A Type Theory with Definitional Univalence

Sebastian Reichelt

January 22, 2023

(Warning: WIP, still contains important gaps)

1 Abstract

We present a way to enrich an intensional type theory (containing Π and Σ types and universes, but lacking induction for the moment) with identity types that *definitionally* match the behavior of identities in Homotopy Type Theory (HoTT) [3]. Specifically, for every type A that is an application of a primitive type former, and instances $a, b : A$, the expression $[a = b]$ reduces according to the extensionality principle for that type former as known from HoTT – in particular, univalence if A is a universe. We construct a function that implements dependent elimination for identities (i.e. the J rule, known as path induction in HoTT).

Thus equality has the same properties as in HoTT except that univalence (and function extensionality etc.) holds definitionally, preserving computation – and improving even over cubical type theory [2] in this regard, where univalence is a theorem but not a definitional equality.[1] A practical advantage is that arbitrarily complex algebraic structures can trivially be transported along type equivalences in a theorem prover implementing this type theory, while preserving all properties that can be defined on them. Since the resulting transport functions are reducible to a form that avoids univalence or even function extensionality, those reduced terms can potentially even be exported to theorem provers *not* implementing this type theory.

The single trick that greatly simplifies the construction of this type theory is to replace lambda abstractions with the S, K, and I combinators [4]. Crucially, these combinators as well as all type formers and introduction and elimination rules are well-typed (dependent) functions. Therefore we can assume that the only terms in the theory are primitive constants and well-typed applications thereof (even though a practical implementation may work with lambda abstractions instead or in addition). This enables us to define the behavior of identities by induction over all primitive functions.

2 Base Theory

The type theory we define is based on Martin-Löf dependent type theory with universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$$

The only primitive terms are the constants that we introduce below, and function application, which we write in the form "$f(x)$," with "$f(x, y, \dots)$" as a shorthand for "$f(x)(y)\dots$" All other terms that are usually regarded as primitive, such as type formers and lambda abstractions, are actually applications of primitive constants, as described in the following subsections.

A consequence is that many definitions become recursive. In particular, typing judgments often refer to terms that are defined later. This should not be seen as a problem because the definition of the theory can be separated into four distinct phases:

1. First, all primitive constants are introduced as untyped terms.
2. Then reduction rules are defined on these untyped terms.

[1]The idea seems similar to "higher observational type theory" [1], which at the time of this writing is work in progress.

3. Next, the primitive constants are enriched with typing judgments.
4. Finally, we can verify the type correctness of every individual reduction rule, taking reduction within types into account.

The caveat is that the reduction rules would be completely unintelligible without type information. Therefore we choose the lesser evil of frequently referring to terms that are defined later in the article, sometimes explicitly but often implicitly.

2.1 Primitive Type Formers

We assume that the theory contains the following type formers. These are quite minimalist in order to demonstrate our basic approach; more type formers such as W types can be added if an appropriate definition of equality exists for the types that they produce.

- There are three primitive types $\mathbf{0}, \mathbf{1}, \mathbf{2} : \mathcal{U}_n$ in any universe $\mathcal{U}_n$.
- If $\mathcal{U}_m$ and $\mathcal{U}_n$ are universes, $A : \mathcal{U}_m$, and $P : A \to \mathcal{U}_n$, there is a primitive type $\Pi(P) : \mathcal{U}_{\max(m,n)}$, which we often write as $\prod_{a:A} P(a)$. As usual, for $B : \mathcal{U}_n$ we define $[A \to B] :\equiv \prod_{a:A} B$.
- Similarly, we define $\Sigma(P) : \mathcal{U}_{\max(m,n)}$, often written as $\sum_{a:A} P(a)$, and $[A \times B] :\equiv \sum_{a:A} B$.
- $\mathcal{U}_n : \mathcal{U}_{n+1}$ is considered a type former as well.

