Internalization of extensional equality

Andrew Polonsky

March 6, 2014

Abstract

We propose a natural syntactic account of extensional equality in type theory. Starting from the classical definition of extensional equality by induction on type structure, we show how the logical relation can be internalized in a way that covers all higher dimensions simultaneously. The result is a type theory in which every type has a (non-truncated) globular structure with degeneracy maps. Strong Normalization is expected to hold for the stratified version of this theory.

Going further, we show how the a priori meta-theoretic fact that every term of type theory preserves the extensional relation can likewise be internalized. Thus our theory is augmented with a new operator which computes the transport of a given term over a path in its context. This operator has canonical operational semantics much analogous to the reduction rules of explicit substitution calculi. By using normalizing reduction strategies, we may therefore ensure that type-checking remains decidable.

1 Background: the problem of extensionality

In recent years, the problem of extensionality in type theory has received increasing attention. In part, this is due to type theory emerging as the language of choice for computer formalisation of mathematics. (Gonthier, Asperti, Avigad, Bertot, Cohen, Garillot, Roux, Mahboubi, O’Connor, Biha, Pasca, Rideau, Solovyev, Tassi and Théry (2013), The Univalent Foundations Program (n.d.).)

The fundamental notion of this language, that of a type, is a notion of collection which bases membership on the syntactic form of the objects. Accordingly, the notion of equality between objects of a given type is likewise based on their syntactic form: two expressions are judged as denoting equal objects if one can be transformed into another by a finite sequence of syntactic manipulations.
Since a general number-theoretic function can in principle be implemented in any number of ways, there will be different expressions defining the same function which cannot be transformed from one to another using syntactic manipulations only. For example, the function which maps (a code of) a vector of numbers to a rearrangement of it listing the numbers in increasing order, can be implemented using bubble-sort or quick-sort processes, and these cannot be transformed into one another by local simplifications.

Therefore, basing equality on syntactic form alone leads to the failure of function extensionality, the principle stating that two functions are equal if they are pointwise equal:

$$\forall x, y \ (x =_A y \Rightarrow f x =_B gy) \implies f =_{A \rightarrow B} g$$

At the same time, this principle is deeply embedded into the language and culture of mathematics. Indeed, the encoding of the intuitive notion of a function into the formal language of set theory as a set of ordered pairs makes extensionality an inalienable component of the lexical meaning of the word “function”. The above principle then becomes a linguistic one.

In order to develop set-theoretic mathematics in type theory, it is convenient to introduce a notion of equality which would justify the above principle, and which could therefore be called extensional equality. Unfortunately, the known ways of doing so result in violation of key design principles of type theory.

The classical approach suggested by Martin-Löf (1984) is to extend the definitional (syntactic) equality between expressions, by allowing expressions to be judged as equal whenever the corresponding statement is proved in the system’s logic. (That is, mathematical equality is reflected back into the syntax). In general, such proofs cannot be synthesized mechanically, while checking that a term has a given type still requires checking that two terms are equal. As a result, type checking becomes undecidable, leaving type theory without one of its characteristic features.

A more recent idea is due to Voevodsky (2006), who discovered a single sentence in the language of type theory which, when assumed as an axiom, makes the intensional identity type behave like the extensional one.

1 Curiously, at the time of writing these words, the definition of the word “function” given by Google is that it is an “expression with one or more variables”. This corresponds to the type-theoretic notion of a function — a lambda term — and not the set-theoretic notion (a set of pairs having certain properties).

Likewise, the middle-school definition of a function as a “black box” which transforms its input into output is more faithfully captured by the lambda calculus viewpoint.
The additional benefit of working with this axiom is that one also obtains universe extensionality (Hofmann and Streicher (1996)), which states that isomorphic types are equal. (Indeed, the univalence axiom is a formulation of universe extensionality.)

However, assuming an axiom in type theory without specifying its combinatorial behavior with respect to other symbols results in the loss of another crucial property of canonicity. This is the property that every element of a given mathematical structure, defined by whatsoever means, can be computed by trivial simplification steps to a canonical representative for this structure. This global normal form theorem is one of the most attractive features of type theory, since it ensures that every proved theorem is fully effective, in the sense of conceptual computability.

Yet another approach is to define extensional equality by induction on type structure. Here one defines the equality relation externally to the type system, by relating certain elements of the (free) term model of the theory. (This is actually the oldest approach to the problem, having led Gandy (1956) to derive for the very first time the logical relations method, which today is the main tool used in meta-theoretic studies of type systems.)

In 1995, in a paper titled “Extensional Equality in the Classical Theory of Types”, William Tait proposed that the notion of extensional equality be identified with the canonical equivalence relation defined by the logical relations principle. Although this proposal has been independently investigated by others (Altenkirch, McBride and Swierstra (2007), Coquand (2011), Licata and Harper (2012)), its has so far been given limited visibility.

Let it thus be made explicit.

**Extensionality Thesis:** The extensional equality type is the canonical equivalence relation defined between elements of the term model of type theory by induction on type structure.

Tait (1995) treated the extensionality thesis in the context of Church’s simple type theory. Here we continue this line of research for Martin-Löf’s theory of type dependency. As compared to the efforts mentioned previously, our main contribution is the internalization of the meta-level logical relation via a new type constructor in a way that does not limit the resulting theory to a low dimension. (We do not make the usual assumption of working in a meta-theory with unique identity proofs.)

Our setup admits a uniform proof of the extensionality theorem proved by Tait. This theorem states that every term is “extensional” in the sense of preserving the relation on the types of its variables. Since the proof
of the theorem has manifest computational content, we are able to also internalize the theorem itself. This results in a new operator which allows one to transfer any mathematical construction over a path in the context where it was made.

We shall present these results according to the following schedule.

In the next section we present the system $\lambda e$. This is in some sense a minimal dependent type theory in which the extensionality of every term is witnessed within the system. Section 3 is devoted to proving Tait extensionality theorem for $\lambda e$. The operation sending a term $t$ to its extensionality witness $t^*$ is still defined meta-theoretically, however (indeed, it is essentially a kind of higher-dimensional substitution), so our next goal is to shift this operation into the object language as well. The internalization of the substitution operation is given in Section 4.

2 $\lambda e$

In this section, we describe a type theory in which extensionality of terms is witnessed by terms in the same system. This type theory, denoted $\lambda e$, admits a meta-level operation

$$ (\cdot)^*: \text{Terms}(\lambda e) \to \text{Terms}(\lambda e) $$

which raises by one the dimension of any given term.

The theory provides a type-level internalization of Tait’s extensionality theorem. Simply put, the theorem states that every $\lambda$-term preserves a given relation on the term model provided this relation is logical. The dependent version of the theorem requires one to consider a certain relation on the universe of types, and for every pair of types related by it, a new “heterogeneous” relation between terms of these types.

These two objects — the relation on the universe of types, which we call equivalence, and a relation between $A$ and $B$ whenever $A$ and $B$ are equivalent — are added as new type constructors into the language. That the equivalence relation between types is preserved by every type constructor is witnessed by new term constructors dedicated to this purpose. (They can thus be seen as constructors for the equivalence type.) The relation induced by such a constructor is definitionally expressed in terms of previously defined types, and can be seen as the result of eliminating a witness of equivalence between the two types. There are thus no term constructors for the heterogeneous relation type itself — this type is always rewritten whenever more information about the equivalence becomes available.
As our base system, we take $\lambda^*$, the “naive type theory” where the universe of all types is itself a type. This choice is motivated by the fact that, although inconsistent, this system is by far and away the simplest formulation of dependent type theory. Furthermore, the standard recipe for turning an inconsistent type theory into a consistent one by stratifying the universes can be applied to our case in a straightforward manner, without affecting the proofs. The stratification of $\lambda e$ is given in the Appendix.

The system $\lambda e$ is an extension of $\lambda^*$ presented as follows.

Syntax:

\[
\begin{align*}
A, B, s, t, e & ::= \ast | x | \Pi x.A.B | \Sigma x.A.B | A \Rightarrow B | a \sim_b b \\
& | \lambda x.A.t | st | (s, t) | \pi_1 t | \pi_2 t \\
& | \ast | \Pi^* [x, x', x^*] : A.B | \Sigma^* [x, x', x^*] : A.B | z^* e e
\end{align*}
\]

Typing (curly braces demarcate implicit arguments):

\[
\begin{align*}
\Gamma & \vdash \ast : \ast \\
\Gamma & \vdash A : \ast \\
\Gamma, x & : A \vdash x : A \\
\Gamma & \vdash M : A \\
\Gamma & \vdash B : \ast \\
\Gamma, y & : B \vdash M : A \\
\Gamma & \vdash A : \ast \\
\Gamma, x & : A \vdash B : \ast \\
\Gamma & \vdash \Pi x.A.B : \ast \\
\Gamma & \vdash \Sigma x.A.B : \ast \\
\Gamma & \vdash A : \ast \\
\Gamma, x & : A \vdash B : \ast \\
\{ \{ \Gamma \vdash A : \ast \} \} & \{ \{ \Gamma \vdash B : \ast \} \} \\
\Gamma & \vdash e : A \Rightarrow B \\
\Gamma & \vdash a : A \\
\Gamma & \vdash b : B \\
\Gamma & \vdash a \sim_b b : \ast \\
\{ \{ \Gamma \vdash A : \ast \} \} & \{ \{ \Gamma, x \vdash A \Rightarrow B : \ast \} \} \\
\Gamma, x & : A \vdash b : B \\
\frac{\Gamma \vdash \lambda x.A.b : \Pi x.A.B}{\Gamma \vdash \lambda x.A.b : \Pi x.A.B} \\
\{ \{ \Gamma \vdash A : \ast \} \} & \{ \{ \Gamma, x : A \vdash B : \ast \} \} \\
\Gamma & \vdash f : \Pi x.A.B \\
\Gamma & \vdash a : A \\
\Gamma & \vdash f a : B[a/x]
\end{align*}
\]
\[ \{ \Gamma \vdash A : * \} \quad \{ \Gamma, x : A \vdash B : * \} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x] \]

\[ \Gamma \vdash (a, b) : \Sigma x : A.B \]

\[ \{ \Gamma \vdash A : * \} \quad \{ \Gamma, x : A \vdash B : * \} \quad \Gamma \vdash p : \Sigma x : A.B \]

\[ \Gamma \vdash \pi_1 p : A \]

\[ \Gamma \vdash \pi_2 p : B[\pi_1 p/x] \]

\[ \Gamma \vdash M : A \quad \Gamma \vdash B : * \quad A = B \]

\[ \Gamma \vdash M : B \]

\[ \ast^* \vdash \ast \leq \ast \]

\[ \{ \Gamma \vdash A : * \} \quad \{ \Gamma, x : A \vdash B : * \} \quad \{ \Gamma \vdash A' : * \} \quad \{ \Gamma, x' : A' \vdash B' : * \} \quad \Gamma \vdash A^* : A \leq A' \quad \Gamma, x : A, x' : A', x^* : x \leq A, x^* \vdash B^* : B \leq B' \]

\[ \Pi^*[x, x', x^*] : A^*.B^* : \Pi x.A.B \leq \Pi x'.A'.B' \]

\[ \Sigma^*[x, x', x^*] : A^*.B^* : \Sigma x.A.B \leq \Sigma x'.A'.B' \]

\[ \{ \Gamma \vdash A : * \} \quad \{ \Gamma \vdash B : * \} \quad \{ \Gamma \vdash A' : * \} \quad \{ \Gamma \vdash B' : * \} \quad \Gamma \vdash A^* : A \leq A' \quad \Gamma \vdash B^* : B \leq B' \]

\[ \ast^* \vdash A^*B^* : (A \leq B) \leq (A' \leq B') \]

