1 Hypothetico-General Judgments

Last week, we looked briefly at the semantic judgment of computational dependent type theory with a one element context. This can be extended hypothetical-generally with the idea functionality:

Consider a h-g judgment:

\[ x_1 : A_1, ..., x_n : A_n \geq A \doteq A' \text{ type} \]

and the corresponding judgment on terms:

\[ x_1 : A_1, ..., x_n : A_n \geq M \doteq M' \in A \]

They are defined mutually recursively by induction on \( n \):

\[
\begin{align*}
\text{n = 0 :} & \quad \cdot \geq A \doteq A' \text{ type iff } A \doteq A' \text{ type and} \\
\text{n = k + 1 :} & \quad x_1 : A_1, ..., x_{k+1} : A_{k+1} \geq A \doteq A' \text{ type iff} \\
& \quad \begin{cases} 
M_1 \doteq M'_1 \in A_1, \ M_2 \doteq M'_2 \in [M_1/x_1]A_2, ..., \ M_{k+1} \doteq M'_{k+1} \in [M_i/x_i]^{k+1}A_{k+1} \quad \text{implies} \\
[M_i/x_i]^{k+1}A \doteq [M'_i/x_i]^{k+1}A' \text{ type and} \\
x_1 : A_1, ..., x_{k+1} : A_{k+1} \geq M \doteq M' \in A \\
M_1 \doteq M'_1 \in A_1, \ M_2 \doteq M'_2 \in [M_1/x_1]A_2, ..., \ M_{k+1} \doteq M'_{k+1} \in [M_i/x_i]^{k+1}A_{k+1} \quad \text{implies} \\
[M_i/x_i]^{k+1}M \doteq [M'_i/x_i]^{k+1}M' \in [M_i/x_i]^{k+1}A 
\end{cases}
\end{align*}
\]

Note we define the unary versions of the h-g judgments as reflexive cases (e.g. \( \Gamma \geq A \text{ type} \) iff \( \Gamma \geq A \text{ type} \)). We can check that the h-g judgments are symmetric and transitive, and by the “p.e.r” trick, we can get reflexivity as well. In addition, the h-g judgments also have the structural properties of entailment:

1. \( \Gamma, x : A, \Gamma' \geq x \in A \)
2. if \( \Gamma \geq J \) and \( \Gamma \geq A \text{ type} \) then \( \Gamma, x : A \geq J \)
3. if \( \Gamma, x : A, \Gamma' \geq J \text{ type} \) and \( \Gamma \geq M \doteq M' \in A \) then \( \Gamma[M/x] \Gamma' \geq [M/x]J \text{ type} \)

Where \( J \) any antecedent of a semantic h-g judgment. With these properties, we can embed a logic in this framework:

1. inhabitantion: \( A \text{ true} \) iff there is a term \( M \) s.t. \( M \in A \)
2. equality of proofs: \( \text{Eq}_A(M, N) \)
1.1 Function Equality

Semantic function equality is extensional, since functionality is built into the definition of the semantic judgments:

\[ \lambda x:A. M = \lambda x:A. M' \in (x : A) \rightarrow B \text{ iff } x : A \gg M = M' \in B \]

where the context is a mapping of variables to closed values.

1.2 Equality Type

The type \( \text{Eq}_A(M, N) \) internalizes semantic judgmental equality:

\[ \text{Eq}_A(M, N) \equiv \text{Eq}_A(M', N') \text{ type iff } A = A' \text{ type, } M = M' \in A, \text{ and } N = N' \in A \]

refl\(_A(M) \in \text{Eq}_A(M, N) \text{ iff } M = N \in A \]

2 FTT and CTT

As before, we relate FTT and CTT with theorems that state the sufficiency of the former for the latter. First, we extend the erasure operation to identity types, introduction, and elimination:

\[ |\text{id}_A(M, N)| \triangleq \text{Eq}|_A(|M|, |N|) \]

\[ |\text{refl}_A(M)| \triangleq \text{refl} \]

\[ |J_{a,b,c,C}(a.Q)| \triangleq |\text{id}|_A(M, N/(a)) \quad \text{(where } P : \text{id}_A(M, N)) \]

1. If \( \Gamma \vdash M : A \) then \( |\Gamma| \gg |M| \in |A| \)
2. If \( \Gamma \vdash M = M' : A \) then \( |\Gamma| \gg |M| = |M'| \in |A| \)
3. If \( \Gamma \vdash \text{type} \) then \( |\Gamma| \gg |\text{type}| \)
4. If \( \Gamma \vdash A = A' \text{ type} \) then \( |\Gamma| \gg |A| = |A'| \text{ type} \)

The proof proceeds by nested induction on the 4 FTT type derivations. We show the crucial case of typing for \( J \).

