Algebraic programming language theory

Ambrus Kaposi

EFOP-3.6.3-VEKOP-16-2017-00002

PLC Department Workshop
Bugyi
11 January 2019

SZÉCHENYI 2020
Magyarország Kormánya
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

A program is a

string

sequence of lexical elements

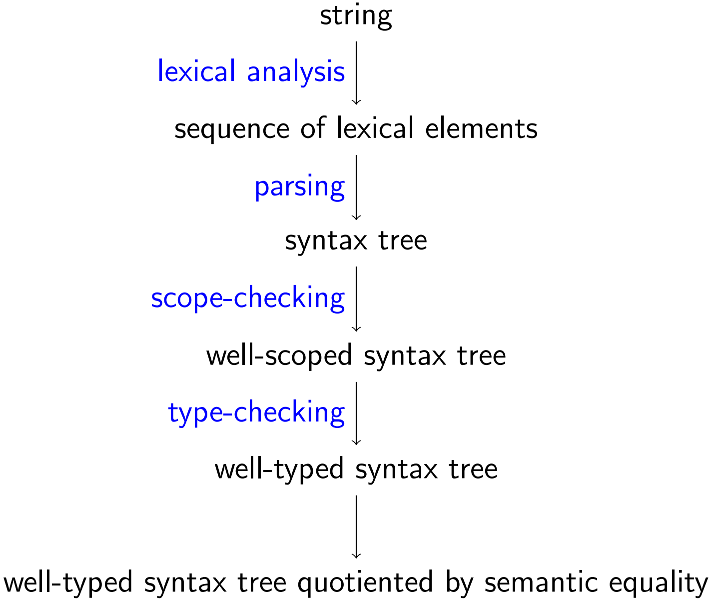
syntax tree

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

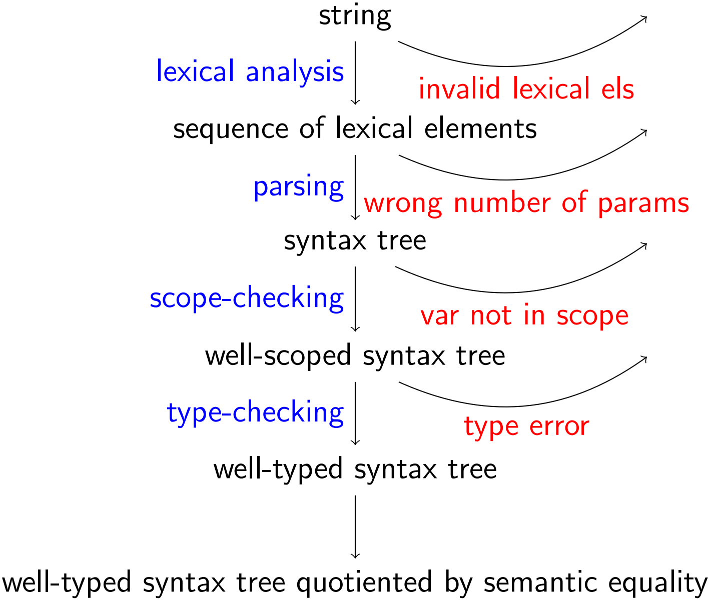
type-checking ↓

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

string

↓ lexical analysis

sequence of lexical elements `"(1+2)+3" = "(1 + 1) + 3"`

↓ parsing

syntax tree $[(,1,+,1,),+,3] = [(,(,1,+,1,),),+,3]$

↓ scope-checking

well-scoped syntax tree $\lambda x.x = \lambda y.y$

↓ type-checking

well-typed syntax tree

↓

algebraic syntax $(\lambda x.x + x)\,3 = 6$

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:

$$
\begin{aligned}
\mathsf{C} \quad &: \mathsf{Set} \\
- \otimes - \quad &: \mathsf{C} \to \mathsf{C} \to \mathsf{C} \\
\mathsf{u} \quad &: \mathsf{C} \\
-^{-1} \quad &: \mathsf{C} \to \mathsf{C} \\
\mathsf{ass} \quad &: (a \otimes b) \otimes c = a \otimes (b \otimes c) \\
\mathsf{idl} \quad &: \mathsf{u} \otimes a = a \\
\mathsf{idr} \quad &: a \otimes \mathsf{u} = a \\
\mathsf{invl} \quad &: a^{-1} \otimes a = \mathsf{u} \\
\mathsf{invl} \quad &: a \otimes a^{-1} = \mathsf{u}
\end{aligned}
$$

An algebraic structure

Groups A and B and a group homomorphism f.