Reduction:

\[ (\lambda x : A.s)t \quad \rightarrow \quad s[t/x] \]

\[ \pi_1(s_1, s_2) \quad \rightarrow \quad s_i \]

\[ A \sim_{\ast^*} B \quad \rightarrow \quad A \leq B \]

\[ f \sim_{\Pi^*[x, x', x^*] : A^*.B^*} f' \quad \rightarrow \quad \Pi x : A\Pi x'.A'\Pi x^* : x \leq_{A^*} x'. f x \leq_{B^*} f' x' \]

\[ p \sim_{\Sigma^*[x, x', x^*] : A^*.B^*} p' \quad \rightarrow \quad \Sigma a^* : \pi_1 p \leq_{A^*} \pi_1 p'. \pi_2 p \leq_{B^*} \pi_2 p'[\pi_1 p, \pi_1 p', a^*/x, x', x^*] \pi_2 p' \]

\[ e \sim_{\ast^*} a^*B^* e' \quad \rightarrow \quad \Pi a : A\Pi a'.A'\Pi a^* : a \leq_{A^*} a'. \Pi b : B\Pi b'.B'\Pi b^* : b \leq_{B^*} b'. \quad (a \sim_{e} b) \leq (a' \sim_{e'} b') \]

In the remainder, we will often see a pattern where three operations of the same type appear in a row, like the triple-product sequences in the above. To reduce clutter in such expressions, we introduce the following notations.
\[
\prod \left( \begin{array}{c}
x : A \\
y : B \\
z : C
\end{array} \right) T := \Pi x : A \Pi y : B \Pi z : C. T
\]
\[
\lambda \left( \begin{array}{c}
x : A \\
y : B \\
z : C
\end{array} \right) t := \lambda x : A \lambda y : B \lambda z : C. t
\]
\[
M \left( \begin{array}{c}
N_1 \\
N_2 \\
N_3
\end{array} \right) := MN_1 N_2 N_3
\]
\[
M \left[ \begin{array}{c}
al/x \\
b/y \\
c/z
\end{array} \right] := M[a/x][b/y][c/z]
\]
\[
\Pi^* \left[ \begin{array}{cc}
x & y \\
z
\end{array} \right] : A^* . B^* := \Pi^*[x, y, z] : A^* . B^*
\]
\[
\Sigma^* \left[ \begin{array}{cc}
x & y \\
z
\end{array} \right] : A^* . B^* := \Sigma^*[x, y, z] : A^* . B^*
\]

Using these abbreviations, the rules for ∼-reduction look like

\[
A \sim^* B \rightarrow A \equiv B
\]
\[
f \sim^* \Pi^* A^* B^* f' \rightarrow \Pi \left( \begin{array}{c}
a : A \\
a' : A'
\end{array} \right) f \equiv B^*[a, a'/x, x', x^*] f' x'
\]
\[
p \sim^* \Sigma^* A^* B^* p' \rightarrow \Sigma \left( \begin{array}{c}
a : \pi_1 p \sim^* \pi_1 p'
\end{array} \right) \pi_2 p \equiv B^*[\pi_1 p, \pi_1 p', a'/x, x', x^*] \pi_2 p'
\]
\[
e \sim^* A^* B^* e' \rightarrow \Pi \left( \begin{array}{c}
a : A \\
a' : A'
\end{array} \right) \Pi \left( \begin{array}{c}
b : B \\
b' : B'
\end{array} \right) (a \sim e b) \equiv (a' \sim e' b')
\]

Notice that the type “constructor” ∼ acts much like a defined function: whenever reduction of \( e \) yields a root symbol which is a constructor (neither a variable nor a redex), then the definition of \( a \sim e b \) can be immediately unfolded based on that symbol.

In fact, the constructions which are to follow are technically much cleaner if we take this equivalent approach, and thus we do so. The reason for the presentation above is the geometric interpretation of \( a \sim e b \) as the type of “dependent paths” from \( a \) to \( b \) lying over the path in the context which induced the equivalence \( e \).

We now make precise the procedure whereby all of the glucose in this choice of notation may be removed.
1. The \( \sim_e \)-formation rule is replaced by the rule
\[
\frac{\{ A : * \} \{ B : * \} \ e : A \equiv B}{\bar{e} : A \rightarrow B \rightarrow *}
\]

2. Every occurrence of \( a \sim_e b \) is replaced by
\[
\bar{e}ab
\]

3. Each of the reduction rules having the form
\[
a \sim_e b \quad \rightarrow \quad T[e,a,b]
\]
is redisplayed as
\[
\bar{e} \quad \rightarrow \quad \lambda a : A \lambda b : B. T[e,a,b]
\]

4. When \( e \) is large, the term \( \bar{e} \) may be rendered as \( e \sim \) or \( \sim(e) \).

With these conventions, the \( \sim \)-reduction rules may be rendered as
\[
A \sim_e B \quad \rightarrow \quad A \equiv B
\]

\[
f \sim_{\Pi^* A^* B^*} f' \quad \rightarrow \quad \prod \left( a : A \atop a' : A' \atop a^* : a \sim A \ast a' \right) \sim_{B^* \left( \frac{a' / x'}{a^* / x^*} \right)} f x f' x'
\]

\[
p \sim_{\Sigma^* A^* B^*} p' \quad \rightarrow \quad \sum \left( a^* : \Pi_{1 \rightarrow A^*} \pi_{1p} \atop a^* : \Sigma_{1 \rightarrow A^*} \pi_{1p'} \right) \sim_{B^* \left( \frac{\pi_{1p}/x}{\pi_{1p'}/x'} \right)} \pi_{2p} \pi_{2p'}
\]

\[
e \sim_{\ast^* A^* B^*} e' \quad \rightarrow \quad \prod \left( a : A \atop a' : A' \atop a^* : a \sim A \ast a' \right) \prod \left( b : B \atop b' : B' \atop b^* : b \sim B \ast b' \right) (a \sim_e b) \equiv (a' \sim_e b')
\]

Throughout the rest, we will continue to use the notation \( a \sim_e b \) for its visual merit. We shall call this the implicit notation; the alternative form \( \bar{e}ab \) will be referred to as the explicit notation.

3. **The \((\cdot)^*\) operator**

We now define the map \((\cdot)^* : \text{Terms}(\lambda e) \rightarrow \text{Terms}(\lambda e)\). The intention is that, if \( t(x_1, \ldots, x_n) : A(\bar{x}) \) is a term, then
\[
t^* = t^* \left( \begin{array}{c} x_1 \\ x'_1 \\ \vdots \\ x_n \\ x'_n \end{array} \right) : t(\bar{x}) \sim_{A^* (x_1^*, \ldots, x_n^*)} t'(\bar{x}')
\]
gives the transport of $t$ over a formal path in the context.

**Definition.** Let $t \mapsto t'$ be the operation of apostrophizing every variable, bound or otherwise.

**Lemma.**

- $(M[N/x])'=M'[N'/x']$
- $M=N \implies M'=N'$
- $\Gamma \vdash M:A \implies \Gamma' \vdash M':A'$

**Proof.** Kickass. \hfill $\square$

**Definition.** The operation $t \mapsto t^*$ is defined by induction on the term structure.

In the equations that follow, the symbols $A_*, B_*, a_*$, etc. are free variables: the appearance of $*$ in a subscript is merely a suggestive choice of naming the variables.

\[
\begin{align*}
(*)^* &= *^* \\
(x)^* &= x^* \\
(\Pi x:A.B)^* &= \Pi^*\left[\begin{array}{c} x \\ x' \\ x^* \end{array}\right] : A^* \cdot B^* \\
(\Sigma x:A.B)^* &= \Sigma^*\left[\begin{array}{c} x \\ x' \\ x^* \end{array}\right] : A^* \cdot B^* \\
(A \simeq B)^* &= z^* A^* B^* \\
(\sim e)^* &= e^* \\
(\lambda x:A.b)^* &= \lambda x:A \lambda x':A' \lambda x^*: x \sim A^* x'. b^* \\
(fa)^* &= f^*aa^*a^* \\
(a,b)^* &= (a^*,b^*) \\
(\pi_1 p)^* &= \pi_1 p^* \\
(\pi_2 p)^* &= \pi_2 p^*
\end{align*}
\]
\[(*)^* = \lambda (A : B) \lambda (A' : B') \lambda (A \mapsto A') \lambda (B' : B = B'). \]
\[
(\Pi^*[x, x_1, x_2] : A_1, B_1)^* = \lambda \left( f : \Pi x : A, B \quad f' : \Pi x : A', B' \quad f'' : f' \mapsto A^*,B^* \right) \lambda \left( f_1 : \Pi x_1 : A_1, B_1 \quad f_1' : \Pi x_1' : A_1', B_1' \right).
\]
\[
\Pi^*[\begin{bmatrix} a \\ a' \\ a^* \end{bmatrix} : A^*] \times \Pi^*[\begin{bmatrix} a_1 \\ a_1' \end{bmatrix} : A_1^*] \times \Pi^*[\begin{bmatrix} a^* \end{bmatrix} : A^*] = \Pi\left( a \quad a' \quad a^* \quad a_1 \quad a_1' \quad a^* \right)
\]
\[
(\Sigma^*[x, x_1, x_2] : A_1, B_1)^* = \lambda \left( p : \Sigma x : A, B \quad p' : \Sigma x : A', B' \quad p'' : p' \mapsto A^*,B^* \right) \lambda \left( p_1 : \Sigma x_1 : A_1, B_1 \quad p_1' : \Sigma x_1' : A_1', B_1' \right).
\]
\[
\Sigma^*[\begin{bmatrix} a_1 \quad a_{p_1} \quad a_{p_1'} \quad a^* \end{bmatrix} : A_1^*] \times \Pi^*[\begin{bmatrix} a_1 \end{bmatrix} : A_1^*] \times \Pi^*[\begin{bmatrix} a_1' \end{bmatrix} : A_1^*] = \Sigma\left( a_1 \quad a_{p_1} \quad a_{p_1'} \quad a^* \right)
\]
\[
(\varepsilon^* A_1 B_1)^* = \lambda \left( e : A \mapsto B \quad e' : e \mapsto e' \right) \lambda \left( e_1 : B_1 \quad e_1' : e_1 \mapsto b_1 \right).
\]
\[
\Pi^*[\begin{bmatrix} a \\ a' \\ a^* \end{bmatrix} : A^*] \times \Pi^*[\begin{bmatrix} a_1 \\ a_1' \end{bmatrix} : A_1^*] \times \Pi^*[\begin{bmatrix} a^* \end{bmatrix} : A_1^*] = \Pi\left( a \quad a' \quad a^* \quad a_1 \quad a_1' \quad a^* \right)
\]