\[ (E) \]

\[ \Gamma, a : A, b : A, p : \text{id}_A(a, b) \vdash C \text{ type} \]

\[ \Gamma \vdash J_{a,b,p,C}(a.Q)(P) : [M, N, P/a, b, p|C] \quad \Gamma \vdash \text{id}_A(M, N) \]

By IH on the third premise, we have \( |\Gamma| \gg |P| \in |\text{id}_A(M, N)| \). Unfolding the definitions of erasure and Eq, we have \( |\Gamma| \gg |M| = |N| \in |A| \). By IH on the second premise, we have \( |\Gamma, a : A| \gg a.Q \in |a, a, \text{refl}_A(a)/a, b, p|C| \). Unfolding the erasure, we get \( |\Gamma, a : A| \gg a.Q \in |a, a, \text{refl}_A(a)/b, p|C| \). By the substitution
property, we get $|\Gamma| \gg |[M]/a|Q| \in |[M], [M], \text{refl}/a,b,p|C|$. Now by IH on the first premise, we have $|\Gamma, a : A, b : A, p : \text{Id}_A(a,b)| \gg |C| \equiv |C|$ type, or that $|\Gamma, a : A, b : A| \gg |\text{Eq}_A([a], [b])| \gg |C| \equiv |C|$ type. By functionality on $|\Gamma| \gg |[M] \equiv |[M]| \equiv |N| \in |A|$, and $|\Gamma| \gg |P| \equiv \text{refl} \in \text{Eq}_A([M], [N])$, we have $|\Gamma| \gg |[M], [M], \text{refl}/a,b,p,C| \equiv |[M], [N], [P]/a,b,p,C|$. Thus we get $|\Gamma| \gg |[M]/a|Q| \equiv |[M], [N], [P]/a,b,p|C|$, which by definition is $|\Gamma| \gg |J_{a,b,c,C}(a.Q)(P)| \in |[M], N, P/a,b,p,C|$.

\[\Box\]

3 An Explicit Construction of Computational Dependent Types

In this section, we give a rigorous set-theoretic construction of computational dependent types (CDTs), as introduced by Allen [1987]. In this construction, we define the system of computational dependent types as the least fixed point of a monotone operator over a lattice of candidate type-systems. The existence of such a fixed point is guaranteed by the Knaster-Tarski theorem.

3.1 Background: Knaster-Tarski Fixed Point Theorem

A complete lattice $(L, \leq)$ consists in a set $L$ along with a pre-order $\leq$ such that all subsets of $L$ admit a greatest lower bound (aka meet) and a least upper bound (aka join). More formally, for every subset $X \subseteq L$, there are elements $\bigwedge X$ and $\bigvee X$ in $L$ such that:

$$\forall x \in X, \bigwedge X \leq x \quad \text{and} \quad \forall y, (\forall x \in X, y \leq x) \implies y \leq \bigwedge X$$

$$\forall x \in X, x \leq \bigvee X \quad \text{and} \quad \forall y, (\forall x \in X, x \leq y) \implies \bigvee X \leq y.$$ 

**Remark.** If $L$ is infinite, this is a stronger requirement than just having binary meet and join operators $\wedge$ and $\vee$ on $L$ along with smallest and largest elements $\top$ and $\bot$ (in which case $(L, \leq)$ is called a plain lattice).

A function $f : L \to L$ is said to be monotone if it preserves $\leq$ in the sense that

$$\forall x, y \in L, x \leq y \implies f(x) \leq f(y).$$

We can now state Knaster-Tarski theorem.

**Theorem.** Let $(L, \leq)$ a complete lattice and $f : L \to L$ a monotone function. Then, $f$ admits a complete lattice of fixed points. In particular, it admits a least fixed point $\mu f$ which is given by the meet of its pre-fixed points:

$$\mu f = \bigwedge \{x \in L \mid f(x) \leq x \}.$$

In practice, $(L, \leq)$ is often a powerset lattice: $(L, \leq) = (\mathcal{P}(E), \subseteq)$ where $E$ is a set and $\subseteq$ refers to set inclusion. In this case, the meet operation corresponds to set intersection and the join operation corresponds to set union. Therefore, if $f : \mathcal{P}(E) \to \mathcal{P}(E)$ is monotone, $\mu f$ is the intersection of every subset of $E$ that is closed under $f$:

$$\mu f = \bigcap \{X \subseteq E \mid f(X) \subseteq X\}.$$

\[\text{A pre-order is a reflexive and transitive relation.}\]
3.2 Construction of Computational Dependent Types

