One syntax to rule them all

D. Ahman & A. Bauer

University of Ljubljana

Notes for the talk delivered at the workshop on Syntax & Semantics of Type Theory, in Stockholm on May 20, 2022.

Overview

1. Variables, expressions & substitutions
2. Higher-rank syntax
3. Syntax as relative monad
4. Other kinds of syntax

Variables, expressions, substitutions

In a formal system, such as dependent type theory, there are several kinds of symbols/variables:

- <u>variables</u>: $x : \mathbb{N} \vdash \lambda y : \mathbb{N}.\ x + y : \mathbb{N} \to \mathbb{N}$
- <u>meta-variables</u>:
$$\frac{\vdash A \qquad x : A \vdash B(x)}{\vdash \Pi(x : A).\, B(x)}$$
- <u>symbols</u> (constructors) $\Pi, \Sigma, 0, \mathrm{suc}, \ldots$

Each of these has an associated notion of substitution:

- substitution: variables $\to$ expressions
$$x \mapsto \mathrm{suc}\, 0 + y$$
- instantiation: meta-variables $\to$ abstracted expressions
$$B \mapsto x.\, \mathrm{Vec}(\mathrm{suc}\, x)$$
abstracted var.
- transformation: symbol $\to$ expression with abstracted meta-variables
$$\exists \mapsto A\ B.\ \|\Sigma(x : A).\, B(x)\|$$

In fact, there are also **renamings** for each of these notions.

Renamings & substitution obey various equational laws, e.g.

$$\rho_*(\sigma_* t) = (\rho_* \sigma)_* (\rho_* t)$$

where ρ = variable renaming
σ = variable substitution
$f_* e$ = stands for "action of f on e"

There are <u>many</u> such laws for each kind of variables, as well as more laws governing interactions between them.

- How do we know we are not missing some laws?
- It is very tedious to prove all the laws three times (we did it)
- It is very tedious to implement several kinds of variables, abstractions & substitution (we did it)

Can we combine these into a single notion?

Higher-rank syntax

WARNING: We are doing raw syntax.
There are no types!
(We shall discuss them later.)

The syntax of a formal system may comprise several **syntactic classes**:

- first-order logic: term, formula
- type theory: term, type
- cubical type theory: dimension, term, type

We therefore suppose a given set

$$\mathrm{Class}$$

of syntactic classes. (Example: $\mathrm{Class} = \{\mathrm{tm}, \mathrm{ty}\}$)

Next, each kind of variables has an associated notion of **arity**:

- <u>variable arity</u>: is just a syntactic class, as variables take no arguments and bind nothing

Example: $A : ty$, $x : tm$, $y : tm$

- <u>metavariable arity</u>:

$$M : (\underbrace{[\alpha_1, \dots, \alpha_n]}_{\text{classes of args}}, \underbrace{cl}_{\text{class of } M})$$

A meta-variable is applied to some expressions, but it does not bind anything.

Example: $M : ([tm, ty], ty)$

$M(7, \text{List Bool})$ a type expression

- <u>symbol arity</u>

$$S : (\underbrace{[\mu_1, \dots, \mu_n]}_{\text{metavariable arities}}, cl)$$

class

A symbol takes arguments and (possibly) binds variables in them.

For each argument we specify

- how it binds variables and what classes they have
- the class of the argument

But this info is the same as a metavariable arity!

<u>Example</u>:

$$\Pi : (\, [\underbrace{([\,], ty)}_{\substack{1^{st}\text{ argument}\\ \text{is a type expression,}\\ \text{no binding}}}, ([tm], ty)], ty\,]$$

2^{nd} argument is a type expression, it binds one term variable.

<u>Exercise</u>: "$\lambda x : A . t(x)$"

$$\lambda : ([\underbrace{([\,], ty)}_{A}, (\underbrace{[\underbrace{tm}_{x}], tm}_{t})], tm)$$

If we make it explicit that variables take no argument by writing

$$x : (\underbrace{[\,]}_{\text{no args}}, cl)$$

then all these arities have the same form:

$$(\underbrace{[\alpha_1, \ldots, \alpha_n]}_{\text{arities of arguments}}, \underbrace{cl}_{\text{class}})$$

So we define a datatype

$$\text{Arity} = \text{List Arity} \times \text{Class}$$

What are its elements?