We can regard Π and Σ as (dependent) functions

$$\Pi, \Sigma : \prod_{A:\mathcal{U}_m} [[A \to \mathcal{U}_n] \to \mathcal{U}_{\max(m,n)}],$$

so that all type formers are just constants with specific types.

2.2 Other Primitive Constants

We define all introduction and elimination rules, including the three combinators which are essentially the introduction rules for Π types, as primitive constants with appropriate types.

- $\mathsf{elim}_{\mathbf{0},A} : \mathbf{0} \to A$ for any type A.
- $\star : \mathbf{1}$.
- $0, 1 : \mathbf{2}$.
- $\mathsf{elim}_{\mathbf{2},A} : A \to A \to [\mathbf{2} \to A]$ for any type A.
- $\mathsf{I}_A : A \to A$ for any type A.
- $\mathsf{K}_{AB} : B \to [A \to B]$ for all types A and B.
- $\mathsf{S}_{PQ} : \left[\prod_{a:A} \prod_{b:P(a)} Q(a,b)\right] \to \prod_{f:\Pi(P)} \prod_{a:A} Q(a, f(a))$

  for $A : \mathcal{U}_l$, $P : A \to \mathcal{U}_m$, and $Q : \prod_{a:A}[P(a) \to \mathcal{U}_n]$.

  I, K, and S define how to interpret function definitions of the form

  $$\begin{aligned} f : A &\to B_a \\ a &\mapsto b_a \end{aligned}$$

  throughout this article.

  – If b_a is a, then $f :\equiv \mathsf{I}_A$.

  – If b_a is a constant $b : B$, i.e. a does not appear in b_a, then $f :\equiv \mathsf{K}_{AB}(b)$.

  – If b_a is a function application $g_a(c_a)$ with $c_a : C_a$ and $g_a : \prod_{c:C_a} B_{ac}$, then $f :\equiv \mathsf{S}_{PQ}(g, c)$ with

$$\begin{array}{llll} P : A \to \mathcal{U}_m & Q : A \to C_a \to \mathcal{U}_n & g : A \to \prod_{c:C_a} B_{ac} & c : A \to C_a \\ \quad a \mapsto C_a, & \quad (a,c) \mapsto B_{ac}, & \quad a \mapsto g_a, & \quad a \mapsto c_a. \end{array}$$

Note that in the definition of Q, we already extended the function notation to nested functions, which we write as functions with multiple arguments, each of which may depend on the previous.

- $\mathsf{intro}_{\Sigma(P)} : \prod_{a:A}[P(a) \to \Sigma(P)]$ for $A : \mathcal{U}_m$ and $P : A \to \mathcal{U}_n$.
- $\mathsf{fst}_{\Sigma(P)} : \Sigma(P) \to A$ for $A : \mathcal{U}_m$ and $P : A \to \mathcal{U}_n$.
- $\mathsf{snd}_{\Sigma(P)} : \prod_{p:\Sigma(P)} P(\mathsf{fst}_{\Sigma(P)}(p))$ for $A : \mathcal{U}_m$ and $P : A \to \mathcal{U}_n$.

2.3 Reduction Rules

The reduction rules should be entirely unsurprising.

- $\mathsf{elim}_{\mathbf{2},A}(a, b, 0) :\equiv a$
- $\mathsf{elim}_{\mathbf{2},A}(a, b, 1) :\equiv b$
- $\mathsf{I}_A(a) :\equiv a$
- $\mathsf{K}_{AB}(b, a) :\equiv b$
- $\mathsf{S}_{PQ}(g, f, a) :\equiv g(a, f(a))$
- $\mathsf{fst}_{\Sigma(P)}(\mathsf{intro}_{\Sigma(P)}(a, b)) :\equiv a$
- $\mathsf{snd}_{\Sigma(P)}(\mathsf{intro}_{\Sigma(P)}(a, b)) :\equiv b$

3 Identity Types

For each $T : \mathcal{U}_n$ and $x, y : T$ we define a new type $[x = y] : \mathcal{U}_n$, technically by introducing a new type former but with reduction rules that provide a ‘definition’ for it by case analysis on T.