**Lemma.** \((M[N/x])^* = M^*[N/x, N'/x', N^*/x^*]\)

**Proof.** By induction on the structure of \(M\).

\(\square\)

**Lemma.** \(M = N \implies M^* = N^*\)

**Proof.** By induction on the length of the reduction–expansion sequence in \(M = N\), it suffices to show

\[ M \rightarrow N \implies M^* \rightarrow N^* \quad (1) \]

First we argue that it is enough to consider contractions at the root of the term.
Indeed, suppose that \( M = C[s] \), \( N = C[t] \), and \( s \rightarrow t \) by contraction at the root.

Let \( C_0 = C[x_0] \), where \( x_0 \) is fresh.

Using Lemma 3, we write

\[
M^* = C_0[s/x_0]^* = C_0[s/x_0, s'/x_0', s^*/x_0^*]
\]

\[
N^* = C_0[t/x_0]^* = C_0[t/x_0, t'/x_0', t^*/x_0^*]
\]

Since \( \beta \)-reduction is itself a congruence, it suffices to verify that the instances of each variable are reducible. That \( s \rightarrow t \) and \( s' \rightarrow t' \) is clear; that \( s^* \rightarrow t^* \) remains to be proved.

There is thus no loss of generality in assuming that the redex is contracted at the root.

We now treat each reduction rule in order. For the \( \sim \)-rules, we use the explicit notation of dependent equality type.

\( \beta \) Given \( s = (\lambda x : A.M)N \rightarrow M[N/x] = t \), we are to show that

\[
((\lambda x : A.M)N)^* \rightarrow (M[N/x])^*
\]

Indeed,

\[
((\lambda x : A.M)N)^* = (\lambda x : A.M)^*NN'^*N^*
\]

\[
= (\lambda x : A\lambda x' A': \lambda x^* : x \sim_A x'.M^*)NN'^*N^*
\]

\[
\rightarrow M^*[N/x, N'/x', N^*/x^*]
\]

\[
= M[N/x]^*
\]

by Lemma 3. Thus \( s^* \rightarrow t^* \).

\( \beta \Sigma \) We have

\[
(\pi_1(M, N))^* = \pi_1(M, N)^* = \pi_1(M^*, N^*) \rightarrow M^*
\]

\[
(\pi_2(M, N))^* = \pi_2(M, N)^* = \pi_2(M^*, N^*) \rightarrow N^*
\]

\( \ast \ast \) Consider

\[
\sim_* \quad \rightarrow \quad \lambda A : * \lambda B : * , A \simeq B
\]

We have

\[
(\sim_*^*)^* = \lambda A : * \lambda A' : * \lambda A^* : A \sim_* A'
\]

\[
\lambda B : * \lambda B' : * \lambda B^* : B \sim_* B' \sim_* B' , \simeq A^* B^*
\]

\[
= (\lambda A : * \lambda B : * , A \simeq B)^*
\]
Consider
\[ \sim (\Pi^* [x_1, x_n] : A_s \cdot B_s) \]
\[ \rightarrow \lambda f : \Pi x.AB \lambda f_1 : \Pi x_1 : A_1 \cdot B_1. \]
\[ \Pi a : A \Pi a_1 : A_1 \Pi a_2 : a \sim_{A_s} a_1. \]
\[ f a \sim_{B_s [\alpha a_1 a_2 / x_1 x_n]} (f_1 a_1) \]

Let \( T \) be the reduct on the right. We have
\[ \sim (\Pi^* [x_1, x_n] : A_s \cdot B_s)^* \]
\[ = \prod^* \left[ \begin{array}{c}
\alpha \\frac{x}{a} \\
\alpha' \\frac{x'}{a'}
\end{array} \right]: A^* \Pi^* \left[ \begin{array}{c}
\alpha_1 \\frac{x_1}{a_1} \\
\alpha'_1 \\frac{x'_1}{a'_1}
\end{array} \right]: A_1^* \Pi^* \left[ \begin{array}{c}
\alpha_2 \\frac{x_2}{a_2} \\
\alpha'_2 \\frac{x'_2}{a'_2}
\end{array} \right]: A_2^* \left( \frac{a}{a'} \left( \frac{a_1}{a_1'} \right) \right). \]

By inspection, this is exactly \( T^* \). Let’s check the innermost quantifier:
\[ (\Pi a_1 : a \sim_{A_s} a_1. \]
\[ f a \sim_{B_s [\alpha a_1 a_2 / x_1 x_n]} (f_1 a_1) \]
\[ = \prod^* \left[ \begin{array}{c}
\alpha \\frac{x}{a} \\
\alpha' \\frac{x'}{a'}
\end{array} \right]: A^* \left( \frac{a}{a'} \left( \frac{a_1}{a_1'} \right) \right). \]
\[ (\sim B_s [\alpha a_1 a_2 / x_1 x_n] (f_a) (f_1 a_1))^* \]
\[ (\sim B_s [\alpha a_1 a_2 / x_1 x_n] (f_a) (f_1 a_1))^* \]
\[ = (B_s [\alpha a_1 a_2 / x_1 x_n] (f_a) (f_1 a_1))^* \]
\[ = B_s^* \left[ \begin{array}{c}
\alpha \frac{a}{x} \\
\alpha' \frac{a'}{x'}
\end{array} \right]: a \sim_{A_s} a_1 \left( \frac{a}{a'} \left( \frac{a_1}{a_1'} \right) \right). \]

as required.

Other cases are treated similarly.

\[ \square \]

**Definition.** Let \( \Gamma = (x_1 : A_1, \ldots, x_n : A_n) \) be a context. Put
\[ \Gamma^* := \left( \begin{array}{cccc}
x_1 : A_1 \\
x_1' : A_1'
\end{array} \right) \]
\[ \left( \begin{array}{cccc}
x_n : A_n \\
x_n' : A_n'
\end{array} \right) \]
\[ \left( \begin{array}{cccc}
x_1 \sim_{A_1} x_1' \\
x_n \sim_{A_n} x_n'
\end{array} \right) \]

12
Theorem. \[ \Gamma \vdash M : A \quad \implies \quad \Gamma^* \vdash M^* : M \sim_{A^*} M' \]

Proof. We proceed by induction on \( \Gamma \vdash M : A \).

Axiom For the axiom rule \( \Gamma \vdash \, * : * \), we have

\[ \vdash \, * : * \sim * \]

By conversion rule, \( \vdash \, * : * \sim_{A^*} * \).

Variable Given the derivation

\[ \frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A} \]

we have, by induction hypothesis, that

\[ \Gamma^* \vdash A^* : A \sim_{A^*} A' \]

and hence \( A^* : A \equiv A' \).

Clearly, \( \Gamma^* \equiv \Gamma' \vdash * : * \).

Then \((\Gamma, x : A)^* = (\Gamma^*, x : A, x^* : x \sim_{A^*} x')\) is a valid context, and

\[ (\Gamma, x : A)^* \vdash x^* : x \sim_{A^*} x' \]

Weakening Given the derivation

\[ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : *}{\Gamma, y : B \vdash M : A} \]

the induction hypothesis gives that

\[ \Gamma^* \vdash M^* : M \sim_{A^*} M' \]
\[ \Gamma^* \vdash B^* : B \equiv B' \]

Then \((\Gamma, y : B)^* = (\Gamma^*, y : B, y' : B', y^* : y \sim_{B^*} y')\) is a valid context, and

\[ (\Gamma, y : B)^* \vdash M^* : M \sim_{A^*} M' \]

(by applying weakening thrice).
**Formation of** $\Pi, \Sigma$ Suppose we are given

$$\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *$$

$$\Gamma \vdash \Pi x.A.B : *$$

Induction gives

$$\Gamma^* \vdash A^* : A \simeq A'$$

$$\Gamma^*, x:A, x':A', x^* : x \sim A^*, x' \vdash B^* : B \simeq B'$$

By lemmata, we also have apostrophized versions of these:

$$\Gamma' \vdash A' : *$$

$$\Gamma', x' : A' \vdash B' : *$$

Together, the given data, the primed version, and the inductive version, provide the hypotheses necessary for the application of the $\Pi^*$-rule:

$$\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *$$

$$\Gamma' \vdash A' : * \quad \Gamma', x' : A' \vdash B' : *$$

$$\Gamma \vdash A^* : A \simeq A' \quad \Gamma, x : A, x' : A', x^* : x \sim A^*, x' \vdash B^* : B \simeq B'$$

$$\Pi^*[x, x', x^*] : A^*.B^* : \Pi x.A.B \simeq \Pi x'.A'.B'$$

Since $(\Pi x.A.B)^* = \Pi^*[x, x', x^*] : A^*.B^*$, the above judgement has the desired form.

The case of $\Sigma$-formation is treated congruently.