3.2.1 Candidate type systems

A candidate type system is a ternary relation between values, values, and binary relations over values. Intuitively, if \( \tau(A, B, \phi) \) then \( A \) and \( B \) are closed values corresponding to equal types and \( \phi \) is a binary relation over closed values that corresponds to equality of values within type \( A \) (or \( B \)). For example, a type system for booleans would be

\[ \tau_{\text{bool}} = \{(\text{bool}, \text{bool}, \beta)\}, \quad \beta = \{(\text{true}, \text{true}), (\text{false}, \text{false})\}. \]

3.2.2 Real type systems

A real type system is defined as a candidate type system for which the following properties hold (every free variable is universally quantified):

1. **Functionality:** \( \tau(A, B, \phi) \) and \( \tau(A, B, \phi') \) imply \( \phi = \phi' \)
2. **P.E.R valued:** if \( \tau(A, B, \phi) \), then \( \phi \) is a partial equivalence relation \( \footnote{A partial equivalence relation is a symmetric and transitive relation.} \)
3. **Symmetry:** \( \tau(A, B, \phi) \) implies \( \tau(B, A, \phi) \)
4. **Transitivity:** \( \tau(A, B, \phi) \) and \( \tau(B, C, \phi') \) imply \( \tau(A, C, \phi) \).

**Remark.** Another way to formulate transitivity would be as follows:

\( \tau(A, B, \phi) \) and \( \tau(B, C, \phi') \) imply \( \tau(A, C, \phi) \) and \( \phi = \phi' \).

This is equivalent to our first formulation. Indeed, suppose \( \tau(A, B, \phi) \) and \( \tau(B, C, \phi') \). Using symmetry and transitivity, we have \( \tau(B, B, \phi) \) and \( \tau(B, B, \phi') \) (p.e.r trick). Using functionality, we get \( \phi = \phi' \).

The set of all candidate type systems forms a complete lattice for set inclusion. This is not true of the set of all real type systems, which is why we need the notion of candidate type-system (see section 3.2.4).

3.2.3 Truth judgments relative to a real type system

We define the following truth judgments relative to a real type system \( \tau \):

- \( \tau \vdash A \equiv B \) type if and only if \( \tau^\phi(A, B, \phi) \) for some \( \phi \)
- \( \tau \vdash M \equiv N \in A \) if and only if \( \tau^\phi(A, B, \phi) \) and \( \phi^\phi(M, N) \) for some \( \phi \)

where the lifted relations \( \tau^\phi \) and \( \phi^\phi \) are defined as follows:

- \( \phi^\phi(M, N) \) iff there exists values \( V, W \) such that \( M \downarrow V, N \downarrow W \) and \( \phi(V, W) \)
- \( \tau^\phi(A, B, \phi) \) iff there exists values \( V, W \) such that \( A \downarrow V, B \downarrow W \) and \( \tau(V, W, \phi) \).
3.2.4 Defining \( \tau_0 \) as a least fixed point

We can define the type system \( \tau_0 \) of computational dependent types as the least fixed-point of a monotone operator

\[
\Phi(\tau) \triangleq \text{BOOL} \cup \text{NAT} \cup \text{EQ}(\tau) \cup \text{PI}(\tau) \cup \text{SIGMA}(\tau).
\]

After we define \text{BOOL}, \text{NAT}, \text{EQ}, \text{PI} and \text{TAU}, we must prove that the resulting operator \( \Phi \) is monotone and that \( \tau_0 \triangleq \mu\Phi \) is a real type-system.

**Booleans** As seen earlier,

\[
\text{BOOL} = \{(\text{bool}, \text{bool}, \beta) \mid \beta = \{(\text{true}, \text{true}), (\text{false}, \text{false})\}.
\]

**Natural numbers** We define \text{NAT} = \{(\text{nat}, \text{nat}, \mu N) \mid \mu N \text{ is the least fixed-point of the following monotone operator over the complete lattice of relations over values:}

\[
N(\alpha) = \{(z, z) \cup \{(s(n), s(m)) \mid \alpha(m, n)\}.
\]

**Equality** We have \( \text{EQ}(\tau)(A_0, A'_0, \epsilon) \) if and only if

- \( A_0 = \text{Eq}_A(M, N) \) and \( A'_0 = \text{Eq}_A(M', N') \) for some \( A, A', M, M', N, N' \)
- \( \tau^\delta(A, A', \phi), \phi^\delta(M, M') \) and \( \phi^\delta(N, N') \) for some \( \phi \)
- \( \epsilon = \{(\text{refl, refl}) \mid \phi^\delta(M, N) \) and \( \{\} \) otherwise.