- For $a, b : \mathbf{1}$, $[a = b] :\equiv \mathbf{1}$.
- For $a, b : \mathbf{2}$, $[a = b] :\equiv \mathsf{elim}_{\mathbf{2},[a=b]}(\mathsf{elim}_{\mathbf{2},[a=b]}(\mathbf{1}, \mathbf{0}, a), \mathsf{elim}_{\mathbf{2},[a=b]}(\mathbf{0}, \mathbf{1}, a), b)$.
- For $A : U_m$, $P : A \to \mathcal{U}_n$, and $f, g : \Pi(P)$, $[f = g] :\equiv \prod_{a:A} [f(a) = g(a)]$.
- For $A : U_m$, $P : A \to \mathcal{U}_n$, and $p, q : \Sigma(P)$, $[p = q] :\equiv \sum_{e:\mathsf{fst}(p)=\mathsf{fst}(q)}[\mathsf{snd}(p) =^{\mathsf{ap}_P(e)} \mathsf{snd}(q)]$, where ap and the notation “$=^f$” are defined below.
- For $A, B : \mathcal{U}_n$ we define $[A = B] :\equiv [A \simeq B]$, where “$\simeq$” denotes the type of equivalences as constructed in Homotopy Type Theory (HoTT).

  For $e : A = B$, $a : A$, and $b : B$, we define $[a =^e b] :\equiv [e(a) = b]$, analogously to HoTT.

refl, composition, and inverse are defined by induction in the obvious way.

3.1 Functoriality

For $P : A \to \mathcal{U}_n$, $f : \Pi(P)$, and $a, b : A$, we define a function

$$\mathsf{ap}_f : \prod_{e:a=b} [f(a) =^{\mathsf{ap}_P(e)} f(b)]$$

by induction over all primitive functions f (including type formers) and their applications, i.e. we introduce this function as a primitive constant but also define reduction rules for all possible cases.

The type of ap refers to ap itself, but this is unproblematic because reduction rules are not affected by types. For example, if P is constant, so we have $f : A \to B$ for some B, then $\mathsf{ap}_P(e) \equiv \mathsf{refl}_B$ as defined below, and the typing judgment of ap_f reduces to

$$\mathsf{ap}_f : a = b \to f(a) = f(b).$$

Note that when f maps to a type, ap_f maps to an equivalence between types; for now we will only specify the forward direction in such cases, and leave the remaining data as an exercise for the reader.

- $\mathsf{ap}_{\Pi_A} : P = P' \to \Pi(P) = \Pi(P')$, for $P, P' : A \to \mathcal{U}_n$, is defined by

$$\begin{aligned} \mathsf{ap}_{\Pi_A} : \left[\prod_{a:A}[P(a) \simeq P'(a)]\right] &\to \Pi(P) \to A \to P'(a) \\ (e_P, f, a) &\mapsto e_P(a, f(a)). \end{aligned}$$

- $\mathsf{ap}_{\Pi} : \prod_{e_A : A = A'}[\Pi_A =^{\mathsf{ap}_Q(e_A)} \Pi_{A'}]$, where

$$\begin{aligned} Q : \mathcal{U}_m &\to U_{\max(m,n)+1} \\ A &\mapsto [[A \to \mathcal{U}_n] \to \mathcal{U}_{\max(m,n)}] \end{aligned}$$

  so that

$$\begin{aligned} \mathsf{ap}_Q : [A \simeq A'] \to [[A \to \mathcal{U}_n] \to \mathcal{U}_{\max(m,n)}] &\to [A' \to \mathcal{U}_n] \to \mathcal{U}_{\max(m,n)} \\ (e_A, R, P') &\mapsto R(P' \circ e_A), \end{aligned}$$

  i.e. $\mathsf{ap}_{\Pi} : \prod_{e_A : A \simeq A'} \prod_{P' : A' \to \mathcal{U}_n} [\Pi(P' \circ e_A) \simeq \Pi(P')]$, is defined by

