

Formalization of ATT

Aidan Ewart

January 31, 2021

In this document I present the syntax, big-step semantics, and typing rules of ATT. Let a and v refer to an infinite set of ‘parameter symbols’. Then, the ATT grammar can be defined by:

s	$::=$	$\mathcal{U}_n, n \in \mathbb{N}$	(Sorts)
\mathbf{R}	$::=$	$a\bar{\mathbf{R}}$	(Reducible Expressions)
\mathbf{N}	$::=$	$a\bar{\mathbf{V}}$	(Neutral Terms)
\mathbf{V}	$::=$	$\lambda x : \mathbf{V}. \mathbf{V} \mid \Pi_{x:\mathbf{V}} \mathbf{V} \mid s \mid v \mid \mathbf{N}$	(Values)
\mathbf{T}	$::=$	$\lambda x : \mathbf{T}. \mathbf{T} \mid \Pi_{x:\mathbf{T}} \mathbf{T} \mid s \mid v \mid \mathbf{TT} \mid \mathbf{T} : \mathbf{T}$	(Expressions)

Note that reducible expressions \mathbf{R} are a subset of values \mathbf{V} , and are simply ‘values without binders’; therefore, we use the same operations such as substitution $\mathbf{V}[x \mapsto \bar{\mathbf{V}}]$ on reducible expressions as we do on values, and indeed treat reducible expressions as values, in an effort to syntactically simplify the rules below.

1 Core ATT

Core ATT is specified by a set of mutually defined rules:

$\Gamma \vdash \text{Valid}$	(Context Validity)
$\Gamma \vdash \mathbf{T} : \mathbf{V}$	(Typing)
$\Gamma \vdash \mathbf{T} \rightsquigarrow^{nf} \mathbf{V}$	(Evaluation)
$\Gamma \vdash \mathbf{V} \subseteq \mathbf{V}$	(Subtyping)
$\Gamma \vdash \mathbf{V} \triangleright \mathbf{V}$	(Conversion)
$\Gamma \vdash \mathbf{V} =_{\alpha\delta\eta\rho} \mathbf{V}$	(Equivalence)

1.1 Contexts

A context (denoted by Γ) in ATT is a set of rules containing definitions, reductions, and type annotations. I use the predicate $\Gamma \vdash \text{Valid}$ to mean a well-formed context, as created with the inference rules (plus the usual rules of weakening, contraction, and so on). Contexts contain parameter annotations $v : \mathbf{V}$, definitions $v : \mathbf{V} := \bar{\mathbf{V}}$, and rho-reductions $[\bar{v} : \bar{\mathbf{V}}]\mathbf{R} \mapsto \mathbf{V}$.

$\frac{}{\cdot \vdash \text{Valid}}$	$\frac{\Gamma \vdash e : T \quad \Gamma \vdash e \rightsquigarrow^{nf} e' \quad v \notin \Gamma \quad \Gamma \vdash \text{Valid}}{\Gamma, v : T := e' \vdash \text{Valid}}$
$\frac{\Gamma \vdash e : T \quad \Gamma \vdash e \rightsquigarrow^{nf} S \quad a \notin \Gamma \quad \Gamma \vdash \text{Valid}}{\Gamma, a : S \vdash \text{Valid}}$	
$\frac{\Gamma, \bar{x} : \bar{X} \vdash e \rightsquigarrow^{nf} r \quad r \in \mathbf{R} \quad \Gamma, \bar{x} : \bar{X} \vdash f : T \quad \Gamma, \bar{x} : \bar{X} \vdash f \rightsquigarrow^{nf} g \quad \Gamma \vdash \text{Valid} \quad \Gamma, \bar{x} : \bar{X} \vdash \text{Valid}}{\Gamma, [\bar{x} : \bar{X}]r \mapsto g \vdash \text{Valid}}$	

For following sections I leave the $\Gamma \vdash \text{Valid}$ constraint implicit.

