Applications of algebraic effect theories

Žiga Lukšič
Mentor: doc. dr. Matija Pretnar

9. 5. 2019

Reasoning about programs

$$f_1 = \texttt{fun}\ x \mapsto x + x$$
$$f_2 = \texttt{fun}\ x \mapsto 2 * x$$

We know mathematical properties of operations $+$ and $*$ so we can argue that the functions are equal (whatever that means).

Harder with effectful operations, such as *print*.

$$\begin{aligned} \mathit{print\_twice} &= \texttt{fun}\ s \mapsto \mathit{print}\ s;\ \mathit{print}\ s \\ \mathit{print\_double} &= \texttt{fun}\ s \mapsto \mathit{print}\ s \,\hat{}\, s \end{aligned}$$

“Implementing print”

We give meaning to operations with handlers. An example:

$$
\begin{aligned}
&\mathit{collect\_prints} \ = \ \mathtt{handler}\ \{\\
&\quad \mid \mathit{print}(s;\ k) \mapsto\\
&\qquad\quad \mathtt{do}\ (v, s') \leftarrow k\ ()\ \mathtt{in}\ (v, s\,\hat{}\,s')\\
&\quad \mid \mathtt{ret}\ x \mapsto (x, "")\\
&\}
\end{aligned}
$$

What happens when we combine them?

What happens when we combine them?

$$\texttt{with } \textit{collect\_prints} \texttt{ handle } (\textit{print\_twice } \text{"}\textit{test}\text{"})$$

What happens when we combine them?

$$\texttt{with}\ \mathit{collect\_prints}\ \texttt{handle}\ (\mathit{print\_twice}\ \text{"}\mathit{test}\text{"})$$

When the first *print* occurs, our handler ‘intercepts’ the operation call

$$\mathit{print}(\text{"}\mathit{test}\text{"};\ ().\mathit{print}\ \text{"}\mathit{test}\text{"})$$

What happens when we combine them?

$$\texttt{with } \mathit{collect\_prints} \texttt{ handle } (\mathit{print\_twice} \text{ "}\mathit{test}\text{"})$$

When the first *print* occurs, our handler 'intercepts' the operation call

$$\mathit{print}(\text{"}\mathit{test}\text{"};\, ().\mathit{print} \text{ "}\mathit{test}\text{"})$$

It 'activates' the *print* branch of the handler and it first wraps the continuation

$$k = \texttt{fun } () \mapsto \texttt{with } \mathit{collect\_prints} \texttt{ handle } (\mathit{print} \text{ "}\mathit{test}\text{"})$$

What happens when we combine them?

$$\texttt{with}\ \mathit{collect\_prints}\ \texttt{handle}\ (\mathit{print\_twice}\ \text{"}\mathit{test}\text{"})$$

When the first *print* occurs, our handler 'intercepts' the operation call

$$\mathit{print}(\text{"}\mathit{test}\text{"}; ().\mathit{print}\ \text{"}\mathit{test}\text{"})$$

It 'activates' the *print* branch of the handler and it first wraps the continuation

$$k = \texttt{fun}\ ()\mapsto \texttt{with}\ \mathit{collect\_prints}\ \texttt{handle}\ (\mathit{print}\ \text{"}\mathit{test}\text{"})$$

and proceeds with the evaluation

$$\begin{aligned}&\texttt{do}\ (v, s') \leftarrow k\ ()\ \texttt{in}\\ &(v, \text{"}\mathit{test}\text{"}\ \hat{}\ s')\end{aligned}$$

When using the handler *collect_prints* the functions *print_twice* and *print_double* are indeed equal.

This does not hold for arbitrary *print* handlers.

When using the handler *collect_prints* the functions *print_twice* and *print_double* are indeed equal.

This does not hold for arbitrary *print* handlers.

We want reasoning tools that can separate handlers from the rest of our code!

The theory of algebraic effects consists of operations and **equations** between them.

We can use equations such as

$$\mathit{print}\ s_1;\ \mathit{print}\ s_2 \sim \mathit{print}\ s_1\,\hat{}\ s_2$$

as reasoning tools.

The theory of algebraic effects consists of operations and **equations** between them.

We can use equations such as

$$\textit{print } s_1;\ \textit{print } s_2 \sim \textit{print } s_1\hat{}\ s_2$$

as reasoning tools.

Can we achieve that without restricting ourselves to a global effect theory?

Idea

Include the desired equations in types.

$$\underline{C} = A!\Sigma/\mathcal{E}$$

Operations of type $\underline{C}$ either return a value of type A or call an operation from Σ in the effect theory $\mathcal{E}$.

Equations in $\mathcal{E}$ tell us what computations we deem equal.

