Algebraic programming language theory

Ambrus Kaposi
EFOP-3.6.3-VEKOP-16-2017-00002
PLC Department Workshop
Bugyi
11 January 2019
A program is a

string

sequence of lexical elements

syntax tree

well-scoped syntax tree

well-typed syntax tree

well-typed syntax tree quotiented by semantic equality
Steps

- string
- lexical analysis
- sequence of lexical elements
- parsing
- syntax tree
- scope-checking
- well-scoped syntax tree
- type-checking
- well-typed syntax tree
- well-typed syntax tree quotiented by semantic equality
Errors

- **string**
- **lexical analysis** → **invalid lexical els**
- **sequence of lexical elements**
- **parsing** → **wrong number of params**
- **syntax tree**
- **scope-checking** → **var not in scope**
- **well-scoped syntax tree**
- **type-checking** → **type error**
- **well-typed syntax tree**
- **well-typed syntax tree** "quotiented by semantic equality"
Equalities

\[ \lambda x.x \Rightarrow \lambda y.y \]

\[(\lambda x.x + x)\ 3 = 6\]
Nonsense theorems

string

sequence of lex elements

spaces don’t matter

AST

redundant bracket removal preserves ws removal

well-scoped syntax tree

α-renaming preserves matching brackets

well-typed syntax tree

α-renaming preserves typing

algebraic syntax

β-reduction preserves typing
An algebraic structure

A group has the following components:

\[
\begin{align*}
C & : \text{Set} \\
- \otimes - & : C \to C \to C \\
u & : C \\
-^{-1} & : C \to C \\
\text{ass} & : (a \otimes b) \otimes c = a \otimes (b \otimes c) \\
\text{idl} & : u \otimes a = a \\
\text{idr} & : a \otimes u = a \\
\text{invl} & : a^{-1} \otimes a = u \\
\text{invl} & : a \otimes a^{-1} = u
\end{align*}
\]
An algebraic structure
Groups $A$ and $B$ and a group homomorphism $f$.

$$C_A := \mathbb{Z}$$

$$m \otimes_A n := m + n$$

$$u_A := 0$$

$$m^{-1}_A := -m$$

the laws hold

$$C_B := \mathbb{Z}_3$$

$$m \otimes_B n := m + n \pmod{3}$$

$$u_B := 0$$

$$m^{-1}_B := 3 - m$$

the laws hold

$$f_C : C_A \rightarrow C_B$$

$$f_C m := m \pmod{3}$$

$$f_\otimes : f_C (m \otimes_A n) = f_C m \otimes_B f_C n$$

$$f_u : f_C u_A = u_B$$

$$f_{-1} : f_C (m^{-1}_A) = (f_C m)^{-1}_B$$
Another algebraic structure

An algebra for the expression language has the following components:

<table>
<thead>
<tr>
<th>Ty</th>
<th>: Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tm</td>
<td>: Ty → Set</td>
</tr>
<tr>
<td>Bool</td>
<td>: Ty</td>
</tr>
<tr>
<td>Nat</td>
<td>: Ty</td>
</tr>
<tr>
<td>true</td>
<td>: Tm Bool</td>
</tr>
<tr>
<td>false</td>
<td>: Tm Bool</td>
</tr>
<tr>
<td>if–then–else–</td>
<td>: Tm Bool → Tm A → Tm A → Tm A</td>
</tr>
<tr>
<td>num</td>
<td>: ℤ → Tm Nat</td>
</tr>
<tr>
<td>isZero</td>
<td>: Tm Nat → Tm Bool</td>
</tr>
<tr>
<td>if_β_1</td>
<td>: if true then t else t' = t</td>
</tr>
<tr>
<td>if_β_2</td>
<td>: if false then t else t' = t'</td>
</tr>
<tr>
<td>isZero_β_1</td>
<td>: isZero (num 0) = true</td>
</tr>
<tr>
<td>isZero_β_2</td>
<td>: isZero (num (1 + n)) = false</td>
</tr>
</tbody>
</table>
Syntax and homomorphisms

The syntax for the expression language is an algebra $\text{Ty}_S$, $\text{Tm}_S$, $\text{Bool}_S$, etc, such that there is a homomorphism from it to any other algebra $A$. The homomorphism is called:

- an interpreter if $\text{Ty}_A = \text{Set}$ and $\text{Tm}_A T = T$ in the target algebra
- a compiler if $\text{Ty}_A = \text{Ty}'_S$ and $\text{Tm}_A T' = \text{Tm}'_S T'$ for some other syntax in the target algebra
- an optimisation/program transformation that preserves types and conversion if $\text{Ty}_A = \text{Ty}_S$, and $\text{Tm}_A T = \text{Tm}_S T$ in the target algebra
Old style approach

\[ Ty ::= \text{Bool} \mid \text{Nat} \]
\[ Tm ::= \text{true} \mid \text{false} \mid \text{if } t \text{ then } t' \text{ else } t'' \mid \text{num } n \mid \text{isZero } t \]
\[ (- : -) \subseteq Tm \times Ty \]
\[ (- \mapsto -) \subseteq Tm \times Tm \]
\[ \begin{align*}
\text{true} : \text{Bool} & \\
\text{false} : \text{Bool} & \\
n \in \mathbb{N} & \\
\text{num } n : \text{Nat} & \\
\text{if true then } t \text{ else } t' & \mapsto t \\
t & \mapsto t_1 \\
\text{if } t \text{ then } t' \text{ else } t'' & \mapsto \text{if } t_1 \text{ then } t' \text{ else } t'' \\
\text{isZero } t : \text{Nat} & \\
\text{isZero} (\text{num } 0) & \mapsto \text{true} \\
\text{isZero} (\text{num } (1 + n)) & \mapsto \text{true} \\
\text{if false then } t \text{ else } t' & \mapsto t' \\
\text{if true then } t \text{ else } t' & \mapsto t \\
\text{if false then } t \text{ else } t' & \mapsto t' \\
t & \mapsto t' \\
\text{isZero } t & \mapsto \text{isZero } t' \\
\end{align*} \]
Conversion is the reflexive, transitive, symmetric closure of \(- \mapsto -\).
What can you do on the high level?

We described the syntax of (a subset of) Agda using this technique and wrote a total interpreter for it. We also wrote compilers:

- Closure conversion: towards machine code
- Compile types to setoids: add function extensionality to Agda
- Compile types to reflexive graphs: add parametricity to Agda
- Future: extending a programming language with new principles
- Future: static analysis

You need to respect equalities. You can’t print terms, only normal forms.
Why is it good? (i) less boilerplate. (ii) guides you on the path.
These are very general notions of algebras, not well studied. We started describing them, they are called QIITs (next week POPL, Lisbon). You need a good metatheory (logic) to reason about them, i.e. type theory.