that the set $\mathrm{Tree}_\Sigma(\mathrm{X})$ is generated inductively by the following two ways:

1. For every $\mathrm{v} \in \mathrm{X}$, $(\mathrm{return\ v}) \in \mathrm{Tree}_\Sigma(\mathrm{X})$

2. For every $\kappa : \mathrm{A_i} \to \mathrm{Tree}_\Sigma(\mathrm{X})$ and $\mathrm{p} \in \mathrm{P_i}$, $\mathrm{op_i}(\mathrm{p}; \kappa) \in \mathrm{Tree}_\Sigma(\mathrm{X})$

Unlike algebra of operations with finite arity, I could not imagine the "well-foundedness" of this definition Thus, I consult to proof assistant. What follows is my attempt at defining the Free Tree in Coq:

```
Inductive FreeTree
  {X : Type}
  {param_map : nat -> Type}
  {arity_map : nat -> Type} : Type :=
| ret (v : X)
| op (index : nat) (p : param_map index) (k : arity_map index -> FreeTree).
```

It may be clearer now how we can construct an element of $\mathrm{Tree}_\Sigma(\mathrm{X})$:

- Initially we only have (ret v)
- We can now build functions of type $\mathrm{A_i} \to \mathrm{Tree}_\Sigma(\mathrm{X})$, except the function will always return (ret v) for some $\mathrm{v} \in \mathrm{X}$
- But how with the function, we can use ($\mathrm{op_i}$) to construct trees of different form
- Build up from there

1.2 What Each Part of the Tree Represents

Let us discuss $\mathrm{Free}_\Sigma(\mathrm{X})$ represents

1.2.1 $\mathbf{Free}_\Sigma(\mathbf{X})$

This is supposed to represents a computation that outputs a value of type `X`, with the potential to call operations with side effects from the signature Σ

1.2.2 return v

This corresponds to the `return` from Haskell's monad definition. For every value in the type `X`, we put it into an "effectful" context

1.2.3 $\mathbf{op_i}(\mathbf{p}\ ;\ \kappa)$

i Identifies the operation

p Operation parameter

κ The continuation for the remaining computation

κ is supposed to represent the continuation of the operation call. If we look back at signature, you can see each operation is defined as:

$\mathrm{op_i} : \mathrm{P_i} \text{ -> } \mathrm{A_i}$

This tells us that operator expects some parameters. When it received all the parameters, it will produce a value of type $\mathrm{A_i}$ Now looking back at κ, we see it indeed represents the continuation after the operation call.

1.3 Lifting Functions to Homomorphisms

The free model, have the universal lifting property:

$$\begin{tikzcd} X \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathsf{Tree}_{\Sigma}(X) \arrow[d, "\overline{f}"] \\ & M \end{tikzcd}$$

In particular, $\overline{f}$ is defined "inductively":

- $\overline{f}(return\ v) = f(v)$
- $\overline{f}(op_i(p, \kappa) = op_i(p; \lambda a.\overline{f}(\kappa(a))) = op_i(p; \overline{f} \circ \kappa)$

Given elements of $\mathrm{Tree}_{\Sigma}(\mathrm{X})$ are "well-founded", the recursive definition of $\overline{f}$ will eventually terminates. Coq also accepts the definition:

```
Fixpoint lifting (X Y : Type) (param_map : nat -> Type) (arity_map : nat -> Type)
  (f : X -> @FreeTree Y param_map arity_map)
  (tree : @FreeTree X param_map arity_map) :
  @FreeTree Y param_map arity_map :=
  match tree with
  | ret v => f v
  | op i p k => op i p (fun a => lifting X Y param_map arity_map f (k a))
  end.
```

In particular, we can specialize M to $\mathrm{Tree}_{\Sigma}(\mathrm{Y})$ for the same signature, and consider some f : X $\rightarrow$ $\mathrm{Tree}_{\Sigma}(\mathrm{Y})$, Such f get can get lifted to $\overline{f}$: $\mathrm{Tree}_{\Sigma}(\mathrm{X})$ $\rightarrow$ $\mathrm{Tree}_{\Sigma}(\mathrm{Y})$.