- rank 0: $([\,], cl)$ variable
- rank 1: $(\underbrace{[\alpha_1, \ldots, \alpha_n]}_{\text{rank } 0}, cl)$ meta-variable
- rank 2: $(\underbrace{[\alpha_1 \ldots \alpha_n]}_{\text{rank } 1}, cl)$ symbol

We can go on!

rank 3: $(\underbrace{[\alpha_1, \ldots, \alpha_n]}_{\text{rank } 2}, cl)$ But what is this?

A schema!

<u>Example:</u>

$$\alpha_\Pi = ([([], ty), ([tm], ty)], ty)$$

$\$: \alpha_\Pi$ a Π-like operator

"Every Π-like operator preserves equivalence."

$$\beta := ([\underbrace{\alpha_\Pi}_{\$}, \underbrace{([], ty)}_{A}, \underbrace{([], ty)}_{A'}, \underbrace{([tm], ty)}_{B}, \underbrace{([tm], ty)}_{B'}], ty),$$

$\mathrm{eqPreserve}(\$, A, A', B, B') :=$

$\Pi(e : A \simeq A').\,(\Pi(x:A).\, Bx \simeq B(ex)) \to \$(A,B) \simeq \$(A', B')$

(NB: Only syntax, no typing info about A, A', B, B'.)

The arity of eqPreserve is β.

- rank 4, rank 5, ... What about these?

Remark:

There is no difference between a variable and an argument-less metavariable.

- Variable x : $([], tm)$
- metavariable M : $([], tm)$

(We will see later how to recover it):

How do we form expressions?

Recall how we usually form expressions:

Given: a signature Σ +
a metavariable context Θ +
a variable context Γ
.. a class cl

we have

$$\mathrm{Expr}(\Sigma, \Theta, \Gamma, cl)$$

the set of all expressions of class cl for signature Σ, metavariables Θ, and variables Γ.

In our setting

$([\Sigma, \Theta, \Gamma], \mathrm{cl})$ is just an arity!

Terminology:

$$(\underbrace{[\alpha_1, \ldots, \alpha_n]}, \mathrm{cl})$$

call this a **syntactic context**

(it specifies variables and their arities but no typing information).

We use letters γ, δ for syntactic contexts.

They form a monoid (they're just lists):

$\mathbb{0} = [\,]$ empty synt. context

$\gamma \oplus \delta$ concatenation.

Henceforth we use de Bruijn indices.

The i-th variable of γ is written as var_i and γ_i is its arity.

Write $(\beta, \mathrm{cl})_i \in \gamma$ if

(β, cl) is the i-th element of γ.

We can now give an inductive definition of expressions. Because all kinds of variables have been unified, there is a single constructor:

$\mathrm{Expr}\ \gamma\ cl$ "expressions of class cl in syntactic context γ"

Construct inductively:

given $(\beta, cl)_i \in \gamma$ and a map t assigning to each $(\delta_j, cl_j)_j \in \beta$ an expression $t_j \in \mathrm{Expr}\,(\gamma \oplus \delta_j)\ cl_j$,

we form an expression

$i \grave{}\, t$ "i-th variable applied to arguments t"

This is more readable in Agda code, which is attached to these notes on math.andrej.com.

Remark:

$$t_j \in \mathrm{Expr}\,(\underbrace{\gamma \oplus \delta_j}_{\text{synt. context is extended by variables } \delta_j \text{ which are then bound in } i\grave{}\,t})\ cl_j$$

j-th argument

Syntax as relative monad

So far we have

- variables & arities
- inductively generated (raw) expressions

Still missing:

- renamings
- substitutions
- equations

We get these by defining a suitable relative monad.