$$\begin{aligned} \mathsf{ap}_{\Pi} : [A \simeq A'] \to [A' \to \mathcal{U}_n] \to \Pi(P' \circ e_A) &\to A' \to P'(a') \\ (e_A, P', f, a') &\mapsto \mathsf{ap}_{P'}(\mathsf{rightInv}_{e_A}(a'), f(e_A^{-1}(a'))), \end{aligned}$$

  where $\mathsf{rightInv}$ denotes the appropriate identity in the definition of type equivalence.

- $\mathsf{ap}_{\Sigma_A} : P = P' \to \Sigma(P) = \Sigma(P')$, for $P, P' : A \to \mathcal{U}_n$, is defined by

$$\begin{aligned} \mathsf{ap}_{\Sigma_A} : \left[\prod_{a:A}[P(a) \simeq P'(a)]\right] &\to \Sigma(P) \to \Sigma(P') \\ (e_P, p) &\mapsto (a, e_P(\mathsf{fst}(p), \mathsf{snd}(p))). \end{aligned}$$

- $\mathsf{ap}_{\Sigma} : \prod_{e_A : A = A'}[\Sigma_A =^{\mathsf{ap}_Q(e_A)} \Sigma_{A'}]$, where

$$\begin{aligned} Q : \mathcal{U}_m &\to U_{\max(m,n)+1} \\ A &\mapsto [[A \to \mathcal{U}_n] \to \mathcal{U}_{\max(m,n)}] \end{aligned}$$

  as above, is defined by

$$\begin{aligned} \mathsf{ap}_{\Sigma} : [A \simeq A'] \to [A' \to \mathcal{U}_n] \to \Sigma(P' \circ e_A) &\to \Sigma(P') \\ (e_A, P', p) &\mapsto (e_A(\mathsf{fst}(p)), \mathsf{snd}(p)). \end{aligned}$$

- We also need to consider the type former for the identity type itself, i.e.

$$\begin{aligned} \mathbf{Id}_A : A \to A &\to \mathcal{U}_n \\ (a, b) &\mapsto [a = b]. \end{aligned}$$

  We define its reduction rules generically, but we will need to verify that these match the corresponding rule when A is an application of a primitive type former.

  $\mathsf{ap}_{\mathbf{Id}_A(a)} : b = b' \to [a = b] = [a = b']$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathbf{Id}_A(a)} : b = b' \to a = b &\to a = b' \\ (e_b, f) &\mapsto e_b \circ f. \end{aligned}$$

- $\mathsf{ap}_{\mathbf{Id}_A} : a = a' \to [a = b] = [a' = b]$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathbf{Id}_A} : a = a' \to a = b &\to a' = b \\ (e_a, f) &\mapsto f \circ e_a^{-1}. \end{aligned}$$

- $\mathsf{ap}_{\mathbf{Id}} : \prod_{e_A : A = A'} [\mathbf{Id}_A =^{\mathsf{ap}_P(e_A)} \mathbf{Id}_{A'}]$, where

$$\begin{aligned} P : \mathcal{U}_n &\to U_{n+1} \\ A &\mapsto [A \to A \to \mathcal{U}_n] \end{aligned}$$

so that

$$\begin{aligned} \mathsf{ap}_P : [A \simeq A'] \to [A \to A \to \mathcal{U}_n] &\to A' \to A' \to \mathcal{U}_n \\ (e_A, R, a', b') &\mapsto R(e_A^{-1}(a'), e_A^{-1}(b')), \end{aligned}$$

is defined by

$$\begin{aligned} \mathsf{ap}_{\mathbf{Id}} : [A \simeq A'] \to e_A^{-1}(a') = e_A^{-1}(b') &\to a' = b' \\ (e_A, f) &\mapsto (\mathsf{rightInv}_{e_A}(b'))^{-1} \circ \mathsf{ap}_{e_A}(f) \circ \mathsf{rightInv}_{e_A}(a'). \end{aligned}$$