1.4 Pushing Continuation In

Let us now look at some more familiar notations that will thread the ideas together

1.4.1 let ... in ... Notation

We first make a few assumptions:

- x ∈ X
- **x is bounded in e**
- e ∈ Tree_Σ(Y)

Now, let us consider the following block of code:

```
do x <- op(p ; \y -> c) in e
-- or
let x = op(p ; \y -> c) in e
```

λ **x. e** now behaves like a function of type X → Tree_Σ(Y), so we and we **define** the following:

- $\lambda r.\ \texttt{do}\ x \leftarrow r\ \texttt{in}\ e \equiv \overline{\lambda x.e}$
- $\lambda r.\ \texttt{let}\ x = r\ \texttt{in}\ e \equiv \overline{\lambda x.e}$

With the previously defined definition of lifting, we now get the following property through

$$
\begin{aligned}
\texttt{do}\ x \leftarrow \mathrm{op}(p;\lambda y.c)\ \texttt{in}\ e &= (\lambda r.\ \texttt{do}\ x \leftarrow r\ \texttt{in}\ e)(\mathrm{op}(p;\lambda y.c)) \\
&= \overline{(\lambda x.e)})\,(\mathrm{op}(p;\lambda y.c)) && \text{(Definition)} \\
&= \mathrm{op}(p;\overline{(\lambda x.e)} \circ (\lambda y.c)) && \text{(Lifted homomorphism)} \\
&= \mathrm{op}(p;\lambda y.\overline{(\lambda x.e)}(c)) \\
& \qquad\qquad \text{(We have to assume } y \notin \mathsf{fv}(\lambda x.e)\text{)} \\
&= \mathrm{op}(p;\lambda y.(\lambda r.\ \texttt{do}\ x \leftarrow r\ \texttt{in}\ e)\,(c)) \\
& && \text{(Definition)} \\
&= \mathrm{op}(p;\lambda y.\ \texttt{do}\ x \leftarrow c\ \texttt{in}\ e)) && \text{(Definition)}
\end{aligned}
$$

A similar process can be done for `let ... in ...`

$$\begin{aligned}
\texttt{let}\ x = \mathrm{op}(p; \lambda y.c)\ \texttt{in}\ e &= (\lambda r.\ \texttt{let}\ x = r\ \texttt{in}\ e)(\mathrm{op}(p; \lambda y.c)) \\
&= \overline{(\lambda x.e)}(\mathrm{op}(p; \lambda y.c)) \\
&= \mathrm{op}(p; \overline{(\lambda x.e)} \circ (\lambda y.c)) \\
&= \mathrm{op}(p; \lambda y.\overline{(\lambda x.e)}(c)) \\
&= \mathrm{op}(p; \lambda y.(\lambda r.\ \texttt{let}\ x = r\ \texttt{in}\ e)\,(c)) \\
&= \mathrm{op}(p; \lambda y.\ \texttt{let}\ x = c\ \texttt{in}\ e)
\end{aligned}$$

1.4.2 Monadic Bind

We will do a similar process for monadic bind `>>=`.

Let us first examine the type signature of `>>=` in Haskell:

```
(>>=) : (Monad m) => (M a) -> (a -> M b) -> (M b)
```

We understand `M b` as a computation that gives a value of type `b`, with possible side effects represented by the monad `M`.

Let us instantiate `(M -)` = Tree_Σ(-). This turns the type of the second argument of `(>>=)` into (a $\to$ Tree_Σ(b)).

What is the meaning of `(>>=)`? it is precisely generating the unique lifting of (a $\to$ Tree_Σ(b)) to (Tree_Σ(a) $\to$ Tree_Σ(b)) (Well, more precisely, that is the job of `flip (>>=)`)

Thus we can define the following

$$\lambda r.r \ \ggg\!=\ f = \overline{f}$$

And get the following equations.

$$\begin{aligned}
\mathrm{op}(p; \lambda y.c)\ \ggg\!=\ f &= (\lambda r.r\ \ggg\!=\ f)(\mathrm{op}(p; \lambda y.c) \\
&= \overline{f}(\mathrm{op}(p; \lambda y.c) \\
&= \mathrm{op}(p; \overline{f} \circ (\lambda y.c)) \\
&= \mathrm{op}(p; \lambda y.\overline{f}(c)) \\
&= \mathrm{op}(p; \lambda y.(\lambda r.r\ \ggg\!=\ f)(c)) \\
&= \mathrm{op}(p; \lambda y.c\ \ggg\!=\ f)
\end{aligned}$$

1.5 What's the Significance?

The reason you should care about these is because it makes reading the dynamic semantic of any effect handler paper much easier.

When I first started reading effect handlers papers, I didn't know why the small-step semantic of $\mathsf{op}_\mathsf{i}(\dots)$ behaved the way it did:

- Why are we capturing the continuation?
- Why can we push the continutation into the operation?

Andrej's paper (??,) exposes the algebraic reasoning behind those design decisions.