1.2 Well-Typed Terms

1.2.1 Terms

$$\begin{array}{c}
\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \quad \frac{i > j}{\Gamma \vdash \mathcal{U}_j : \mathcal{U}_i} \quad \frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau : \tau'} \\
\\
\frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma, v : \tau' \vdash e : v}{\Gamma \vdash \lambda v : \tau. e : v} \quad \frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma, v : \tau' \vdash e : \mathcal{U}_j \quad k \geq i, j}{\Gamma \vdash \Pi_{v:\tau} e : \mathcal{U}_k} \\
\\
\frac{\Gamma \vdash f : \Pi_{v:\tau} e \quad \Gamma \vdash x : v \quad \Gamma \vdash x \rightsquigarrow^{nf} x' \quad \Gamma \vdash v \subseteq \tau}{\Gamma \vdash fx : e[v \mapsto x']} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \subseteq v}{\Gamma \vdash e : v}
\end{array}$$

1.2.2 Conversions and Subtyping

As conversion and subtyping is conducted between values (i.e. β -normal forms), there is no β -conversion rule; instead, we have the $\Gamma \vdash \mathbf{T} \rightsquigarrow^{nf} \mathbf{V}$ evaluation rule.

δ -conversion corresponds to the 'unfolding' of definitions in the context:

$$\frac{(x : T := e) \in \Gamma}{\Gamma \vdash x \triangleright_{\delta} e}$$

ρ -conversion corresponds to the application of ρ -reduction rules in the context:

$$\frac{([\overline{x : X}]ae_0 \dots e_m \mapsto e) \in \Gamma \quad \Gamma \vdash \overline{t : X}}{\Gamma \vdash (at_0 \dots t_m)[\overline{x \mapsto t}] \triangleright_{\rho} e[\overline{x \mapsto t}]}$$

And finally the α and η -conversion rules correspond to the normal α -equivalence and η -expansion rules of the λ -calculus:

$$\frac{\Gamma \vdash x =_{\alpha} y}{\Gamma \vdash x \triangleright_{\alpha} y} \quad \frac{\Gamma \vdash f : \Pi_{x:D} R}{\Gamma \vdash f \triangleright_{\eta} \lambda x : D. fx}$$

The convertability relation $=_{\alpha\delta\eta\rho}$ is the transitive, reflexive, closure of $=_{\alpha} \cup \triangleright_{\delta} \cup \triangleright_{\eta} \cup \triangleright_{\rho}$, and relates all convertible terms:

$$\begin{array}{c}
\frac{\Gamma \vdash x \triangleright_{\alpha} y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_{\delta} y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_{\eta} y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_{\rho} y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \quad \frac{\Gamma \vdash x =_{\alpha\delta\eta\rho} y}{\Gamma \vdash y =_{\alpha\delta\eta\rho} x} \\
\\
\frac{\Gamma \vdash x =_{\alpha\delta\eta\rho} y \quad \Gamma \vdash y =_{\alpha\delta\eta\rho} z}{\Gamma \vdash x =_{\alpha\delta\eta\rho} z}
\end{array}$$

If two terms are $\alpha\delta\eta\rho$ -convertable, then they are, for all intents and purposes, equivalent, and therefore subtypes of each other by reflexivity.

$$\begin{array}{c}
\frac{\Gamma \vdash x \triangleright_{\alpha\delta\eta\rho} y}{\Gamma \vdash x \subseteq y} \quad \frac{\Gamma \vdash x \triangleright_{\alpha\delta\eta\rho} y}{\Gamma \vdash y \subseteq x} \quad \frac{i \geq j}{\Gamma \vdash \mathcal{U}_j \subseteq \mathcal{U}_i} \quad \frac{\Gamma \vdash \tau_0 \subseteq \tau_1 \quad \Gamma \vdash v_0 \subseteq v_1}{\Gamma \vdash \Pi_{v:\tau_0} v_0 \subseteq \Pi_{u:\tau_1} v_1} \\
\\
\frac{\Gamma \vdash \tau_0 \subseteq \tau_1 \quad \Gamma \vdash v_0 \subseteq v_1}{\Gamma \vdash \lambda v : \tau_0. v_0 \subseteq \lambda u : \tau_1. v_1} \quad \frac{\Gamma \vdash \tau_0 \subseteq \tau_1 \quad \Gamma \vdash v_0 \subseteq v_1}{\Gamma \vdash \tau_0 v_0 \subseteq \tau_1 v_1}
\end{array}$$