**$\sim$-Formation** This is like the previous case, but easier; given

$$\Gamma \vdash A : * \quad \Gamma \vdash B : *$$

we have that $\Gamma' \vdash A' : *, \Gamma' \vdash B' : *$, and also, by induction, that

$$\Gamma^* \vdash A^* : A \simeq A'$$

$$\Gamma^* \vdash B^* : B \simeq B'$$

These data allow us to apply the $\sim^*$-rule, yielding

$$\Gamma^* \vdash \sim^* A^*B^* : (A \simeq B) \simeq (A' \simeq B')$$

which type converts to $(A \simeq B) \sim_{*} (A \simeq B)'$, as required.
\textbf{\textit{\textsim\text{-Formation}}} Suppose we are given

\[
\begin{array}{c}
\Gamma \vdash A : * \\
\Gamma \vdash B : * \\
\Gamma \vdash e : A \simeq B
\end{array}
\]

Then we have $\Gamma' \vdash \textit{\textsim} e' : A' \rightarrow B' \rightarrow *$, and by induction

\[
\Gamma^* \vdash e^* : e \sim (A \Rightarrow B)^* e'
\]

We work in $\Gamma^*$. Reducing the type of $e^*$, we get

\[
e^* : \Pi \left( \frac{a : A}{a'} \right) \Pi \left( \frac{b : B}{b'} \right) \left( (a \sim_e b) \simeq (a' \sim_{e'} b') \right)
\]

When written in explicit form, this looks like

\[
e^* : \Pi \left( \frac{a}{a'} \right) \Pi \left( \frac{b}{b'} \right) \left( \sim_e a b \simeq \sim_{e'} a'b' \right)
\]

By definition of $(\cdot)^*$,

\[(A \rightarrow B \rightarrow *)^* = \Pi^*[x, x', x^*] : A^* \Pi^*[y, y', y^*] : B^*, *^*
\]

Thus, for any $E : A \rightarrow B \rightarrow *$, $E' : A' \rightarrow B' \rightarrow *$, we find that

\[
E \sim_{(A \rightarrow B \rightarrow *)^*} E' = \Pi \left( \frac{x : A}{x'} \right) \Pi \left( \frac{y : B}{y'} \right) E x y \simeq E' x y'
\]

In particular, the type of $e^*$ is exactly

\[(\sim_e) \sim_{(A \rightarrow B \rightarrow *)^*} (\sim_{e'})
\]

Since $(\sim^*) = e^*$, we conclude that

\[
\Gamma^* \vdash (\sim^*) : (\sim_e) \sim_{(A \rightarrow B \rightarrow *)^*} (\sim_{e'})
\]

\textbf{Abstraction} Given

\[
\begin{array}{c}
\Gamma \vdash A : * \\
\Gamma, x : A \vdash B : * \\
\Gamma, x : A \vdash b : B
\end{array}
\]

\[
\Gamma \vdash \lambda x : A. b : \Pi x : A. B
\]
We have, by IH, that
\[
\Gamma^* \vdash A^* : A \\
\Gamma^*, x : A, x' : A', x^* : x_{\sim A} \cdot x' \vdash B^* : B \cong B' \\
\Gamma^*, x : A, x' : A', x^* : x_{\sim A} \cdot x' \vdash b^* : b \sim_{B^*} b'
\]

Observe that our target type converts as
\[
(\lambda x : A. b)^* \sim_{(\Pi x : A. B)^*} (\lambda x' : A'. b')^*
\]

The subject of this judgement is equal to \((\lambda x : A. b)^*\).

The type predicate converts to
\[
(\lambda x : A. b)^* \sim_{(\Pi x : A. B)^*} (\lambda x' : A'. b')^*.
\]

**Application** Suppose we are given
\[
\begin{array}{llll}
\Gamma \vdash A : * & \Gamma, x : A \vdash B : * & \Gamma \vdash f : \Pi x : A. B & \Gamma \vdash a : A \\
\hline
\Gamma \vdash fa : B[a/x]
\end{array}
\]

The induction hypotheses are
\[
\begin{array}{ll}
\Gamma^* \vdash A^* : A \cong A' \\
\Gamma^*, x : A, x' : A', x^* : x_{\sim A} \cdot x' \vdash B^* : B \cong B' \\
\Gamma^* \vdash f^* : f \sim_{(\Pi x : A. B)^*} f' \\
\Gamma^* \vdash a^* : a \sim_{A^*} a'
\end{array}
\]

Working in \(\Gamma^*\), we need to show that
\[
(\text{fa})^* : f a \sim_{B[a/x]^*} f' a'
\]
Equivalently, we need to show that

\[ f^* a^* a^* : f a \sim B[a,a',a^*/x,x',x^*] f' a' \]  \(3\)

(where we used the substitution lemma to rewrite \(B[a/x]^*\)). Applying the conversion rule to (2) gives

\[ f^* := \prod \left( \frac{x : A^'}{x' : x^*} \right) f x \sim_B f' x' \]

Then, by a triple use of the application rule, we have

\[ f^* a^* a^* : f a \sim B^*[a,a',a^*/x,x',x^*] f' a' \]

which is typographically consistent with (3).

**Pairing** Let us be given

\[ \Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x] \]

\[ \Gamma \vdash (a,b) : \Sigma x : A.B \]

We work in \(\Gamma^*\). By induction, we have

\[ a^* : a \sim_A a' \]
\[ b^* : b \sim_B[a/x], b' \]

We may rewrite the latter as

\[ b^* : b \sim_B[a,a',a^*/x,x',x^*] b' \]  \(4\)

Using these data, the following sequence of judgements may be verified:

\[ (\Gamma, x : A)^* \vdash B^* : B \approx B' \]
\[ \Gamma^*, x^* : a \sim_{A^*} a' \vdash B^*[a,a',a^*/x,x',x^*] : B[a/x] \approx B'[a'/x'] \]
\[ \Gamma^*, x^* : a \sim_{A^*} a' \vdash b \sim_B[a,a',a^*/x,x',x^*] b' : * \]
\[ \Gamma^* \vdash b \sim_B[a,a',a^*/x,x',x^*] b' : * \]
\[ \Gamma^* \vdash (a^*, b^*) : \Sigma a^* : a \sim_{A^*} a'. b \sim_{B^*[a,a',a^*/x,x',x^*]} b' \]
\[ \Gamma^* \vdash (a^*, b^*) : (a,b) \sim_{\Sigma^*[x,x',x^*] A^*, B^*} (a', b') \]
\[ \Gamma^* \vdash (a,b)^* : (a,b) \sim_{(\Sigma x : A.B)^*} (a', b') \]
Projections  
Next, we consider the inference rules

\[
\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \quad \Gamma \vdash \Sigma x. A \cdot B}{\Gamma \vdash \pi_2 p : B[\pi_1 p / x]}
\]

We have

\[
\begin{align*}
\Gamma^* & \vdash p^* : p \sim (\Sigma x. A \cdot B)^* p' \\
\Gamma^* & \vdash p^* : p \sim \Sigma[x, x', x^*] A^*. B^* p'
\end{align*}
\]

The judgements (p1) and (p2) are of the required form.

Conversion  
Next, suppose we are given the inference

\[
\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : * \quad A = B}{\Gamma \vdash M : B}
\]

We are to show that \( \Gamma^* \vdash M^* : M \sim_{B^*} M' \).

From the given data, we know that

\[
\begin{align*}
M & : B \\
M' & : B' \\
B^* & : B \simeq B'
\end{align*}
\]

Thus

\[M \sim_{B^*} M' : *\] (5)

By Lemma 1, we have

\[M \sim_{A^*} M' = M \sim_{B^*} M'\] (6)

By IH, we also have \( \Gamma^* \vdash M^* : M \sim_{A^*} M' \). Using (5) and (6), we may apply the conversion rule to this judgment to obtain

\[
\Gamma^* \vdash M^* : M \sim_{B^*} M'
\]
Let us be given

\[ ** : * \cong * \]

We are asked to show that

\[
(*^\ast)^* : * \cong (\ast^* \ast^*)^* \ast^*
\]

\[
(*^\ast)^* : * \cong (\ast^* \ast^*)^* \ast^*
\]

\[
(*^\ast)^* : \Pi \left( \begin{array}{c} A : * \\ A' : * \\ A'' : A \end{array} \right) \Pi \left( \begin{array}{c} B : * \\ B' : * \\ B'' : B \end{array} \right). (A \cong_* B) \cong (A' \cong_* B')
\]

Unfolding the definition of \((\ast^\ast)^\ast\), we have

\[
(*^\ast)^\ast = \chi \left( \begin{array}{c} A : * \\ A' : * \\ A' = A' \end{array} \right) \chi \left( \begin{array}{c} B : * \\ B' : * \\ B' = B' \end{array} \right). \ast^\ast A^\ast B^\ast
\]

By inspection, this term has the desired type.

\textbf{\Pi\footnotesize{-congruence}} Let us be given

\[
\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *
\]

\[
\Gamma \vdash A_1 : * \quad \Gamma, x_1 : A_1 \vdash B_1 : *
\]

\[
\Gamma \vdash A_2 : A \cong A_1 \quad \Gamma, x, x_1 : A_1, x_2 : x \cong A_2, x_1 \vdash B_2 : B \cong B_1
\]

\[
\Pi^\ast [x, x_1, x_2] : A_2. B_2 : \Pi x : A.B \cong \Pi x_1 : A_1.B_1
\]

Recall that

\[
(\Pi^\ast [x, x_1, x_2] : A_2. B_2)^\ast = \chi \left( \begin{array}{c} a \\ a' \\ a^* \end{array} \right) \Pi^\ast \left( \begin{array}{c} a_1 \\ a_1' \\ a_1^* \end{array} \right) : \Pi^\ast \left( \begin{array}{c} a_1 \\ a_1' \\ a_1^* \end{array} \right) : A_2^\ast \Pi^\ast \left( \begin{array}{c} a_1 \\ a_1' \\ a_1^* \end{array} \right) : A_2^\ast \Pi^\ast \left( \begin{array}{c} a_1 \\ a_1' \\ a_1^* \end{array} \right) : A_2^\ast \Pi^\ast \left( \begin{array}{c} a_1 \\ a_1' \\ a_1^* \end{array} \right).
\]

We are to show that this term has type

\[
\Pi^\ast A_2 B_2 \cong (\Pi x : A.B)^\ast (\Pi x_1 : A_1.B_1)^\ast, \Pi^\ast A_2' B_2'
\]
First, note that the type of the equivalence appearing in the index of the dependent relation in (8) is

\[(\Pi x:A.B \simeq \Pi x:A_1.B_1) \simeq (\Pi x:A'.B' \simeq \Pi x:A'_{1}.B'_{1})\]

(This is by the induction hypothesis on \(\Pi x:A.B\) and \(\Pi x:A_1.B_1\). Although “\(\Pi x:A.B : *\)” does not appear among the premises of this rule, the required statement can be obtained by inlining the proof of the \(\Pi\)-formation case. The hypotheses there are provided by the induction hypotheses on the premises given here.)