- $\mathsf{ap}_{\mathsf{I}_A} : a = a' \to \mathsf{I}_A(a) = \mathsf{I}_A(a')$ is $\mathsf{I}_{[a=a']}$.
- $\mathsf{ap}_{\mathsf{I}} : \prod_{e_A : A = A'} [\mathsf{I}_A =^{\mathsf{ap}_P(e_A)} \mathsf{I}'_A]$, where

$$\begin{aligned} P : \mathcal{U}_n &\to U_{n+1} \\ A &\mapsto [A \to A] \end{aligned}$$

so that

$$\begin{aligned} \mathsf{ap}_P : [A \simeq A'] \to [A \to A] &\to A' \to A' \\ (e_A, f, a') &\mapsto e_A(f(e_A^{-1}(a'))), \end{aligned}$$

is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{I}} : [A \simeq A'] \to A' &\to e_A(e_A^{-1}(a')) = a' \\ (e_A, a') &\mapsto \mathsf{rightInv}_{e_A}(a'). \end{aligned}$$

- $\mathsf{ap}_{\mathsf{K}_{AB}(b)} : a = a' \to \mathsf{K}_{AB}(b, a) = \mathsf{K}_{AB}(b, a')$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{K}_{AB}(b)} : a = a' &\to b = b \\ e_a &\mapsto \mathsf{refl}_b. \end{aligned}$$

- $\mathsf{ap}_{\mathsf{K}_{AB}} : b = b' \to \mathsf{K}_{AB}(b) = \mathsf{K}_{AB}(b')$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{K}_{AB}} : b = b' &\to A \to b = b' \\ (e_b, a) &\mapsto e_b. \end{aligned}$$

- $\mathsf{ap}_{\mathsf{K}_A} : \prod_{e_B : B = B'} [\mathsf{K}_{AB} =^{\mathsf{ap}_P(e_B)} \mathsf{K}_{AB'}]$, where

$$\begin{aligned} P : \mathcal{U}_n &\to \mathcal{U}_{\max(m,n)+1} \\ B &\mapsto [B \to [A \to B]] \end{aligned}$$

so that

$$\begin{aligned} \mathsf{ap}_P : [B \simeq B'] \to [B \to [A \to B]] &\to B' \to A \to B' \\ (e_B, f, b', a) &\mapsto e_B(f(e_B^{-1}(b'), a)), \end{aligned}$$

is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{K}_A} : [B \simeq B'] \to B' \to A &\to e_B(e_B^{-1}(b')) = b' \\ (e_B, b', a) &\mapsto \mathsf{rightInv}_{e_B}(b') \end{aligned}$$

- $\mathsf{ap}_{\mathsf{K}} : \prod_{e_A:A=A'}[\mathsf{K}_A =^{\mathsf{ap}_P(e_A)} \mathsf{K}_{A'}]$, where

$$\begin{aligned} P : \mathcal{U}_n &\to \mathcal{U}_{\max(m,n)+1} \\ A &\mapsto \prod_{B:\mathcal{U}_n}[B \to [A \to B]] \end{aligned}$$

  so that

$$\begin{aligned} \mathsf{ap}_P : [A \simeq A'] \to \left[\prod_{B:\mathcal{U}_n}[B \to [A \to B]]\right] &\to \mathcal{U}_n \to B \to A' \to B \\ (e_A, f, B, b, a') &\mapsto f(b, e_A^{-1}(a')), \end{aligned}$$

  is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{K}} : [A \simeq A'] \to \mathcal{U}_n \to B \to A' &\to b = b \\ (e_A, B, b, a') &\mapsto \mathsf{refl}_b \end{aligned}$$