1.2.3 Evaluation

$$\begin{array}{c}
\frac{}{\Gamma \vdash v \rightsquigarrow^{nf} v} \quad \frac{}{\Gamma \vdash \mathcal{U}_i \rightsquigarrow^{nf} \mathcal{U}_i} \quad \frac{\Gamma \vdash f \rightsquigarrow^{nf} \lambda v.e \quad \Gamma \vdash x \rightsquigarrow^{nf} x' \quad \Gamma \vdash e[v \mapsto x'] \rightsquigarrow^{nf} t}{\Gamma \vdash fx \rightsquigarrow^{nf} t} \\
\\
\frac{\Gamma \vdash e \rightsquigarrow^{nf} e'}{\Gamma \vdash e : \tau \rightsquigarrow^{nf} e'} \quad \frac{\Gamma \vdash e \rightsquigarrow^{nf} e'}{\Gamma \vdash \lambda v : \tau.e \rightsquigarrow^{nf} \lambda v.e'} \quad \frac{\Gamma \vdash e \rightsquigarrow^{nf} e' \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau'}{\Gamma \vdash \Pi_{v:\tau} e \rightsquigarrow^{nf} \Pi_{v:\tau'} e'}
\end{array}$$

2 The Vernacular Language

To interact with the ATTh system, you use the *vernacular* interface, a command-oriented ‘programming language’ supporting a number of operations.

One can use the `Definition` command to add a δ -expansion for a name to the context:

Definition `id := fun (X: Type) (x: X) \Rightarrow x.`

The `Axiom` command adds top-level parameters to the context:

Axiom `false : forall (X: Type), x.`

The `Reduction` command adds ρ -reductions to the context (note that these are entirely unrestricted, and therefore break many metatheoretic properties when abused):

Axiom `Mu : forall (A: Type), (A \rightarrow A) \rightarrow A.`

Reduction `(A: Type) (f: A \rightarrow A) (Mu A f) := f (Mu A f).`

The `Inductive` command is essentially a packaging of `Axiom` and `Reduction` commands for strictly-positive inductive types:

Inductive `nat : Type :=`
`| S : nat \rightarrow nat`
`| Z : nat.`

has the same effect as

Axiom `nat : Type.`

Axiom `Z : nat.`

Axiom `S : nat \rightarrow nat.`

Axiom `rec_nat : forall (P: nat \rightarrow Type),`
`(P Z) \rightarrow (forall (n: nat), (P n) \rightarrow P (S n)) \rightarrow forall (n: nat), P n.`

Reduction `(P: nat \rightarrow Type) (z: P Z) (sn: forall (n: nat), (P n) \rightarrow P (S n))`
`(rec_nat P z sn Z) := z.`

Reduction
`(P: nat \rightarrow Type)`
`(z: P Z)`
`(sn: forall (n: nat), (P n) \rightarrow P (S n))`
`(n: nat)`
`(rec_nat P z sn (S n)) := sn n (rec_nat P z sn n).`

Naturals in the source are elaborated to their inductive form; i.e. 3 would become `S (S (S Z))`.

The `Print` command can be used to display the definition of multiple names in the context:

Print `id false.`

Print `All.`

or to print the graph representing the (algebraic) universe heirarchy:

Print `Universes.`

The `Check` command comes in two forms. One can either infer the type of an expression:

Check `rec_nat`.

or check to see if adding arbitrary universe constraints (specified by their universe identifier) would create a non-well-founded universe hierarchy:

Check Constraints `1 = 2, 3 <= 2`.

The `Transparent` and `Opaque` commands can be used to specify whether a name should be aggressively δ and ρ -reduced, or not, respectively. Names are `Opaque` by default:

Transparent `rec_nat`.

Definition `add := fun (x y: nat) => rec_nat (fun _ => nat) y (fun _ p => S p) x`.

Transparent `add`.

Compute `add 4 5`.

(Results in 9 *)*

Opaque `add`.

Compute `add 4 5`.

(Results in add 4 5. *)*

The `Eval` and `Compute` commands can be used to perform computation with a given reduction strategy, or the default one:

Compute `add 4 5`.

(Results in add 4 5. *)*

Eval `(unfolding add) in add 4 5`.

Eval `(match 9) in add 4 5`.

Eval `ehnf in add 4 5`.

Eval `esnf in add 4 5`.

Eval `ehnf (unfolding add) esnf (match 9) in add 4 5`.

(Results in 9 *)*

The available reduction strategies are:

- `unfolding ...` which acts as a local variant of `Transparent`
- `match exp` which tries to match the term to `exp`, resulting in `exp`
- `ehnf` ('Expanded Head Normal Form') aggressively expands the topmost term of the expression
- `esnf` ('Expanded Spine Normal Form') aggressively expands the topmost term of the expression, recursing for the bodies of λ and Π abstractions.

These reduction strategies are applied in left-to-right order, except for `unfolding ...` strategies, which are applied throughout execution.