We begin by looking closer at the relation associated to this equivalence. By \(\sim\)-reduction, we have

\[\sim(z^*(\Pi x:A.B)^*(\Pi x_1:A_1.B_1)^*)) = \lambda e : (\Pi x:A.B \simeq \Pi x_1:A_1.B_1) \lambda e' : (\Pi x':A'.B' \simeq \Pi x'_1:A'_{1}.B'_{1}).\]

\[\prod \left( f : \Pi x:A.B \atop f' : \Pi x':A'.B' \atop f' : f \sim (\Pi x:A.B) \ast f' \right) \prod \left( f_1 : \Pi x_1:A_1.B_1 \atop f'_1 : \Pi x'_1:A'_{1}.B'_{1} \atop f'_1 : f_1 \sim (\Pi x_1:A_1.B_1) \ast f'_1 \right) \]

\[(f \sim e f_1) \simeq (f' \sim e f'_1)\]

When this term is applied to \(\Pi^*A.B_s\) and \(\Pi^*A'_sB'_s\), so as to become (4), we see immediately that \(\lambda\)-abstractions appearing at the root of (7) match correctly the \(\Pi\)-types above.

We are thus left to verify that the matrix of these abstractions — the triple-\(\Pi^*\) subexpression of (7) — has type

\[(f \sim \Pi^*A.B_s f_1) = (f' \sim \Pi^*A'_sB'_s f'_1) = (9)\]

\[\Pi x.\Pi a_1: A_1 \Pi a_2: A_2 : a \sim A_s a_1.(fa \sim B_s[a,a_1,a_2/x,x_s] f_1 a_1) \]

\[= \Pi a'_s:A_s \Pi a'_1: A'_1 : a' \sim A'_s a'_1.(f' a' \sim B'_s[a',a'_1,a'_2/x',x'_s] f'_1 a'_1)\]

Looking at (7) again, we see that the equivalences specified in the triple-\(\Pi^*\) expression correctly match the domains of the products above.

That the third equivalence

\[A_2^s \left( \begin{array}{c} a \\ a' \\ a^* \end{array} \right) \left( \begin{array}{c} a_1 \\ a'_1 \\ a^*_1 \end{array} \right) : (a \sim A_s a_1) \simeq (a' \sim A'_s a'_1)\]

2 We recall that in order to construct an equivalence between two \(\Pi\) types, one needs to construct an equivalence between their domains of quantification, and, for every dependent line between these domains (ie, a pair of terms related by the equivalence), an equivalence between the corresponding fibers of the dependent type (quantification matrices).

This possibility is precisely the content of the \(\Pi^*\)-constructor, which gives an equivalence between two \(\Pi\) terms from an equivalence between their domains and a map transporting paths between domains to equivalences of fibers.
has the right type uses induction hypothesis on \(A_s\).

By induction hypothesis on \(B_s\), we have that

\[
(\Gamma, x : A, x_1 : A_1, x_a : x \sim_{A_s} x_1)^* \vdash B_s^* : B_s \sim_B B^*_1 B_s'
\]

or, using alternative notation,

\[
(\Gamma, x : A, x_1 : A_1, x_a : x \sim_{A_s} x_1)^* \vdash B_s^* : (z^* B^* B_s^*)^* B_s B_s'
\]

Plugging in the terms in our context, we get

\[
\Gamma^* \vdash B_s^* \begin{bmatrix} a/x \\ a'/x' \\ a^*/x^* \\ a^*/x^*
\end{bmatrix} \begin{bmatrix} a_1/x_1 \\ a'_1/x'_1 \\ a^*_1/x^*_1 \\ a^*_1/x^*_1
\end{bmatrix} \begin{bmatrix} a_s/x_s \\ a'_s/x'_s \\ a^*_s/x^*_s \\ a^*_s/x^*_s
\end{bmatrix}
\]

\[
: \begin{bmatrix} a/x \\ a'/x' \\ a^*/x^* \\ a^*/x^*
\end{bmatrix} \begin{bmatrix} a_1/x_1 \\ a'_1/x'_1 \\ a^*_1/x^*_1 \\ a^*_1/x^*_1
\end{bmatrix} \begin{bmatrix} a_s/x_s \\ a'_s/x'_s \\ a^*_s/x^*_s \\ a^*_s/x^*_s
\end{bmatrix} \sim B_s[B_s][\bar{a}/\bar{x}] B_s'[\bar{a}'/\bar{x}']
\]

\[
= \prod \begin{bmatrix} y : B[a] \\ y' : B'[a']
\end{bmatrix} \begin{bmatrix} y_1 : B_1[a_1] \\ y'_1 : B'_1[a'_1]
\end{bmatrix} \begin{bmatrix} y_{1} : B_{s_1} [a_{s_1} a_{s_1}] y'_{1}
\end{bmatrix}
\]

\[
(y \sim_{B_s[\bar{a}]} y_1) \sim (y' \sim_{B'_s[\bar{a}']} y'_{1})
\]

Next, we recall that

\[
f^* : \prod\left( \begin{array}{c} x : A \\ x' : A' \\ x : x \sim_{A^*} x'
\end{array} \right) f x \sim_{B^*} f' x'
\]

\[
f'_1 : \prod\left( \begin{array}{c} x_1 : A_1 \\ x'_1 : A'_1 \\ x : x_1 \sim_{A_1^*} x'_1
\end{array} \right) f_1 x_1 \sim_{B_1^*} f'_1 x'_1
\]

It follows that

\[
f^* a a' a^* : B^* \begin{bmatrix} a/x \\ a'/x' \\ a^*/x^* \\ a^*/x^*
\end{bmatrix} \sim (f a)(f' a')
\]

\[
f'_1 a_1 a' a^*_1 : B_1^* \begin{bmatrix} a_1/x_1 \\ a'_1/x'_1 \\ a^*_1/x^*_1 \\ a^*_1/x^*_1
\end{bmatrix} \sim (f_1 a_1)(f'_1 a'_{1})
\]

Putting this together with (10), we get

\[
B_s^* \begin{bmatrix} a \\ a \\ a^* \\ a^*
\end{bmatrix} \begin{bmatrix} a_1 \\ a'_1 \\ a^*_1 \\ a^*_1
\end{bmatrix} \begin{bmatrix} a_s \\ a'_s \\ a^*_s \\ a^*_s
\end{bmatrix} \begin{bmatrix} f a \\ f' a' \\ f^* a^* \\ f^* a^* a^*
\end{bmatrix} \begin{bmatrix} f_1 a_1 \\ f'_1 a'_1 \\ f_1 a_1 a'_1 a^*_1
\end{bmatrix}
\]

\[
: (f a \sim_{B_s[\bar{a}] f_1 a_1}) \sim (f' a' \sim_{B'_s[\bar{a}'] f'_1 a'_{1})}
\]

This matches the expression in (9), concluding this case.
\[\Sigma\text{-congruence}\] Let us be given

\[
\begin{align*}
\Gamma &\vdash A : \ast & \Gamma, x : A &\vdash B : \ast \\
\Gamma &\vdash A_1 : \ast & \Gamma, x_1 : A_1 &\vdash B_1 : \ast \\
\Gamma &\vdash A_s : A \simeq A_1 & \Gamma, x, x_1 : A_1, x_s : x \sim_{A_s} x_1 &\vdash B_s : B \simeq B_1
\end{align*}
\]

We are to check that \((\Sigma^* [x, x_1, x_s] : A_s, B_s)^*\) has type

\[
(\Sigma^* [x, x_1, x_s] : A_s, B_s) \sim (\Sigma^* [x, x_1, x_s] : A_s, B_s)'
\]

\[
(\Sigma^* [x, x_1, x_s] : A_s, B_s) \sim \sim' (\Sigma^* [x, x_1, x_s] : A_s, B_s)'
\]

\[
\prod_{(p : \Sigma x : A, B, p') : (\Sigma x, x_1, x_s) : A_s, B_s)^*} \left( p \sim (\Sigma^*[x, x_1, x_s] : A_s, B_s, p_1) \right)
\]

Laying down this type on top of

\[
(\Sigma^* [x, x_1, x_s] : A_s, B_s)^*
\]

\[
\lambda \left( p : \Sigma x : A, B, p' : \Sigma x' : A', B', p' : p \sim \Sigma x : A, B, p' \right) \lambda \left( p_1 : \Sigma x_1 : A_1, B_1, p'_1 : \Sigma x'_1 : A'_1, B'_1, p'_1 : p_1 \sim \Sigma x : A, B, p' \right)
\]

\[
\sum^* \left[ a_s : a_1, p_1 p_1, a'_1, p'_1, a'_1 : a_s \sim (\Sigma^*[x, x_1, x_s] : A_s, B_s)^* a'_s \right] A_s^* \left[ a_s, p, p' \right] ( p_1, p'_1)
\]

\[
B_s^* \left[ \begin{array}{c}
\pi_1 p x \\
\pi_1 p' x'
\end{array} \right] \left[ \begin{array}{c}
\pi_1 p_1 x_1 \\
\pi_1 p_1' x'_1
\end{array} \right] \left[ \begin{array}{c}
\pi_2 p x_1 \\
\pi_2 p' x'_1
\end{array} \right] \left[ \begin{array}{c}
\pi_2 p_1 x_1 \\
\pi_2 p_1' x'_1
\end{array} \right]
\]

one can discern that the \(\Pi\)- and \(\lambda\)-binders have similar domains.

