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.

Harder with effectful operations, such as *get*, which reads a value from memory.

$$g_1 = \texttt{fun}\ () \mapsto get\ () + get\ ()$$
$$g_2 = \texttt{fun}\ () \mapsto 2 * get\ ()$$

Reasoning about programming

When working with algebraic effects we can use the equations that hold between operations.

$$\texttt{do}\ \_ \leftarrow \mathit{get}\,()\ \texttt{in}\ \mathit{get}\,() \sim \mathit{get}\,()$$

We can prescribe a global effect theory with the above equation.

But this hinders the use of handlers

Idea

Make equations great again!

Reintroduce equations into algebraic effects and handlers by including them 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 (nothing new)

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

Type Syntax (mostly old stuff)

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

Type Syntax (new stuff)

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

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

Example

We have written a program using nondeterministic choice $\mathit{choose} : () \to \texttt{bool}$. We obtain a binary non-deterministic choice from the abbreviation:

$$c_1 \oplus c_2 \stackrel{\text{def}}{=} \mathit{choose}((); y.\texttt{if } y \texttt{ then } c_1 \texttt{ else } c_2)$$

We didn't pay any attention to the order of arguments of $\oplus$ so we wish to make sure that the arguments commute when evaluated

$$\emptyset; z1, z2 \vdash z_1 \oplus z_2 \sim z_2 \oplus z_1 \qquad (\textsc{comm})$$

and so we give our program the type $\mathit{nondetProg} : \texttt{int}!\{\mathit{choose}\}/(\textsc{comm})$

Now we want to play with our program, but don't want to write all the handlers ourselves!!!

So we find a library for working with $\mathit{yield} : \mathtt{int} \to \mathtt{unit}$ and in it a handler

$$\mathit{sumYielded} : \mathtt{unit}!\{\mathit{yield}\}/(\textsc{order}) \Rightarrow \mathtt{int}!\emptyset/\emptyset$$

which does not care about the order of yielded values, as expressed by

$$\mathit{yield}(x; \_.\mathit{yield}(y; \_.z)) \sim \mathit{yield}(y; \_.\mathit{yield}(x; \_.z)) \qquad (\textsc{order})$$

To go from *choose* to *yield* we write a handler that yields all possible outcomes of our program

$$
\begin{aligned}
&\mathit{yieldAll} \;=\; \texttt{handler}\;\{\\
&\quad \mid \mathit{choose}((\,);\; k) \mapsto k\;\texttt{true};\, k\;\texttt{false}\\
&\quad \mid \texttt{ret}\; x \mapsto \mathit{yield}(x;\; \_.\texttt{ret}\;(\,))\\
&\}
\end{aligned}
$$

It clearly has the type $\texttt{int}!\{\mathit{choose}\}/\emptyset \Rightarrow \texttt{unit}!\{\mathit{yield}\}/\emptyset$, but due to the type of our program, any handler used on *nondetProg* needs to respect $(\textsc{comm})$.

Typing rules

$$\frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{with}\ v\ \mathtt{handle}\ c : \underline{D}}$$

When handling computations, the equations in the types must match as well.

Most typing rules are largely unchanged. The only interesting rule is for typing handlers.

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

The typing part of $\Gamma \vdash h : \Sigma \Rightarrow \underline{D}$ respects $\mathcal{E}$ is as expected

$$\frac{}{\Gamma \vdash \emptyset : \emptyset \Rightarrow \underline{D}}$$

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \qquad \Gamma, x : A_{op}, k : B_{op} \to \underline{D} \vdash c_{op} : \underline{D} \qquad op \notin \Sigma}{\Gamma \vdash h \cup \{op(x;\ k) \mapsto c_{op}\} : (\Sigma \cup \{op : A_{op} \to B_{op}\}) \Rightarrow \underline{D}}$$

but to get the *respects* part we need to use a logic...

We can use different kinds of logics

We can use any logic that implements some kind of respects relation
(though there are requirements for denotational semantics to make sense).

The simplest logic we can use is the free logic, in which

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D}}{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \emptyset}$$

and corresponds to the conventional approach where we ignore equations.