- $\mathsf{ap}_{\mathsf{S}_{PQ}(g,f)} : a = a' \to \mathsf{S}_{PQ}(g,f,a) = \mathsf{S}_{PQ}(g,f,a')$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{S}_{PQ}(g,f)} : a = a' &\to g(a, f(a)) = g(a', f(a')) \\ e_a &\mapsto \mathsf{ap}_{g(a')}(\mathsf{ap}_f(e_a)) \circ \mathsf{ap}_g(e_a, f(a)) \\ &\quad = \mathsf{ap}_g(e_a, f(a')) \circ \mathsf{ap}_{g(a)}(\mathsf{ap}_f(e_a)). \end{aligned}$$

- $\mathsf{ap}_{\mathsf{S}_{PQ}(g)} : f = f' \to \mathsf{S}_{PQ}(g,f) = \mathsf{S}_{PQ}(g,f')$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{S}_{PQ}(g)} : f = f' \to A &\to g(a, f(a)) = g(a, f'(a)) \\ (e_f, a) &\mapsto \mathsf{ap}_{g(a)}(e_f(a)). \end{aligned}$$

- $\mathsf{ap}_{\mathsf{S}_{PQ}} : g = g' \to \mathsf{S}_{PQ}(g) = \mathsf{S}_{PQ}(g')$ is defined by

$$\begin{aligned} \mathsf{ap}_{\mathsf{S}_{PQ}} : g = g' \to \Pi(P) \to A &\to g(a, f(a)) = g'(a, f(a)) \\ (e_g, f, a) &\mapsto e_g(a, f(a)). \end{aligned}$$

- $\mathsf{ap}_{\mathsf{S}_P} : \prod_{e_Q:Q=Q'}[\mathsf{S}_{PQ} =^{\mathsf{ap}_R(e_Q)} \mathsf{S}_{PQ'}]$, where

$$\begin{aligned} R : \mathcal{U}_{\max(l,m,n+1)} &\to \mathcal{U}_{\max(l,m,n)+1} \\ Q &\mapsto \left[\left[\prod_{a:A}\prod_{b:P(a)} Q(a,b)\right] \to \prod_{f:\Pi(P)}\prod_{a:A} Q(a, f(a))\right] \end{aligned}$$

  so that

$$\begin{aligned} &\mathsf{ap}_R : \left[\prod_{a:A}\prod_{b:P(a)}[Q(a,b) \simeq Q'(a,b)]\right] \to \\ &\left[\left[\prod_{a:A}\prod_{b:P(a)} Q(a,b)\right] \to \prod_{f:\Pi(P)}\prod_{a:A} Q(a, f(a))\right] \to \\ &\left[\prod_{a:A}\prod_{b:P(a)} Q'(a,b)\right] \to \Pi(P) \to A \to Q'(a, f(a)) \\ &(e_Q, h, g', f, a) \mapsto e_Q(a, f(a), h(e_Q^{-1}(g'), f, a)) \end{aligned}$$

(with $e_Q^{-1}(g')$ defined in the obvious way), is defined by

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{S}_P} : \left[\prod_{a:A}\prod_{b:P(a)}[Q(a,b) \simeq Q'(a,b)]\right] \to \\
&\left[\prod_{a:A}\prod_{b:P(a)} Q'(a,b)\right] \to \Pi(P) \to A \to e_Q(a, f(a), (e_Q(a, f(a)))^{-1}(g'(a, f(a)))) = g'(a, f(a)) \\
&(e_Q, g', f, a) \mapsto \mathsf{rightInv}_{e_Q(a,f(a))}(g'(a, f(a))) \\
&\qquad \equiv \mathsf{ap}_{\mathsf{S}_{PQ'}}(\mathsf{rightInv}_{e_Q}(g'), f, a)
\end{aligned}$$

(with $\mathsf{rightInv}_{e_Q}(g')$ defined in the obvious way).