The terms will forever be united in a valid typing judgment if

\[
\sum^* \left[ a_s : a_1, p_1 p_1, a'_1, p'_1, a'_1 : a_s \sim (\Sigma^*[x, x_1, x_s] : A_s, B_s)^* a'_s \right] A_s^* \left[ a_s, p, p' \right] ( p_1, p'_1)
\]

has type

\[
(p \sim (\Sigma^*[x, x_1, x_s] : A_s, B_s, p_1) \sim (\Sigma^*[x', x'_1, x'_s] : A'_s, B'_s, p'_1)
\]

\[
\left( \sum^* \left[ x \begin{array}{c}
x \\
x_s
\end{array} \right] : A_s, B_s \right) p_1 p_1 \sim \left( \sum^* \left[ \begin{array}{c}
x \\
x_s
\end{array} \right] : A'_s, B'_s \right) p'_1
\]

\[
\sum_{a_s : a_1 p_1 \sim A_1, p_1} \sum_{a'_1 p'_1 \sim A'_1, p'_1} B_s^* \left[ \begin{array}{c}
\pi_1 p x \\
\pi_1 p' x'
\end{array} \right] \left[ \begin{array}{c}
\pi_1 p_1 x_1 \\
\pi_1 p_1' x'_1
\end{array} \right] \left[ \begin{array}{c}
\pi_2 p x_1 \\
\pi_2 p' x'_1
\end{array} \right] \left[ \begin{array}{c}
\pi_2 p_1 x_1 \\
\pi_2 p_1' x'_1
\end{array} \right]
\]

22
This is indeed the case, for by induction it so happens that

\[ A^* \left( \begin{array}{c} \pi_1 p \\ \pi_1 p' \\ \pi_1 p'' \end{array} \right) \left( \begin{array}{c} \pi_1 p_1 \\ \pi_1 p'_1 \\ \pi_1 p''_1 \end{array} \right) : (A^* \pi_1 p \pi_1 p_1) \sim (A^* \pi_1 p' \pi_1 p'_1) \]

and, for \( a^*_s : A^* \left( \begin{array}{c} \pi_1 p \\ \pi_1 p' \\ \pi_1 p'' \end{array} \right) \left( \begin{array}{c} \pi_1 p_1 \\ \pi_1 p'_1 \\ \pi_1 p''_1 \end{array} \right) \sim^* a^*_s, \)

\[ B^* \left[ \begin{array}{c} \pi_1 p / \pi_1 p' / \pi_1 p'' \\ \pi_1 p'/x \\ \pi_1 p''/x'' \end{array} \right] \left( \begin{array}{c} \pi_2 p \\ \pi_2 p' \\ \pi_2 p'' \end{array} \right) \sim B^* \left[ \begin{array}{c} \pi_1 p / \pi_1 p'/x \\ \pi_2 p / \pi_2 p' \end{array} \right] \pi_2 p_1 \pi_2 p_1 \]

This completes the case of \( \Sigma \)-congruence.

\( \sim \)-congruence The last case left standing is the \( \sim^* \)-constructor:

\[
\begin{align*}
\Gamma & \vdash A : * & \Gamma & \vdash B : * \\
\Gamma & \vdash A_1 : * & \Gamma & \vdash B_1 : * \\
\Gamma & \vdash A_s : A \sim A_1 & \Gamma & \vdash B_s : B \sim B_1 \\
\end{align*}
\]

\[
\begin{align*}
(\sim^* A_s B_s) & : (A \sim B) \sim (A_1 \sim B_1)
\end{align*}
\]

To get it down, we just need to force

\[
(\sim^* A_s B_s)^* = \lambda \left( e : A \equiv B, e' : A' \equiv B', e'' : \sim_{\sim^* A_s B_s} e'' \right) \lambda \left( e_1 : A_1 \equiv B_1, e'_1 : A'_1 \equiv B'_1, e''_1 : \sim_{\sim^* A_s B_s} e''_1 \right) \]

\[
\prod^* \left[ \begin{array}{c} a' \\ a^* \end{array} \right] : A^* \prod^* \left[ \begin{array}{c} a'_1 \\ a^*_1 \end{array} \right] : A^*_1 \prod^* \left[ \begin{array}{c} a'_2 \\ a^*_2 \end{array} \right] : A^*_2 \left( \begin{array}{c} a \\ a' \end{array} \right) \left( \begin{array}{c} a'_1 \\ a^*_1 \end{array} \right) \left( \begin{array}{c} a'_2 \\ a^*_2 \end{array} \right)
\]

\[
\prod^* \left[ \begin{array}{c} b' \\ b^* \end{array} \right] : B^* \prod^* \left[ \begin{array}{c} b'_1 \\ b^*_1 \end{array} \right] : B^*_1 \prod^* \left[ \begin{array}{c} b'_2 \\ b^*_2 \end{array} \right] : B^*_2 \left( \begin{array}{c} b \\ b' \end{array} \right) \left( \begin{array}{c} b'_1 \\ b^*_1 \end{array} \right) \left( \begin{array}{c} b'_2 \\ b^*_2 \end{array} \right)
\]

\[
\sim^* \left( e^* \left( \begin{array}{c} a' \\ a^* \end{array} \right) \left( \begin{array}{c} b \\ b' \end{array} \right) \left( \begin{array}{c} b'_1 \\ b^*_1 \end{array} \right) \right) \left( e''_1 \left( \begin{array}{c} a'_1 \\ a^*_1 \end{array} \right) \left( \begin{array}{c} b'_1 \\ b^*_1 \end{array} \right) \right) \left( e''_2 \left( \begin{array}{c} a'_2 \\ a^*_2 \end{array} \right) \left( \begin{array}{c} b'_2 \\ b^*_2 \end{array} \right) \right)
\]

into

\[
\begin{align*}
\sim^* A_s B_s & \sim_\sim (\sim_{\sim^* (A \equiv B)} \equiv (A_s \equiv B_s))^* \sim^* A'_s B'_s \\
& \equiv \sim^* A_s B_s \sim_{\sim^* (A \equiv B)} \equiv (A_s \equiv B_s) \sim^* A'_s B'_s \\
& = \prod \left( e : A \equiv B, e' : A' \equiv B', e'' : \sim_{\sim^* (A \equiv B)} e'' \right) \prod \left( e_1 : A_1 \equiv B_1, e'_1 : A'_1 \equiv B'_1, e''_1 : \sim_{\sim^* (A \equiv B)} e''_1 \right) \left( e \sim_{\sim^* A_s B_s} e_1 \right) \right) \left( e' \sim_{\sim^* A'_s B'_s} e'_1 \right)
\end{align*}
\]
The lambdas go into the pies quite easily, so we focus on

\[(e \sim_{\ast} A, B) \simeq (e' \sim_{\ast} A', B')\]  

\[
\begin{align*}
& \sim \left( \Pi : \Pi a A_1 \Pi a_1 : a \sim_{A,} a_1 \right) \\
& \sim \left( \Pi b : \Pi b_1 : B_1 \Pi b_1 : b \sim_{B,} b_1 \right) \\
& \sim \left( \Pi : \Pi a' A' \Pi a_1' : a' \sim_{A',} a_1' \right) \\
& \sim \left( \Pi b' : B' \Pi b_1' : B_1 \Pi b_1' : b' \sim_{B',} b_1' \right)
\end{align*}
\]

To inhabit this equivalence type, one needs to construct a sequence of equivalences which pairwise relate the domains of quantification in the sequence of \(\Pi\)-types on each side of the \(\sim\)-sign.

Close inspection will reveal that the six \(\Pi^*\)-constructors appearing in the unfolding of \((\sim_{\ast} A, B)\) do provide such a sequence. For example, by induction hypothesis, we have

\[A^*_s : \prod \left( \begin{array}{c}
\times : A \\
\times' : A' \\
\times'' : x \sim_{A,} x' \\
\end{array} \right) \prod \left( \begin{array}{c}
\times_1 : A_1 \\
\times_1' : A_1' \\
\times_1'' : x_1 \sim_{A_1} x_1' \\
\end{array} \right), \quad x \sim_{A,} x_1 \simeq x' \sim_{A'} x_1'
\]

whence we get the equivalence

\[A^*_s \left( \begin{array}{c}
a \\
a' \\
a_1 \\
a'_1 \\
\end{array} \right) \left( \begin{array}{c}
a_1 \\
a'_1 \\
\end{array} \right) : a \sim_{A,} a_1 \simeq a' \sim_{A'} a'_1
\]

relating the domains of the third \(\Pi\)s in the sequence.

(Similarly, \(B^*_s b b b' b_1 b_1' \simeq b' \sim_{B'} b_1\).

All that remains is to check that

\[
\sim \left( e^* \left( \begin{array}{c}
a \\
a' \\
a_1 \\
a'_1 \\
\end{array} \right) \left( \begin{array}{c}
b \\
b' \\
\end{array} \right) \right) \left( e'_1 \left( \begin{array}{c}
a_1 \\
a'_1 \\
\end{array} \right) \left( \begin{array}{c}
b_1 \\
b'_1 \\
\end{array} \right) \right)
\]

\[
: (a \sim_e b \simeq a_1 \sim_{e_1} b_1) \simeq (a' \sim_{e'} b' \simeq a_1' \sim_{e_1'} b_1')
\]

Indeed, this is attainable from

\[
e^* a a' a^* b b' b^* : (a \sim_e b) \simeq (a' \sim_{e'} b')
\]

\[
e'_1 a_1 a'_1 a_1 b'_1 b_1' b^* : (a_1 \sim_{e_1} b_1) \simeq (a_1' \sim_{e_1'} b_1')
\]

by feeding these terms into an application of the \(\sim\)-rule.

This completes the proof of the theorem. \(\square\)
Internalization at the term level

Now that we have determined the precise computational content of the extensionality theorem, we are ready to add a new operation into our language which will witness this fact internally.

Without further ado, we extend our system as follows:

\[
A, B, s, t, e, \alpha ::= \ast | x | \Pi x:A.B | \Sigma x:A.B | A \sim B | a \sim_c b \\
| \lambda x:A.t | st | (s, t) | \pi_1 t | \pi_2 t \\
| s =_A t | \Pi^*[x, x', x^*]: A.B | \Sigma^*[x, x', x^*]: A.B | z^*ee \\
| s \sim_A t | r(t) | Sub_{x:A}(B, \alpha) | Sub_{x:A}{B}(b, \alpha)
\]

The typing rules for the new term constructors are:

\[
\frac{\Gamma \vdash A: \ast \quad \Gamma \vdash a: A}{\Gamma \vdash a \sim_A a': \ast} \\
\frac{\Gamma \vdash A: \ast \quad \Gamma \vdash a: A}{\Gamma \vdash r(a): a \sim_A a} \\
\frac{\Gamma \vdash A: \ast \quad \Gamma, x:A \vdash B: \ast \quad \Gamma \vdash \alpha: a_1 \sim_A a_2}{\Gamma \vdash Sub_{x:A}(B, a^\ast): B[a_1/x] \sim B[a_2/x]} \\
\frac{\Gamma \vdash A: \ast \quad \Gamma, x:A \vdash B: \ast \quad \Gamma, x:A \vdash b: B \quad \Gamma \vdash \alpha: a_1 \sim_A a_2}{\Gamma \vdash sub_{x:A}{B}(b, \alpha): b[a_1/x] \sim_{Sub_{x:A}{B}(b, \alpha)} b[a_2/x]}
\]

The computation rules for these symbols will ensure that the following identities hold definitionally:

\[
A \sim B = A \sim_{r(\ast)} B \\
A \sim B = A \sim_\ast B \\
a \sim_A a' = a \sim_{r(A)} a' \\
r(A) = Sub_{\alpha\ast}(A, \ast^\ast) \quad o \notin \text{FV}(A) \\
r(a) = sub_{\alpha\ast}{A}(a, \ast^\ast) \quad o \notin \text{FV}(a)
\]
We begin with the definition of the extensional identity type.

\[ A \equiv B \quad \rightarrow \quad A \equiv B \]

\[
f \equiv \Pi_{x:A} B \quad g \quad \rightarrow \quad \prod_{a_1 : A} \prod_{a_2 : A} \left( f a_1 \sim \text{Sub}_{x:A}(B,a_1) \right) g a_2
\]

\[
p \equiv \Sigma_{x:A} B \quad q \quad \rightarrow \quad \sum_{a_1 : A} \pi_1 q \sim \text{Sub}_{x:A}(B,a_1) \pi_2 q
\]

\[
e_1 \equiv A \equiv B \quad e_2 \quad \rightarrow \quad \prod_{a_1 : A} \prod_{a_2 : A} \left( b_1 : B \quad b_2 : B \right) \left( a_1 \sim e_1 b_1 \right) \sim \left( a_2 \sim e_2 b_2 \right)
\]

For the reflexivity operator, we have the following stipulation.