**Dependent function** We have \( \text{PI}(\tau)(A_0, A'_0, \rho) \) if and only if

- \( A_0 = (x : A \rightarrow B) \) and \( A'_0 = (x : A' \rightarrow B') \) for some \( A, A', B, B' \)
- \( \tau(A, A', \alpha) \) for some \( \alpha \)
- there exists a function \( \psi \) mapping pairs of terms to relations over values such that

\[
\alpha^\delta(M, M') \text{ implies } \tau^\delta([M/x]B, [M'/x]B', \psi(M, M')) \text{ for all terms } M, M'
\]
- \( \rho(M, M') \) if and only if
  - \( M = \lambda x. N \) for some \( N \)
  - \( M' = \lambda x. N' \) for some \( N' \)
  - \( \alpha^\delta(P, P') \) implies \( \psi(P, P')^\delta([P/x]N, [P'/x]N') \) for all terms \( P, P' \).

**Dependent products** We have \( \text{SIGMA}(\tau)(A_0, A'_0, \sigma) \) if and only if

- \( A_0 = (x : A \times B) \) and \( A'_0 = (x : A' \times B') \) for some \( A, A', B, B' \)
- \( \tau(A, A', \alpha) \) for some \( \alpha \)
- there exists a function \( \psi \) mapping pairs of terms to relations over values such that

\[
\alpha^\delta(M, M') \text{ implies } \tau^\delta([M/x]B, [M'/x]B', \psi(M, M')) \text{ for all terms } M, M'
\]
- \( \sigma(P, P') \) if and only if
  - \( P = \text{pair}_{A, x, B}(M, N) \) for some \( M, N \)
  - \( P' = \text{pair}_{A', x, B}(M', N') \) for some \( M', N' \)
  - \( \alpha^\delta(M, M') \) and \( \psi(M, M')^\delta(N, N') \).
3.2.5 Proving the typing rules for \( \tau_0 \)

Finally, we must show that all the typical typing rules are true with respect to \( \tau_0 \). For example, we must show that if \( \tau_0 \models M : (x : A \to B) \) and \( \tau_0 \models M : A \) then \( \tau_0 \models MN : [N/x]B \).

3.3 Discussion

In order to show that \( \Phi \) (as defined in 3.2.4) is monotone, it is sufficient to show that \( \text{eq}, \pi \) and \( \sigma \) are monotone. A full proof would be tedious but the core idea that makes it work is that \( \tau \) only appears in positive positions in the definitions of \( \text{eq}(\tau) \), \( \pi(\tau) \) and \( \sigma(\tau) \). In the case of \( \pi \) for example, this reflects the fact that the terms \( A \) and \( [M/x]B \) for all \( M \) can be considered as defined prior to \( x : A \to B \).

This would not hold for the type constructor \( \forall X.A \), which cannot regard \( [B/X]A \) as prior to \( \forall X.A \) for all \( B \), as \( B \) could be \( \forall X.A \). More concretely, here is an attempt of adding universal quantification over types in our language by extending \( \Phi \) with a \text{forall} operator:

**Type quantification** We have \( \text{forall}(\tau)(A_0, A'_0, F) \) if and only if

- \( A_0 = \forall X.A \) and \( A'_0 = \forall X.A' \)
- There is a mapping \( \psi \) from pairs of terms to relations over values such that
  \[
  \tau^\upharpoonright(T, T', -) \text{ implies } \tau^\upharpoonright([T/x]A, [T'/x']A', \psi(T, T')) \text{ for all terms } T, T'
  \]
- \( F(M, M') \) if and only if
  - \( M = \Lambda X.N \) for some term \( N \)
  - \( M' = \Lambda X.N' \) for some term \( N' \)
  - \( \tau^\upharpoonright(T, T', -) \text{ implies } \psi(T, T')^\upharpoonright([T/x]N, [T'/x']N') \text{ for all terms } T, T' \)

Such an operator is not monotone though, as can be seen by the negative occurrences of \( \tau \) in the definition of \( \text{forall}(\tau) \) (highlighted in red).

Similarly, in the definition of \( \text{nat} \), it is crucial that the \( N \) relation is monotone so that \( \mu N \) is well-defined. This monotonicity constraint prevents us from giving a fixed point definition for a type like \( D \equiv D \to D \). Indeed, it is tempting to define a type-system for \( D \) as follows:

\[
\tau_D \equiv \{(D, D, \mu X)\} \text{ where } \mu X \text{ is the least-fixed point of } X:
\]

\[
X(\alpha) \equiv \{(\text{Fun}(f), \text{Fun}(g)) \mid \forall V, W. \alpha(V, W) \implies \alpha^\upharpoonright(f(V), f(W))\}
\]

Such an operator is not monotone though, \( \alpha \) appearing in a negative position in the definition of \( X(\alpha) \). Therefore, \( \mu X \) does not necessarily exist. This makes sense because were a type like \( D \) allowed, we would have non-termination, which we do not want at this stage.

References