$$\begin{aligned} \mathsf{C}_A &:= \mathbb{Z} \\ m \otimes_A n &:= m + n \\ \mathsf{u}_A &:= 0 \\ m^{-1_A} &:= -m \\ \text{the laws hold} & \end{aligned}$$

$$\begin{aligned} \mathsf{C}_B &:= \mathbb{Z}_3 \\ m \otimes_B n &:= m + n \pmod 3 \\ \mathsf{u}_B &:= 0 \\ m^{-1_B} &:= 3 - m \\ \text{the laws hold} & \end{aligned}$$

$$\begin{aligned} f_{\mathsf{C}} &: \mathsf{C}_A \to \mathsf{C}_B \\ f_{\mathsf{C}}\, m &:= m \pmod 3 \\ f_{\otimes} &: f_{\mathsf{C}}\,(m \otimes_A n) = f_{\mathsf{C}}\, m \otimes_B f_{\mathsf{C}}\, n \\ f_{\mathsf{u}} &: f_{\mathsf{C}}\, \mathsf{u}_A = \mathsf{u}_B \\ f_{-1} &: f_{\mathsf{C}}\,(m^{-1_A}) = (f_{\mathsf{C}}\, m)^{-1_B} \end{aligned}$$

Another algebraic structure

An algebra for the expression language has the following components:

$$
\begin{aligned}
&\mathsf{Ty} && : \mathsf{Set}\\
&\mathsf{Tm} && : \mathsf{Ty} \to \mathsf{Set}\\
&\mathsf{Bool} && : \mathsf{Ty}\\
&\mathsf{Nat} && : \mathsf{Ty}\\
&\mathsf{true} && : \mathsf{Tm}\,\mathsf{Bool}\\
&\mathsf{false} && : \mathsf{Tm}\,\mathsf{Bool}\\
&\mathsf{if{-}then{-}else{-}} && : \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A\\
&\mathsf{num} && : \mathbb{N} \to \mathsf{Tm}\,\mathsf{Nat}\\
&\mathsf{isZero} && : \mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,\mathsf{Bool}\\
&\mathsf{if}\beta_1 && : \mathsf{if}\,\mathsf{true}\,\mathsf{then}\,t\,\mathsf{else}\,t' = t\\
&\mathsf{if}\beta_2 && : \mathsf{if}\,\mathsf{false}\,\mathsf{then}\,t\,\mathsf{else}\,t' = t'\\
&\mathsf{isZero}\beta_1 && : \mathsf{isZero}\,(\mathsf{num}\,0) = \mathsf{true}\\
&\mathsf{isZero}\beta_2 && : \mathsf{isZero}\,(\mathsf{num}\,(1+n)) = \mathsf{false}
\end{aligned}
$$

Syntax and homomorphisms

The syntax for the expression language is an algebra Ty_S, Tm_S, Bool_S, etc, such that there is a homomorphism from it to any other algebra A. The homomorphism is called:

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

Old style approach

$$\begin{aligned} \mathsf{Ty} &::= \mathsf{Bool} \mid \mathsf{Nat} \\ \mathsf{Tm} &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' \mid \mathsf{num}\ n \mid \mathsf{isZero}\ t \end{aligned}$$

$$(- : -) \subseteq \mathsf{Tm} \times \mathsf{Ty} \qquad (- \mapsto -) \subseteq \mathsf{Tm} \times \mathsf{Tm}$$

$$\frac{}{\mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Bool}} \qquad \frac{t : \mathsf{Bool} \qquad t' : A \qquad t'' : A}{\mathsf{if}\ t\ \mathsf{then}\ t'\mathsf{else}\ t'' : A}$$

$$\frac{n \in \mathbb{N}}{\mathsf{num}\ n : \mathsf{Nat}} \qquad \frac{t : \mathsf{Nat}}{\mathsf{isZero}\ t : \mathsf{Bool}}$$

$$\frac{}{\mathsf{if\ true\ then}\ t\ \mathsf{else}\ t' \mapsto t} \qquad \frac{}{\mathsf{if\ false\ then}\ t\ \mathsf{else}\ t' \mapsto t'}$$

$$\frac{t \mapsto t_1}{\mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' \mapsto \mathsf{if}\ t_1\ \mathsf{then}\ t'\ \mathsf{else}\ t''} \qquad \frac{}{\mathsf{isZero}\,(\mathsf{num}\,0) \mapsto \mathsf{true}}$$

$$\frac{}{\mathsf{isZero}\,(\mathsf{num}\,(1+n)) \mapsto \mathsf{true}} \qquad \frac{t \mapsto t'}{\mathsf{isZero}\ t \mapsto \mathsf{isZero}\ t'}$$

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.

Challenges

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.