In order to justify the relation

\[ r(t : T) = \text{sub}_{\alpha, T}(T)(t, \ast) \quad \left( \alpha \notin \text{FV}(t) \right) \]

we could just add this equation as a rewrite rule, by orienting it from left to right. This would be semantically valid — and notationally economical.

In practice however, we want the computation of terms to normal forms to proceed efficiently, and this requires performing “garbage collection” as quickly as possible. (This choice is typical of explicit substitution calculi.)

So we better orient these equations the other way:

\[
\text{Sub}_{v,T}(A, \gamma) \quad \rightarrow \quad r(A) \quad \left( \nu \notin \text{FV}(A) \right)
\]

\[
\text{sub}_{v,T}(A)(a, \gamma) \quad \rightarrow \quad r(a) \quad \left( \nu \notin \text{FV}(a) \right)
\]

Still, the reduction rules for \( r(\cdot) \) are otherwise exactly the same as the ones for \( \text{Sub} \) and \( \text{sub} \). We therefore choose to not write them out explicitly, but rather to explain how they may be recovered from the latter.

It is to be achieved thus:

1. Ignore the \( \text{Sub}/\text{sub} \)-rule having the side condition \( \nu \notin \text{FV}(A)/\nu \notin \text{FV}(a) \).

2. Replace every occurrence of \( \text{Sub}_{v,T}(\_ , \gamma) \) by \( r(\_ ) \). Similarly for \( \text{sub} \).

3. Replace every occurrence of \( \text{Sub}_{\xi, (\Sigma_{v,T})A}(\_ , \gamma, x^*) \) by \( \text{Sub}_{x,A}(\_ , x, \gamma, x^*) \). Similarly for \( \text{sub} \).

The upshot is this: if we switch off the “garbage collection” rules (which have the \( \nu \notin \text{FV}(A) \) side condition), then the substitution of \( \gamma \) into \( A \) will proceed to reduce exactly as \( r(A) \) would, and the result will be a normal form in which the reflexivity appears around the variables only.
As a consequence, we have that, writing

\[ \text{Sub}_{\nu : \Gamma}(A(\nu), \gamma) = A(\gamma) \]
\[ \text{sub}_{\nu : \Gamma}\{A(\nu)\}(a(\nu), \gamma) = a(\gamma) \]

it is always the case that the equalities

\[ r(A(x)) = A(r(x)) \]
\[ r(a(x)) = a(r(x)) \]

hold definitionally. This fact may be summarized by the statement:

**Degeneracies are preserved by all constructions of type theory.**

We shall now give reduction rules for the extensionality operators. In doing so, we make heavy use of some notational hubris. First of all —

**QUESTION:** Since we are working in a powerful metalevel type-setting system, why not rename some of the variables in our typing rules?

Next, should our dear reader ever encounter free variables of some term which stand out explicitly, as in \( B = B(x, y) \), this does not yet signify that all the free variables of \( B \) have been displayed, nor that the same variables in other types will be displayed. In particular, if we write

\[ v : G \vdash \Sigma x : A.B(v, x) \]

our intention is that, in general, \( v \) does occur in \( A \) as well as in \( B \). \(^3\)

Also, we will sometimes display implicit arguments to a function by writing them in the subscript.

\(^3\) The purpose of this convention is to anticipate the immanent substitution of the variables being displayed.
In the expressions below, every occurrence of \( \gamma \) is assumed to have type \( \gamma : \rho_1 \vdash \rho_2 \), unless denoted otherwise.

\[
\begin{align*}
\text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(A, \gamma) & \rightarrow r(A) \\
\text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\cdot, \gamma) & \rightarrow \gamma^* \\
\text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\alpha, \gamma) & \rightarrow \gamma \\
\text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\alpha, \gamma) & \rightarrow r(\alpha) \\
\text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\Pi : A.B(v, x), \gamma) & \rightarrow \Pi^* \left[ \begin{array}{c}
 x' \\
 x^* \\
 \end{array} \right] : \text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(A, \gamma). \text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\Sigma : A.B(v, x), \gamma) & \rightarrow \Sigma^* \left[ \begin{array}{c}
 x' \\
 x^* \\
 \end{array} \right] : \text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(A, \gamma). \text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\lambda x : A.B, \gamma) & \rightarrow \lambda x : A.B(a(v), b(v), \gamma) \\
\left( \text{Sub}_{\cdot \cdot \cdot \cdot \cdot \cdot}(\Pi \rightarrow \Sigma \rightarrow \lambda x : A.B : v, x) \right) \end{align*}
\]

Before proceeding to the congruence of type constructors, we stop and ask the reader to consider in earnest the relationship between these reduction rules and the defining clauses of the \( (\cdot)^* \) operator. For example, the equations which correspond to the last five rules are
\[(\lambda x:A.b)^* = \lambda x:A \lambda x':A' \ x' \sim A\ x'. b^* = \lambda x : x : A' \ x' : x \sim A, x'. b^*\]

\[(f a)^* = f^* a a' a^* = f\begin{pmatrix} a' \\ a^* \end{pmatrix}\]

\[(a, b)^* = (a^*, b^*)\]

\[(\pi_1 p)^* = \pi_1 p^*\]

\[(\pi_2 p)^* = \pi_2 p^*\]

From a cursory comparison of the two definitions, one can see that \textit{sub} acts much like an “engine” which processes the meta-level operation \((\cdot)^*\) internally.

Here is the precise statement of this relationship.

**Proposition.** Suppose we are given a sequence of derivations

\[
\begin{align*}
\Gamma & \vdash A : * \\
\Gamma, x : A & \vdash B(x) : * \\
\Gamma, x : A & \vdash b(x) : B(x) \\
\Gamma & \vdash a_{12} : a_1 \sim_A a_2
\end{align*}
\]

where

\[
\Gamma = (v_1, \ldots, v_n)
\]

Then it is the case that

\[
\begin{align*}
\text{Sub}_{x:A}(B, a_{12}) &= B^* \begin{bmatrix}
  v_1/v_1 & \cdots & v_n/v_n \\
  r(v_1)/v_1^* & \cdots & r(v_n)/v_n^*
\end{bmatrix} &
\begin{bmatrix}
  a_1/x \\
  a_2/x'
\end{bmatrix} \\
\text{sub}_{x:A}\{B\}(b, a_{12}) &= b^* \begin{bmatrix}
  v_1/v_1 & \cdots & v_n/v_n \\
  r(v_1)/v_1^* & \cdots & r(v_n)/v_n^*
\end{bmatrix} &
\begin{bmatrix}
  a_1/x \\
  a_2/x'
\end{bmatrix}
\end{align*}
\]

We illustrate this proposition with an example.

Suppose we start with

\[
f : A \to B \vdash \lambda x : A. \lambda y : B. f x : A \to B \to A
\]

\[
\vdash F^* : F \sim_{A\to B} F'
\]

We have
\[
\text{sub}_{f:A\to B}\{A \to B \to A\}(\lambda x:A.\lambda y:B.f\,x,F^*)
\]
\[
= \lambda (x:A \ x':A \ x^*:x \ x^\prime) \ \text{sub}_{\nu(\Sigma f:A\to B)A}\{B \to A\}(\lambda y:B.(\pi_1\nu)(\pi_2\nu),((F^*,x^*),y^*))
\]
\[
= \lambda (x:A \ x':A \ x^*:x \ x^\prime) \ \text{sub}_{\nu(\Sigma f:A\to B)A}\{A\}((\pi_1\nu)(\pi_2\nu),((F^*,x^*),y^*))
\]
\[
= \lambda (x:A \ x':A \ x^*:x \ x^\prime) \ \text{sub}_{\nu(\Sigma f:A\to B)A}\{B \to A\}(\pi_1\nu,((F^*,x^*),y^*))
\]
\[
= \lambda (x:A \ x':A \ x^*:x \ x^\prime) \ \text{sub}_{\nu(\Sigma f:A\to B)A}\{B \to A\}(\pi_1\nu,((F^*,x^*),y^*))
\]
\[
= \pi_1(F^*,x^*)
\]

The two \text{sub} terms can be further simplified as

\[
\text{sub}_{\nu(\Sigma f:A\to B)A}\{B \to A\}(\pi_1\nu,((F^*,x^*),y^*))
\]
\[
= \pi_1(F^*,x^*)
\]

Similarly,

\[
\text{sub}_{\nu(\Sigma f:A\to B)A}\{B \to A\}(\pi_2\nu,((F^*,x^*),y^*))
\]
\[
= \pi_2(F^*,x^*)
\]

30
We may substitute these simplified terms to continue the reduction

\[
\text{sub}_{f:A\rightarrow B}\{A \rightarrow B \rightarrow A\}(\lambda x:A.\lambda y:B.fx,F^*) \\
= \lambda(x:A, x ': A, x^*: x \equiv_A x') \lambda(y:B, y ': B, y^*: y \equiv_B y') \left( F^*(\pi_2\pi_1 v[((F, x), y)/v])((\pi_2\pi_1 v[((F', x'), y')/v]))x^* \right) \\
= \lambda(x:A, x ': A, x^*: x \equiv_A x') \lambda(y:B, y ': B, y^*: y \equiv_B y') \left( F^*(\pi_2\pi_1 x)((\pi_2\pi_1 ((F', x'), y'))x^* \right) \\
= \lambda(x:A, x ': A, x^*: x \equiv_A x') \lambda(y:B, y ': B, y^*: y \equiv_B y') \left( F^*x.x'x^* \right)
\]

The claim is now observed by comparing the result with

\[
(\lambda x:A\lambda y:B.fx)^* = \lambda(x:A, x ': A, x^*: x \equiv_A x') \lambda(y:B, y ': B, y^*: y \equiv_B y') \left( f^*x.x'x^* \right)
\]

The above proposition illustrates the recipe generating the reduction rules for sub: they are obtained from the corresponding (\*)-rules by replacing every occurrence of the operator on the right by a recursive call to sub, where the context (packaged into the "index" type) is updated accordingly so as to maintain veracity of the claim.