- $\mathsf{ap}_{\mathsf{S}_A} : \prod_{e_P:P=P'}[\mathsf{S}_P =^{\mathsf{ap}_R(e_P)} \mathsf{S}_{P'}]$, where

$$\begin{aligned}
&R : \mathcal{U}_{\max(l,m+1)} \to \mathcal{U}_{\max(l,m,n)+1} \\
&P \mapsto \prod_{Q:\prod_{a:A}[P(a)\to\mathcal{U}_n]} \left[\left[\prod_{a:A}\prod_{b:P(a)} Q(a,b)\right] \to \prod_{f:\Pi(P)}\prod_{a:A} Q(a, f(a))\right],
\end{aligned}$$

is defined by

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{S}_A} : \left[\prod_{a:A}[P(a) \simeq P'(a)]\right] \to \left[\prod_{a:A}[P'(a) \to \mathcal{U}_n]\right] \to \\
&\left[\prod_{a:A}\prod_{b':P'(a)} Q'(a,b')\right] \to \Pi(P') \to A \to g'(a, e_P(a, ((e_P(a))^{-1})(f'(a)))) = g'(a, f'(a)) \\
&(e_P, Q', g', f', a) \mapsto \mathsf{ap}_{g'(a)}(\mathsf{rightInv}_{e_P(a)}(f'(a))) \\
&\qquad \equiv \mathsf{ap}_{\mathsf{S}_{P'Q'(g')}}(\mathsf{rightInv}_{e_P}(f'), a)
\end{aligned}$$

(with $\mathsf{rightInv}_{e_P}(f')$ defined in the obvious way).

- $\mathsf{ap}_{\mathsf{S}} : \prod_{e_A:A=A'}[\mathsf{S}_A =^{\mathsf{ap}_R(e_A)} \mathsf{S}_{A'}]$, where

$$\begin{aligned}
&R : \mathcal{U}_l \to \mathcal{U}_{\max(l,m,n)+1} \\
&A \mapsto \prod_{P:A\to\mathcal{U}_m}\ \prod_{Q:\prod_{a:A}[P(a)\to\mathcal{U}_n]} \left[\left[\prod_{a:A}\prod_{b:P(a)} Q(a,b)\right] \to \prod_{f:\Pi(P)}\prod_{a:A} Q(a, f(a))\right],
\end{aligned}$$

is defined by

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{S}} : [A \simeq A'] \to \\
&[A' \to \mathcal{U}_m] \to \left[\prod_{a':A'}[P'(a') \to \mathcal{U}_n]\right] \to \\
&\left[\prod_{a':A'}\prod_{b':P'(a')} Q'(a',b')\right] \to \Pi(P') \to A' \to g'(e_A(e_A^{-1}(a')), f'(e_A(e_A^{-1}(a')))) = g'(a', f'(a')) \\
&(e_A, P', Q', g', f', a') \mapsto \mathsf{ap}_{\mathsf{S}_{P'Q'(g',f')}}(\mathsf{rightInv}_{e_A}(a')).
\end{aligned}$$

- ap for the remaining introduction and elimination functions is straightforward.

ap respects reflexivity and composition, thus turning functions into functors. We also need to ensure that reduction of ap is confluent with other reductions. Informally, we can observe from the type constraints in each case that there is almost always only one possible definition of ap. Two exceptions are currently known:

- There are two possible definitions of $\mathsf{ap}_{\mathsf{S}_{PQ(g,f)}}$, which are equal because naturality is always guaranteed. This equality might not be definitional.
- For an equivalence e between types, the equality between $(e^{-1})^{-1}$ and e might not be definitional.