Recall the definition of a relative monad:

$I : \mathcal{C} \to \mathcal{D}$ $\quad T : \mathcal{C} \to \mathcal{D}$ functors

$\eta : I \to T$ unit $\quad$ lifting $f : IA \to TB$ to $f^* : TA \to TB$

Equations: $\quad \eta^* = id$

$$(g^* \circ f) = g^* \circ f^*$$

$$f^* \circ \eta = f$$

Now we specialize:

- Let $\mathbb{C}$ be the category of syntactic contexts and variable renamings

$$\rho : \gamma \to \delta \quad \text{must preserve arities}$$

The relative monad:

$$\mathrm{Var} : \mathbb{C} \to \mathrm{Set}^{\mathbb{C} \times \mathrm{Class}}$$

$$\gamma \mapsto (\delta, cl) \mapsto \{ i \mid (\delta, cl)_i \in \gamma \}$$

those variables that have arity (δ, cl)

$$\mathrm{Expr} : \mathbb{C} \to \mathrm{Set}^{\mathbb{C} \times \mathrm{Class}}$$

$$\gamma \mapsto \delta, cl \mapsto \mathrm{Expr}(\gamma \oplus \delta, cl)$$

functor

free

abstracted vars

the inductively defined set

$$\eta : \mathrm{Var} \to \mathrm{Expr}$$

$$\eta\, \gamma\, (\delta, cl) : \mathrm{Var}\, \gamma\, (\delta, cl) \longrightarrow \mathrm{Expr}(\gamma \oplus \delta, cl)$$

$$i \mapsto i\,`(\lambda j .\, \eta (\gamma \oplus \delta)(\delta_j, cl_j))$$

where $(\delta_j, cl_j)_j \in \delta$

Lifting:

Given $f : \mathrm{Var}\ \gamma \longrightarrow \mathrm{Expr}\ \delta$

define $f^* : \mathrm{Expr}\ \gamma \longrightarrow \mathrm{Expr}\ \delta$ by

$$f\ e := \text{"substitute } f\,i \text{ for } \mathrm{var}_i \text{ in } e\text{"}$$

We do not give the details here, but this notion subsumes substitutions, instantiations and transformations.

What happens:

(the usual syntactic lemmas) "$\simeq$"

(categorical structure of the relative monad)

So far this is just an observation.

To make it more precise we would try to invert the picture:

Given a relative monad with such-and-such additional structure

$\Downarrow$

The monad expresses a notion of syntax with binding

What precisely is the additional structure?

We do not have a final answer, but one feature that plays a role is the fact that $(\mathbb{C}, \mathbb{O}, \oplus)$ is monoidal and so is Var.

Presumably Expr has to be suitably graded over $\mathbb{C}$ ("to do").

Other kinds of syntax

Observation:

Given a relative monad $I : \mathcal{C} \to \mathcal{D}$, $T : \mathcal{C} \to \mathcal{D}$ and any functor $F : \mathcal{B} \to \mathcal{C}$,

$$I \circ F : \mathcal{B} \to \mathcal{D} \qquad T \circ F : \mathcal{B} \to \mathcal{D}$$

is again a relative monad.

Thus, restricted forms of syntactic contexts are accounted for by precomposing.

<u>Example</u>: STLC theory T

- type constructors $\to$ $\times$
- ground types $A_1, \dots, A_n$
- constants $c_1 \dots c_n$
- variables

$\left.\begin{array}{l} \text{type constructors} \\ \text{ground types} \\ \text{constants} \end{array}\right\}$ express as a syntactic context σ_T

STLC syntactic contexts are just numbers

$$n : \mathbb{N}$$

and renamings maps $\mathrm{Fin}\, n \to \mathrm{Fin}\, m$.
To get STLC syntax, use the functor

$$\begin{aligned} \mathbb{N} &\longrightarrow \mathbb{C} \\ n &\longmapsto \sigma_T, \underbrace{([\,], tm), \dots, ([\,], tm)}_{n} \end{aligned}$$

What about equations?

So far we have not accounted for them. This is future work, but it is clear that the construction of the relative monad can accommodate some quotienting.

Wish list:

1) syntactic restrictions such as "at most one occurrence of each variable"

2) incorporate typing information to get "intrinsic" syntax of type theory

To say more about 1) we need to understand better the categorical setup abstractly.

Accomplishing 2) would be quite nice.