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:

$$
\begin{array}{rcll}
s & ::= & \mathcal{U}_n, n \in \mathbb{N} & \text{(Sorts)} \\
\mathbf{N} & ::= & a\bar{\mathbf{V}} & \text{(Neutral Terms)} \\
\mathbf{V} & ::= & \lambda x : \mathbf{V}.\mathbf{V} \;\mid\; \Pi_{x:\mathbf{V}}\mathbf{V} \;\mid\; s \;\mid\; v \;\mid\; \mathbf{N} & \text{(Values)} \\
\mathbf{T} & ::= & \lambda x : \mathbf{T}.\mathbf{T} \;\mid\; \Pi_{x:\mathbf{T}}\mathbf{T} \;\mid\; s \;\mid\; v \;\mid\; \mathbf{TT} \;\mid\; \mathbf{T} : \mathbf{T} & \text{(Expressions)}
\end{array}
$$

1 Core ATT

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

$$
\begin{array}{ll}
\Gamma \vdash \mathbf{T} : \mathbf{V} & \text{(Typing)} \\
\Gamma \vdash \mathbf{T} \rightsquigarrow^{nf} \mathbf{V} & \text{(Evaluation)} \\
\Gamma \vdash \mathbf{V} \subseteq \mathbf{V} & \text{(Subtyping)} \\
\Gamma \vdash \mathbf{V} \triangleright \mathbf{V} & \text{(Conversion)} \\
\Gamma \vdash \mathbf{V} =_{\alpha\delta\eta\rho} \mathbf{V} & \text{(Equivalence)}
\end{array}
$$

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):

$$
\frac{}{\cdot \vdash \text{Valid}} \qquad
\frac{\Gamma \vdash e : T \qquad v \notin \Gamma \qquad \Gamma \vdash \text{Valid}}{\Gamma, v : T := e \vdash \text{Valid}} \qquad
\frac{\Gamma \vdash e : T \qquad \Gamma \vdash e \rightsquigarrow^{nf} S \qquad a \notin \Gamma \qquad \Gamma \vdash \text{Valid}}{\Gamma, a : S \vdash \text{Valid}}
$$

$$
\frac{\Gamma, \overline{x : X} \vdash e \rightsquigarrow^{nf} av_0 \ldots v_m \qquad \begin{array}{c}\Gamma, \overline{x : X} \vdash e : T \\ \Gamma, \overline{x : X} \vdash f : T\end{array} \qquad \Gamma, \overline{x : X} \vdash f \rightsquigarrow^{nf} g \qquad \Gamma \vdash \text{Valid} \qquad \Gamma, \overline{x : X} \vdash \text{Valid}}{\Gamma, [\overline{x : X}]av_0 \ldots v_m \mapsto g \vdash \text{Valid}}
$$

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

1.2 Well-Typed Terms

1.2.1 Terms

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{i > j}{\Gamma \vdash \mathcal{U}_j : \mathcal{U}_i} \qquad \frac{\Gamma \vdash \tau : \mathcal{U}_i \qquad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \qquad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau : \tau'}$$

$$\frac{\Gamma \vdash \tau : \mathcal{U}_i \qquad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \qquad \Gamma, v : \tau' \vdash e : v}{\Gamma \vdash \lambda v : \tau . e : v} \qquad \frac{\Gamma \vdash \tau : \mathcal{U}_i \qquad \Gamma \vdash \tau \rightsquigarrow^{nf} \tau' \qquad \Gamma, v : \tau' \vdash e : \mathcal{U}_j \qquad k \geq i, j}{\Gamma \vdash \Pi_{v:\tau} e : \mathcal{U}_k}$$

$$\frac{\Gamma \vdash f : \Pi_{v:\tau} e \qquad \Gamma \vdash x : v \qquad \Gamma \vdash x \rightsquigarrow^{nf} x' \qquad \Gamma \vdash v \subseteq \tau}{\Gamma \vdash f x : e[v \mapsto x']} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \subseteq v}{\Gamma \vdash e : v}$$

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 environment:

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

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

$$\frac{([\overline{x : X}] a e_0 \ldots e_m \mapsto e) \in \Gamma \qquad \Gamma \vdash \overline{t : X}}{\Gamma \vdash (a t_0 \ldots t_m)[\overline{x \mapsto t}] \rhd_\rho e[\overline{x \mapsto t}]}$$

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

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

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

$$\frac{\Gamma \vdash x \rhd_\alpha y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \frac{\Gamma \vdash x \rhd_\delta y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \qquad \frac{\Gamma \vdash x \rhd_\eta y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \qquad \frac{\Gamma \vdash x \rhd_\rho y}{\Gamma \vdash x =_{\alpha\delta\eta\rho} y} \qquad \frac{\Gamma \vdash x =_{\alpha\delta\eta\rho} y}{\Gamma \vdash y =_{\alpha\delta\eta\rho} x}$$

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

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.

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

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

1.2.3 Evaluation

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

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

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 agressively 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') agressively expands the topmost term of the expression
- `esnf` ('Expanded Spine Normal Form') agressively 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 throught execution.