

Formalization of ATT

Aidan Ewart

January 29, 2021

In this document I present the syntax, big-step semantics, and typing rules of ATT.

$$\begin{aligned} s &::= \mathcal{U}_n, n \in \mathbb{N} \\ \mathbf{T} &::= \lambda v : \mathbf{T}. \mathbf{T} \mid \Pi_{v:\mathbf{T}} \mathbf{T} \mid s \mid \mathbf{T}\mathbf{T} \mid \mathbf{T} : \mathbf{T} \mid v \end{aligned}$$

1 Core ATT

1.1 Environments

An environment (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 environment, as created with the inference rules (plus the usual rules of weakening, contraction, and so on):

$$\begin{array}{c} \frac{}{\cdot \vdash \text{Valid}} \quad \frac{\Gamma \vdash e : T \quad x \notin \Gamma \quad \Gamma \vdash \text{Valid}}{\Gamma, x : T := e \vdash \text{Valid}} \quad \frac{\Gamma \vdash T \quad a \notin \Gamma \quad \Gamma \vdash \text{Valid}}{\Gamma, a : T \vdash \text{Valid}} \\[10pt] \frac{(a : S) \in \Gamma \quad \Gamma, \overline{x : X} \vdash ae_0 \dots e_m : T \quad \Gamma, \overline{x : X} \vdash e : T \quad \Gamma \vdash \text{Valid} \quad \Gamma, \overline{x : X} \vdash \text{Valid}}{\Gamma, [\overline{x : X}]ae_0 \dots e_m \mapsto e \vdash \text{Valid}} \end{array}$$

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

1.2 Well-Typed Terms

1.2.1 Terms

$$\begin{array}{c} \frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \quad \frac{i > j}{\Gamma \vdash \mathcal{U}_j : \mathcal{U}_i} \quad \frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau : \tau'} \\[10pt] \frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma, v : \tau' \vdash e : v}{\Gamma \vdash \lambda v : \tau. e : v} \quad \frac{\Gamma \vdash \tau : \mathcal{U}_i \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \quad \Gamma, v : \tau' \vdash e : \mathcal{U}_j \quad k \geq i, j}{\Gamma \vdash \Pi_{v:\tau} e : \mathcal{U}_k} \\[10pt] \frac{\Gamma \vdash f : \Pi_{v:\tau} e \quad \Gamma \vdash x : v \quad \Gamma \vdash x \rightsquigarrow^{nf} x' \quad \Gamma \vdash v \subseteq \tau}{\Gamma \vdash fx : e[v \mapsto x']} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \subseteq v}{\Gamma \vdash e : v} \end{array}$$

1.2.2 Conversions and Subtyping

δ -conversion corresponds to the ‘unfolding’ of definitions in the environment:

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

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

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

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

$$\frac{\Gamma \vdash x \rightsquigarrow^{nf} y}{\Gamma \vdash x \triangleright_\beta y} \quad \frac{\Gamma \vdash f : \Pi_{x:D} R}{\Gamma \vdash f \triangleright_\eta \lambda x : D. fx}$$

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

$$\frac{\Gamma \vdash x =_\alpha y}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_\beta y}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_\delta y}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_\eta y}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y} \quad \frac{\Gamma \vdash x \triangleright_\rho y}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y} \quad \frac{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y}{\Gamma \vdash y =_{\alpha\beta\delta\eta\rho} x}$$

$$\frac{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} y \quad \Gamma \vdash y =_{\alpha\beta\delta\eta\rho} z}{\Gamma \vdash x =_{\alpha\beta\delta\eta\rho} z}$$

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

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

$$\frac{\Gamma \vdash \tau_0 \subseteq \tau_1 \quad \Gamma \vdash v_0 \subseteq v_1}{\Gamma \vdash \lambda v : \tau_0. v_0 \subseteq \lambda u : \tau_1. v_1} \quad \frac{\Gamma \vdash \tau_0 \subseteq \tau_1 \quad \Gamma \vdash v_0 \subseteq v_1}{\Gamma \vdash \tau_0 v_0 \subseteq \tau_1 v_1}$$

1.2.3 Evaluation

$$\frac{}{\Gamma \vdash v \rightsquigarrow^{nf} v} \quad \frac{}{\Gamma \vdash \mathcal{U}_i \rightsquigarrow^{nf} \mathcal{U}_i} \quad \frac{\Gamma \vdash f \rightsquigarrow^{nf} \lambda v. e \quad \Gamma \vdash x \rightsquigarrow^{nf} x' \quad \Gamma \vdash e[v \mapsto x'] \rightsquigarrow^{nf} t}{\Gamma \vdash fx \rightsquigarrow^{nf} t}$$

$$\frac{\Gamma \vdash e \rightsquigarrow^{nf} e'}{\Gamma \vdash e : \tau \rightsquigarrow^{nf} e'} \quad \frac{\Gamma \vdash e \rightsquigarrow^{nf} e'}{\Gamma \vdash \lambda v : \tau. e \rightsquigarrow^{nf} \lambda v. e'} \quad \frac{\Gamma \vdash e \rightsquigarrow^{nf} e' \quad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau'}{\Gamma \vdash \Pi_{v:\tau} e \rightsquigarrow^{nf} \Pi_{v:\tau'} e'}$$

2 The Vernacular Language

To interact with the ATT 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 environment:

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

The **Axiom** command adds top-level parameters to the environment:

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

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

Axiom `Mu : forall (A: Type), (A -> A) -> A.`

Reduction `(A: Type) (f: A -> 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 → nat
| Z : nat.
```

has the same effect as

```
Axiom nat : Type.
Axiom Z : nat.
Axiom S : nat → nat.
```

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

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

```
Reduction
(P: nat → Type)
(z: P Z)
(sn: forall (n: nat), (P n) → 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 environment:

```
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 heirarchy:

```
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 through execution.