Making Equations Great Again

Danel Ahman

presenting

what **Žiga Lukšič** and **Matija Pretnar** have been up to

Shonan, 25 March 2019

Local Algebraic Effect Theories

Danel Ahman

presenting

what **Žiga Lukšič** and **Matija Pretnar** have been up to

Shonan, 25 March 2019

2009

We type effect terms as $\Gamma; \Delta \vdash T$, where Δ consists of effect variables $w : (\boldsymbol{\alpha})$, according to the following rules:

$$\frac{\Gamma \vdash \boldsymbol{v} : \boldsymbol{\alpha}}{\Gamma; \Delta \vdash w(\boldsymbol{v})} \quad (w : (\boldsymbol{\alpha}) \in \Delta)$$

$$\frac{\Gamma \vdash \boldsymbol{v} : \boldsymbol{\beta} \qquad \Gamma, \boldsymbol{x}_i : \boldsymbol{\alpha}_i; \Delta \vdash T_i \quad (i = 1, \ldots, n)}{\Gamma; \Delta \vdash \mathrm{op}_{\boldsymbol{v}}(\boldsymbol{x}_i : \boldsymbol{\alpha}_i . T_i)_i} \quad (\mathrm{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}}) \ .$$

Next, *conditional equations* have the form $\Gamma; \Delta \vdash T_1 = T_2 \ (\varphi)$, assuming that $\Gamma; \Delta \vdash T_1$, $\Gamma; \Delta \vdash T_2$, and $\Gamma \vdash \varphi : \mathbf{form}$. Finally, a *conditional effect theory* $\mathfrak{E}$ is a collection of such equations; it would be interesting to develop an equational logic for such theories [18].

2016

5.2 Capturing algebraic equations in an effect system

Matija Pretnar (University of Ljubljana, SI)

Equational theories

The main premise of algebraic effects is that effects can be described with an equational theory consisting of a set of operations and equations between them [7]. For example, non-determinism can be described by an operation `choose` and three equations stating its idempotency, commutativity and associativity. Computations returning values from X are then interpreted as elements of the *free model* of such a theory.

16112

2018

3.11 Make Equations Great Again!

Matija Pretnar (University of Ljubljana, SI)

Joint work of Žiga Lukšič, Matija Pretnar

Algebraic effects have originally been presented with equational theories, i.e. a set of operations and a set of equations they satisfy. Since a significant portion of computationally interesting handlers overrides the effectful behaviour in a way that invalidates the equations, most approaches nowadays assume an empty set of equations.

At the Dagstuhl Seminar 16112, I presented an idea in which the equations are represented locally in computation types [1]. In this way, handlers that do not respect all equations are not rejected but receive a weaker type. In the talk, I presented the progress made and questions that remain open.

References

1 Matija Pretnar. Capturing algebraic equations in an effect system. In *Dagstuhl Seminar 16112*, pages 55–57. 2016. DOI: 10.4230/DagRep.6.3.44

2019

Under consideration for publication in J. Functional Programming

Local Algebraic Effect Theories

Žiga Lukšič and Matija Pretnar∗
University of Ljubljana, Faculty of Mathematics and Physics, Slovenia
(*e-mail:* `ziga.luksic@fmf.uni-lj.si, matija.pretnar@fmf.uni-lj.si`)

Abstract

Algebraic effects are computational effects that can be described with a set of basic operations and equations between them. As many interesting effect handlers do not respect these equations, most approaches assume a trivial theory, sacrificing both reasoning power and safety.

We present an alternative approach where the type system tracks equations that are observed in subparts of the program, yielding a sound and flexible logic, and paving a way for practical optimizations and reasoning tools.

Idea

MEGA: 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)

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

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)

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

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

Example of an (effect) equation

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

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

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

$$\mathit{nondetProg} : \texttt{int} \,!\, \{\mathit{choose}\} \;/\; (\textsc{comm})$$

Example ctd.

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

$$yield \;:\; \texttt{int} \to \texttt{unit}$$

and in that library a handler

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

which doesn't care about the order of yielded values, as expressed by

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

Example ctd. ctd.

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

$$\texttt{int} \,!\, \{\mathit{choose}\} \;/\; \emptyset \implies \texttt{unit} \,!\, \{\mathit{yield}\} \;/\; \emptyset$$

but due to the type of our program

$$\mathit{nondetProg} : \texttt{int} \,!\, \{\mathit{choose}\} \;/\; (\textsc{comm})$$

any handler used on *nondetProg* needs to respect $(\textsc{comm})$.

Typing rules

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

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

Most typing rules are largely unchanged.

The only interesting rule is for typing handlers.

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

Handler correctness

The typing part of

$$\Gamma \vdash h : \Sigma \Rightarrow \underline{D} \text{ respects } \mathcal{E}$$

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 respects relation

- (though there are requirements on these logics for denotational semantics to make sense)

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

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

corresponding to the conventional approach of ignoring equations.

We can use different kinds of logics ctd.

Another option is to use equational logic

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

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

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

$$z_i(v)^h[f_j/z_j]_j = f_i\ v$$

$$(\texttt{if}\ v\ \texttt{then}\ T_1\ \texttt{else}\ T_2)^h[f_j/z_j]_j = \texttt{if}\ v\ \texttt{then}\ T_1^h[f_j/z_j]_j\ \texttt{else}\ T_2^h[f_j/z_j]_j$$

$$op(v;\ y.T)^h[f_j/z_j]_j = c_{op}[v/x, (\texttt{fun}\ y \mapsto T^h[f_j/z_j]_j)/k]$$

We can use different kinds of logics ctd. ctd.

To use the equations of the current theory, we include the rule

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

We can use different kinds of logics ctd. ctd. ctd.

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

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

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

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

the type

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

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

Typing *yieldAll*

But luckily

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

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

$$\texttt{int} \,!\, \{\textit{choose}\} \,/\, (\textsc{comm}) \implies \texttt{unit} \,!\, \{\textit{yield}\} \,/\, (\textsc{order})$$

Combining the parts

We can now safely compose

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

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

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

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

Benefits

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