3.2 Elimination

For $A : \mathcal{U}_m$, $a, a' : A$, $P : \prod_{b:A}[a = b \to \mathcal{U}_n]$, and $e_a : a = a'$, we have

$$\mathsf{ap}_P(e_a) : P(a) =^{\mathsf{ap}_Q(e_b)} P(a'),$$

where

$$\begin{aligned} Q : A &\to \mathcal{U}_{\max(m,n+1)} \\ b &\mapsto [a = b \to \mathcal{U}_n]. \end{aligned}$$

Due to the behavior of ap with respect to equality, we have

$$\begin{aligned} \mathsf{ap}_Q : A \to A \to b = b' \to [a = b \to \mathcal{U}_n] &\to a = b' \to \mathcal{U}_n \\ (b, b', e_b, P, f) &\mapsto P(e_b^{-1} \circ f), \end{aligned}$$

and therefore

$$\mathsf{ap}_P(e_a) : \prod_{f:a=a'} [P(a, e_a^{-1} \circ f) \simeq P(a', f)]$$

and in particular an instance of

$$P(a, \mathsf{refl}_a) \simeq P(a', e_a),$$

corresponding to based path induction.

4 Conclusion

We have constructed a type theory where identity types reduce according to their extensionality principle in HoTT, including univalence, while at the same time behaving as expected regarding substitution principles. A consequence is that in a potential theorem prover implementing this type theory, equality is very well-behaved.

- For each structure, which can be regarded as a nested Σ type, equality of its instances is *by definition* the same as isomorphism.
- Transporting proofs and data along an equivalence of types (in particular, along an isomorphism of algebraic structures), is the same as rewriting along an equality. In contrast to HoTT, no explicit conversion between equality and equivalence is necessary, and the resulting transportation is defined directly in terms of the equivalence instead of containing an irreducible axiom.
- Function extensionality holds by definition.

5 Future Work

We are currently verifying by computer that all reduction rules are well-typed, with respect to all possible reductions performed on the types involved in each particular rule. To ensure consistency, however, stronger guarantees are needed. While termination looks relatively obvious, the potential lack of confluence (due to some necessarily arbitrary choices) is worrying.

To be useful, a type former for inductive types must be added. This should hopefully be fairly straightforward because the correct definition of equality on inductive types is always obvious. A less obvious case is the addition of higher inductive types (HITs), where the situation is exactly reversed: the entire concept of HITs is footed on the complexity of their identity types.

One goal is obviously the implementation of this theory in a theorem prover. To achieve this, it may make sense to translate the reduction rules on combinators back to lambda terms, so that combinators are no longer needed.

A library of mathematics in such a prover would have an interesting property: Proofs and constructions that look simple but contain complex rewrites would reduce to terms that are "exportable," both to other provers and also to theories constructed in the prover itself. For example, such terms will often still be valid when types are replaced with categories, higher categories, or topological spaces. From another angle, this suggests that the prover could contain a reflection mechanism that allows such structures to be treated as types, by providing appropriate mappings.

Finally, there is certainly room for lots of minor improvements. For example, the definition of $[a =^e b]$ as $[e(a) = b]$ often leads to equalities that unnecessarily include the term e^{-1} on the left-hand side instead of e on the right-hand side. In a practical implementation, it would be useful to specialize the definition for the case that a and b are functions and e originates from an equivalence on their domains, so that equality of structures reduces to the usual definition of isomorphism instead of a type that is only equivalent but not definitionally so.

Also, the definition of equality for the type $\mathbf{2}$ is a bit ad-hoc. If $\mathbf{2}$ was regarded as a *universe* containing the *types* $\mathbf{0}$ and $\mathbf{1}$ (i.e. classical propositions, due to the elimination function) instead of values 0 and 1, equivalence of types would yield the correct definition. Although this just seems like a minor curiosity, there may be a useful extension of this principle that generalizes the concept of universes, perhaps in a way that subsumes the envisioned reflection mechanism.

References

[1] Thorsten Altenkirch, Ambrus Kaposi, and Michael Shulmann. Towards higher observational type theory. In *TYPES 2022, 28th International Conference on Types for Proofs and Programs.* `https://types22.inria.fr/files/2022/06/TYPES_2022_paper_37.pdf`, 2022. Accessed: 2023-01-07.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[3] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013. Accessed: 2023-01-07.

[4] Wikipedia. Combinatory logic. `https://en.wikipedia.org/wiki/Combinatory_logic`. Accessed: 2023-01-07.