Another option is to use (along with rules for reflexivity, symmetry, transitivity, substitution, congruences for each construct, and $\beta\eta$-equivalences)

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E} \qquad \Gamma, (x_i : A_i)_i, (f_j : B_j \to \underline{D})_j \vdash T_1^h[f_j/z_j]_j \equiv_{\underline{D}} T_2^h[f_j/z_j]_j}{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E} \cup \left\{(x_i : A_i)_i; (z_j : B_j \to *)_j \vdash T_1 \sim T_2\right\}}$$

$$\frac{\Gamma \vdash h : \Sigma \Rightarrow \underline{D}}{\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \emptyset}$$

where for $h = \{op(x;\ k) \mapsto c_{op}\}_{op}$ we define:

$$\begin{aligned}
z_i(v)^h[f_j/z_j]_j &= f_i\ v \\
(\texttt{if}\ v\ \texttt{then}\ T_1\ \texttt{else}\ T_2)^h[f_j/z_j]_j &= \texttt{if}\ v\ \texttt{then}\ T_1^h[f_j/z_j]_j\ \texttt{else}\ T_2^h[f_j/z_j]_j \\
op(v;\ y.T)^h[f_j/z_j]_j &= c_{op}[v/x, (\texttt{fun}\ y \mapsto T^h[f_j/z_j]_j)/k]
\end{aligned}$$

We also construct a rule with which to use the equations of the current theory

$$\frac{\big((x_i : A_i)_i ; (z_j : B_j \to *)_j \vdash T_1 \sim T_2\big) \in \mathcal{E} \qquad \Gamma \vdash v_i : A_i \qquad \Gamma \vdash f_j : B_j \to A!\Sigma/\mathcal{E}}{\Gamma \vdash (T_1[f_j/z_j]_j)[v_i/x_i]_i \equiv_{A!\Sigma/\mathcal{E}} (T_2[f_j/z_j]_j)[v_i/x_i]_i}$$

We can further extend our logic with induction (and quantifiers and hypotheses)

$$\frac{\Gamma \mid \Psi \vdash c : A!\Sigma/\mathcal{E} \qquad \Gamma, x : A \mid \Psi \vdash \varphi(\texttt{ret}\ x) \qquad \Big[\Gamma, x : A_{op}, k : B_{op} \to A!\Sigma/\mathcal{E} \mid \Psi, \big(\forall y : B_{op}.\ \varphi(k\ y)\big) \vdash \varphi(op(x;\ y.k\ y))\Big]_{op : A_{op} \to B_{op}}}{\Gamma \mid \Psi \vdash \forall c : A!\Sigma/\mathcal{E}.\ \varphi(c)}$$

Sadly, proving (in such a logic) that the handler respects $\mathcal{E}$ has to be done by hand (currently).

Typing *yieldAll*

Suppose we use the suggested logic with induction.

It is not possible to give the handler *yieldAll* the type

$$\texttt{int}!\{\mathit{choose}\}/(\textsc{comm}) \Rightarrow \texttt{unit}!\{\mathit{yield}\}/\emptyset$$

because the order of arguments for $\oplus$ influences the order of yielded values.

Typing *yieldAll*

But luckily *sumYielded* works with the theory $(\textsc{order})$ and it is possible (in the logic with induction) to give *yieldAll* the type

$$\texttt{int}!\{\mathit{choose}\}/(\textsc{comm}) \Rightarrow \texttt{unit}!\{\mathit{yield}\}/(\textsc{order})$$

Combining the parts

We can now safely compose

$$\textit{nondetProg} : \texttt{int}!\{\textit{choose}\}/(\textsc{comm})$$

$$\textit{yieldAll} : \texttt{int}!\{\textit{choose}\}/(\textsc{comm}) \Rightarrow \texttt{unit}!\{\textit{yield}\}/(\textsc{order})$$

$$\textit{sumYielded} : \texttt{unit}!\{\textit{yield}\}/(\textsc{order}) \Rightarrow \texttt{int}!\emptyset/\emptyset$$

We typed *yieldAll* without needing the code of either *nondetProg* or *sumYielded* so everything is entirely modular.

Benefits

- Equations are back!
- Reasoning becomes more modular.
- Libraries can provide tools for reasoning via equations.
- Theories are now local, which removes the drawbacks of global theories.