Due to the ostensibly mechanical character of the process, we shall indefinitely postpone the task of performing these replacements for all of the congruence constructors, until a dire need for such explicitation should arise.

For example, the \(\Pi^*\)-constructor is transliterated from

\[
(\Pi^*[x,x_1,x_s] : A_*, B_*)^* \\
= \lambda(f_1: \Pi x:A, B) \left( \lambda(f': \Pi x:A', B') \left( f_1^*(f: \Pi x:A, B) \pi_1 f'^* (f': \Pi x:A', B') \right) \right) \\
= \lambda(a_1 : A_*) \left( \lambda(a_1': A'_*) \left( \lambda(f_1: \Pi x:A, B) \left( \lambda(f': \Pi x:A', B') \left( f_1^*(f: \Pi x:A, B) \pi_1 f'^* (f': \Pi x:A', B') \right) \right) \right) \right)
\]

\[
B_*^* \left[ a^{1/1}_1/x_1 \right] ^* \left[ a^{1/1}_1/x_1 \right] ^* \left[ f^{a/1}_1 f'^{a'/1}_1 \right] ^* \left( f^{a/1}_1 f'^{a'/1}_1 \right) ^*
\]

\[31\]
into the reduction rule

\[
\text{sub}_{v;\Gamma} \{ \Pi x: A(v). B(v, x) \supseteq \Pi x_1: A_1(v). B_1(v, x_1) \} \{ \Pi x[x, x_1, x_*] : A_*(v). B_*(v, x, x_1, x_*), \gamma \}
\]

\[
\to \lambda \left( f^1: \Pi x^1, A(\mu_1). B(\mu_1, x^1) \right. \\
\left. f^2: \Pi x^2, A(\mu_2). B(\mu_2, x^2) \right) \\
\left. f^{12}: \sim (\Pi^1[x^1, x_1^2, x^{12}]: \text{sub}_{\Gamma}(A(v), \gamma)). \text{sub}_{\xi (\Sigma v: \Gamma) A(v)}(B(\pi_1\xi, \pi_2\xi, (\gamma, x^{12}))) \right) f^1 f^2
\]

\[
\lambda \left( f^1: \Pi x^1, A_1(\mu_1). B_1(\mu_1, x^1) \right. \\
\left. f^2: \Pi x^2, A_1(\mu_2). B_1(\mu_2, x^2) \right) \\
\left. f^{12}: \sim (\Pi^1[x^1, x_1^2, x^{12}]: \text{sub}_{\Gamma}(A_1(v), \gamma)). \text{sub}_{\xi (\Sigma v: \Gamma) A_1(v)}(B_1(\pi_1\xi, \pi_2\xi, (\gamma, x^{12}))) \right) f^1 f^2
\]

\[
\Pi^1 \left[ \begin{array}{c}
\{ a_1^1 \\
\{ a_1^{12} \}
\end{array} \right] : \text{sub}_{\Gamma}(A(v), \gamma) \Pi^1 \left[ \begin{array}{c}
\{ a_2^1 \\
\{ a_2^{12} \}
\end{array} \right] : \text{sub}_{\Gamma}(A_1(v), \gamma)
\]

\[
\left( \text{sub}_{\xi (\Sigma v: \Gamma) A(v)}(B(\pi_11\xi, \pi_21\xi)) \supseteq B_1(\pi_11\xi, \pi_21\xi) \right) \left( B(\pi_11\xi, \pi_21\xi, \pi_21\xi, (\gamma, a^{12}, a_1^{12}, a_2^{12})) \right) \left( f^1 a_1^1 f^2 a_2^1 f^{12} a_1^{12} a_2^{12} \right)
\]

Against the interests of entropy, we do not repeat this exercise for the other congruence constructors.

The remaining piece is to witness extensionality of the extensionality operators themselves. That is, we are to specify how sub acts on Sub and on itself. Let us assume the following setup.

\[
v : \Gamma \vdash A(v) : *
\]

\[
v : \Gamma \vdash a_*(v) : a_1(v) \equiv A(v) a_2(v)
\]

\[
v : \Gamma, x : A(v) \vdash B(v, x)
\]

\[
v : \Gamma, x : A(v) \vdash b(v, x)
\]

We define

\[
\text{sub}_{\Gamma}{\{ B(v, a_1(v)) \equiv B(v, a_2(v)) \}} \left( \text{sub}_{\chi}(A(v)) B(v, x), a_*(v), \gamma \right)
\]

\[
\to \text{sub}_{\phi (\Sigma v: \Gamma) a_1(v) \equiv A(v) a_2(v) B(\pi_1\phi, a_1(\pi_1\phi)) \equiv B(\pi_1\phi, a_2(\pi_1\phi))}
\]

\[
\text{sub}_{\phi (\Sigma v: \Gamma) a_1(v) \equiv A(v) a_2(v) B(\pi_1\phi, a_1(\pi_1\phi)) \equiv B(\pi_1\phi, a_2(\pi_1\phi))}
\]

\[
\text{sub}_{\phi (\Sigma v: \Gamma) a_1(v) \equiv A(v) a_2(v) B(\pi_1\phi, a_1(\pi_1\phi)) \equiv B(\pi_1\phi, a_2(\pi_1\phi))}
\]

\[
\text{sub}_{\phi (\Sigma v: \Gamma) a_1(v) \equiv A(v) a_2(v) B(\pi_1\phi, a_1(\pi_1\phi)) \equiv B(\pi_1\phi, a_2(\pi_1\phi))}
\]

\[
\text{sub}_{\phi (\Sigma v: \Gamma) a_1(v) \equiv A(v) a_2(v) B(\pi_1\phi, a_1(\pi_1\phi)) \equiv B(\pi_1\phi, a_2(\pi_1\phi))}
\]

32
These rules can be understood as the higher-dimensional analogue of composition of substitutions:

\[
B(v, x)[a_*(v)/x][\gamma/v] = Bp[(\gamma, a^*(\gamma))/p]
\]
\[
b(v, x)[a_*(v)/x][\gamma/v] = bp[(\gamma, a^*(\gamma))/p]
\]

This situation is displayed in the diagram appearing on the last page.

This completes our description of the computation rules for extensionality operators.
References

Altenkirch, Thorsten, Conor McBride and Wouter Swierstra (2007). Observational equality, now!, in: Aaron Stump and Hongwei Xi (eds.), PLPV, ACM, pp. 57–68.

Coquand, Thierry (2011). Equality and dependent type theory. http://www.cse.chalmers.se/~coquand/equality.pdf

Gandy, Robin O. (1956). On the axiom of extensionality–part I, J. Symb. Log. 21(1), pp. 36–48.

Gonthier, Georges, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi and Laurent Théry (2013). A machine-checked proof of the odd order theorem, in: Sandrine Blazy, Christine Paulin-Mohring and David Pichardie (eds.), ITP, Lecture Notes in Computer Science 7998, Springer, pp. 163–179.

Hofmann, Martin and Thomas Streicher (1996). The groupoid interpretation of type theory, In Venice Festschrift, Oxford University Press, pp. 83–111.

Licata, Daniel R. and Robert Harper (2012). Canonicity for 2-dimensional type theory, in: John Field and Michael Hicks (eds.), POPL, ACM, pp. 337–348.

Martin-Löf, Per (1984). Intuitionistic Type Theory, Bibliopolis, Naples.

Tait, William W. (1995). Extensional equality in the classical theory of types, in: Werner DePauli-Schimanovich, Eckehart Khler and Friedrich Stadler (eds.), The Foundational Debate, Vienna Circle Institute Yearbook [1995] 3, Springer Netherlands, pp. 219–234.

The Univalent Foundations Program, I.A.S. (n.d.). Homotopy Type Theory: Univalent Foundations of Mathematics, Univalent Foundations.

Voevodsky, Vladimir (2006). A very short note on the homotopy λ-calculus, http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf
Appendix: Stratification of $\lambda e$

The stratified version of $\lambda e$ is obtained by the following recipe.

1. The symbol $\ast$ is replaced by an infinite collection of constants

$$\{\ast_n \mid n \in \omega\}$$

2. The typing rule $\frac{}{\ast : \ast}$ is replaced by the rule scheme (one rule for each number $n$)

$$\frac{}{\ast : \ast_{n+1}}$$

3. A new rule is introduced:

$$\frac{\Gamma \vdash A : \ast_n}{\Gamma \vdash A : \ast_{n+1}}$$

4. The formation rules for $\Pi$ and $\Sigma$ are replaced by

$$\frac{\Gamma \vdash A : \ast_n \quad \Gamma, x : A \vdash B : \ast_m}{\Gamma \vdash \Pi x : A.B : \ast_{\max(m,n)}}$$

$$\frac{\Gamma \vdash \Sigma x : A.B : \ast_{\max(m,n)}}{\Gamma \vdash \Sigma x : A.B : \ast_{\max(m,n)}}$$

5. The $\ast^*$-formation rule is replaced by

$$\frac{\{A : \ast_n\} \quad \{B : \ast_n\} \quad e : A \cong B}{\sim e : A \rightarrow B \rightarrow \ast_n}$$

6. In all other rules, the $\ast$ symbol is replaced $\ast_n$.

By looking closely, one can check the following:

1. If $\Gamma \vdash A : \ast_n$ and $A \rightarrow B$, then $\Gamma \vdash B : \ast_n$.

2. If $\Gamma \vdash M : A$ and $\Gamma \vdash A : \ast_n$, then $\Gamma \vdash M \sim_{A^*} M' : \ast_n$.

From these two facts, it follows that the proofs given above for $\lambda e$ carry over for the stratified versions when $\ast$ is replaced by $\ast_n$ as appropriate.