Term Syntax

$$
\begin{array}{rcll}
\text{values } v & ::= & x & \text{variable} \\
& \mid & (\,) & \text{unit constant} \\
& \mid & \texttt{true} \mid \texttt{false} & \text{boolean constants} \\
& \mid & \texttt{fun}\ x \mapsto c & \text{function} \\
& \mid & \texttt{handler}\ (\texttt{ret}\ x \mapsto c_r; h) & \text{handler} \\
\\
\text{computations } c & ::= & \texttt{if}\ v\ \texttt{then}\ c_1\ \texttt{else}\ c_2 & \text{conditional} \\
& \mid & v_1\ v_2 & \text{application} \\
& \mid & \texttt{ret}\ v & \text{returned value} \\
& \mid & op(v;\ y.c) & \text{operation call} \\
& \mid & \texttt{do}\ x \leftarrow c_1\ \texttt{in}\ c_2 & \text{sequencing} \\
& \mid & \texttt{with}\ v\ \texttt{handle}\ c & \text{handling} \\
\\
\text{operation clauses } h & ::= & \emptyset \mid h \cup \{op(x;\ k) \mapsto c_{op}\} &
\end{array}
$$

Type Syntax

$$
\begin{array}{rcll}
\text{(value) type } A, B & ::= & \texttt{unit} & \text{unit type} \\
& | & \texttt{bool} & \text{boolean type} \\
& | & A \to \underline{C} & \text{function type} \\
& | & \underline{C} \Rightarrow \underline{D} & \text{handler type} \\
\\
\text{computation type } \underline{C}, \underline{D} & ::= & A!\Sigma/\mathcal{E} & \\
\\
\text{signature } \Sigma & ::= & \emptyset \mid \Sigma \cup \{op : A \to B\} &
\end{array}
$$

Type Syntax (additions)

$$
\begin{array}{rcl}
\text{value context } \Gamma & ::= & \varepsilon \mid \Gamma, x : A \\[1em]
\text{template context } \mathtt{Z} & ::= & \varepsilon \mid \mathtt{Z}, z : A \to * \\[1em]
\text{template } T & ::= & z\ v \\
& \mid & \texttt{if } v \texttt{ then } T_1 \texttt{ else } T_2 \\
& \mid & op(v;\ y.T) \\[1em]
\text{(effect) theory } \mathcal{E} & ::= & \emptyset \mid \mathcal{E} \cup \{\Gamma; \mathtt{Z} \vdash T_1 \sim T_2\}
\end{array}
$$

The *any type* $*$ used in template types can be instantiated to any computation type so that we can reuse templates.

Full equation

$$\Gamma; \mathtt{Z} = (x : \mathtt{string}, y : \mathtt{string}); (z : \mathtt{unit} \to *)$$

$$\Gamma; \mathtt{Z} \vdash \mathit{print}(x; \_.\mathit{print}(y; \_.z\ ())) \sim \mathit{print}(x \mathbin{\hat{}} y; \_.z\ ())$$

Assigning types

Assume a slightly richer language (with strings and pairs).

Without equations, *collect_prints* would have the type

$$A!\{print\} \Rightarrow (A * string)!\emptyset$$

Assigning types

Assume a slightly richer language (with strings and pairs).

Without equations, *collect_prints* would have the type

$$A!\{print\} \Rightarrow (A * string)!\emptyset$$

But before we wanted to use the theory

$$E_{no\_sep} = (print\ s_1;\ print\ s_2 \sim print\ s_1 \mathbin{\hat{}}\ s_2)$$

so we could instead use the type

$$A!\{print\}/E_{no\_sep} \Rightarrow (A * string)!\emptyset/\emptyset$$

Another example

$$
\begin{aligned}
&\mathit{use\_newline} \;=\; \texttt{handler}\, \{ \\
&\quad \mid \mathit{print}(s;\ k) \mapsto \mathit{print}\ s;\ \mathit{print}\ "\backslash n";\ k\ () \\
&\quad \mid \texttt{ret}\ x \mapsto x \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
E_{no\_sep} &= \mathit{print}\ s_1;\ \mathit{print}\ s_2 \sim \mathit{print}\ s_1 \mathbin{\hat{}} s_2 \\
E_{sep} &= \mathit{print}\ s_1;\ \mathit{print}\ s_2 \sim \mathit{print}\ s_1 \mathbin{\hat{}} "\backslash n" \mathbin{\hat{}} s_2
\end{aligned}
$$

$$\mathit{use\_newline} : A!\{\mathit{print}\}/E_{sep} \Rightarrow (A)!\{\mathit{print}\}/E_{no\_sep}$$

How to type handlers?

All typing rules remain largely the same as when not using equations.

The only important change is

$$\frac{\Gamma, x : A \vdash c_r : \underline{D} \qquad \Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E}}{\Gamma \vdash \texttt{handler}\,(\texttt{ret}\ x \mapsto c_r; h) : A!\Sigma/\mathcal{E} \Rightarrow \underline{D}}$$

How to type handlers?

We must check $\Gamma \vdash h\colon \Sigma \Rightarrow \underline{D}$ respects $\mathcal{E}$ in a logic, but we have a choice which logic to use.

The general idea is checking that our handler maps equivalent computations to equivalent computations.

The choice of logic also bears impact on the denotational semantics of our language.

Future work

- Implement the new type system.
- Consider options for easier checks of the respects relation.
- How to define subtyping?
- How to use such systems for formalisation (probabilistic programming)?