Finitary Higher Inductive Types in the Groupoid Model

Sections 1-3

Mark Lapidus, Pim Leerkes

December 13, 2024

Outline

Problem Set-Up

Recall.. Martin-Löf introduced $\mathbf{I}(A, a, a')$

- Elements of $\mathbf{I}(A, a, a')$ are proofs that a and a' are equal elements of A.
- Can obtain an infinite tower of higher identity types:

$$A$$
$$\mathbf{I}(A, a, a')$$
$$\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$$
$$\mathbf{I}(\mathbf{I}(\mathbf{I}(A, a, a'), p, p'), \theta, \theta')$$
$$\ldots$$

Problem Set-Up

Recall.. Martin-Löf introduced $\mathbf{I}(A, a, a')$

- Elements of $\mathbf{I}(A, a, a')$ are proofs that a and a' are equal elements of A.
- Can obtain an infinite tower of higher identity types:

$$A$$
$$\mathbf{I}(A, a, a')$$
$$\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$$
$$\mathbf{I}(\mathbf{I}(\mathbf{I}(A, a, a'), p, p'), \theta, \theta')$$
$$\dots$$

Problem Set-Up

Recall.. Martin-Löf introduced $\mathbf{I}(A, a, a')$

- Elements of $\mathbf{I}(A, a, a')$ are proofs that a and a' are equal elements of A.
- Can obtain an infinite tower of higher identity types:

$$A$$
$$\mathbf{I}(A, a, a')$$
$$\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$$
$$\mathbf{I}(\mathbf{I}(\mathbf{I}(A, a, a'), p, p'), \theta, \theta')$$
$$\dots$$

  – Collapses in extensional type theory. Why?

Problem Set-Up

..infinite tower of higher identity types:

$$A$$
$$\mathbf{I}(A, a, a')$$
$$\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$$
$$\mathbf{I}(\mathbf{I}(\mathbf{I}(A, a, a'), p, p'), \theta, \theta')$$
$$\ldots$$

Works in intensional type theory.

- Proved by Hofmann and Streicher using the groupoid model.

Higher Inductive Types (HIT)

Definition

A **higher inductive type** is a type in which all the iterated identity types are generated inductively.

Examples:

- 1-hit: $\mathbb{I}(A, a, a')$
- 2-hit: $\mathbb{I}(\mathbb{I}(A, a, a'), p, p')$

Has a topological interpretation:

- a and a' are points in space;
- p and p' are paths from from a to a';
- θ and θ' are homotopies between p and p'.

Higher Inductive Types (HIT)

Definition

A **higher inductive type** is a type in which all the iterated identity types are generated inductively.

Examples:

- 1-hit: $\mathbf{I}(A, a, a')$
- 2-hit: $\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$

Has a topological interpretation:

- a and a' are points in space;
- p and p' are paths from from a to a';
- θ and θ' are homotopies between p and p'.

Higher Inductive Types (HIT)

Definition

A **higher inductive type** is a type in which all the iterated identity types are generated inductively.

Examples:

- 1-hit: $\mathbf{I}(A, a, a')$
- 2-hit: $\mathbf{I}(\mathbf{I}(A, a, a'), p, p')$

Has a topological interpretation:

- a and a' are points in space;
- p and p' are paths from from a to a';
- θ and θ' are homotopies between p and p'.

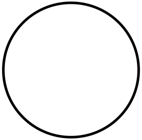
Higher Inductive Types: Examples

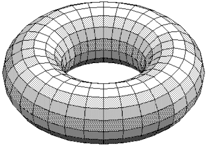
Circle is a 1-hit.

```
Inductive circle : Type :=
| base : circle
| loop : base == base.
```

Torus is a 2-hit (next presentation).

- 2-path represents the surface that commutes meridional and longitudinal loops.





Our Goals

Problem

Formalisation of syntax and semantics is still lacking.

Want to formulate a general theory of higher inductive types:

1. Represent CL as a 1-HIT.
2. Determine general schema for point and path constructors;
3. Construct the elimination and equality rules;

Our Goals

Problem

Formalisation of syntax and semantics is still lacking.

Want to formulate a general theory of higher inductive types:

1. Represent CL as a 1-HIT.
2. Determine general schema for point and path constructors;
3. Construct the elimination and equality rules;

Dependent Type Theory Background: Notation

Function Types

- *Dependent function types*: $(x : A) \to B(x)$
- *Non-dependent function types*: $A \to B$

Identity Types

$$a =_A a' \quad \text{(preferred notation)} \quad \text{instead of} \quad \mathsf{I}(A, a, a')$$

Function Application

- If $x : A \vdash C$, then we write $C(x)$ to emphasize dependence.
- Write $C(a)$ to denote substituting a for x in C.

Dependent Type Theory Background: Notation

Function Types

- *Dependent function types*: $(x : A) \to B(x)$
- *Non-dependent function types*: $A \to B$

Identity Types

$$a =_A a' \quad \text{(preferred notation)} \quad \text{instead of} \quad \mathbf{I}(A, a, a')$$

Function Application

- If $x : A \vdash C$, then we write $C(x)$ to emphasize dependence.
- Write $C(a)$ to denote substituting a for x in C.

Dependent Type Theory Background: Notation

Function Types

- *Dependent function types*: $(x : A) \to B(x)$
- *Non-dependent function types*: $A \to B$

Identity Types

$$a =_A a' \quad \text{(preferred notation)} \quad \text{instead of} \quad \mathbf{I}(A, a, a')$$

Function Application

- If $x : A \vdash C$, then we write $C(x)$ to emphasize dependence.
- Write $C(a)$ to denote substituting a for x in C.

Introduction Rule

$$\mathrm{refl} : (x : A) \to x =_A x$$

Elimination Rule (Induction Principle)

$$J_C : (x : A) \to C(x, \mathrm{refl}(x)) \to (y : A) \to (z : x =_A y) \to C(y, z)$$

where $x : A, y : A, z : x =_A y \vdash C(y, z)$.

Equality Rule

$$J_C(x, d, x, \mathrm{refl}(x)) = d$$

Dependent Type Theory Background: Rules

Introduction Rule

$$\mathsf{refl} : (x : A) \to x =_A x$$

Elimination Rule (Induction Principle)

$$J_C : (x : A) \to C(x, \mathsf{refl}(x)) \to (y : A) \to (z : x =_A y) \to C(y, z)$$

where $x : A, y : A, z : x =_A y \vdash C(y, z)$.

Equality Rule

$$J_C(x, d, x, \mathsf{refl}(x)) = d$$

Dependent Type Theory Background: Rules

Introduction Rule

$$\mathrm{refl} : (x : A) \to x =_A x$$

Elimination Rule (Induction Principle)

$$J_C : (x : A) \to C(x, \mathrm{refl}(x)) \to (y : A) \to (z : x =_A y) \to C(y, z)$$

where $x : A, y : A, z : x =_A y \vdash C(y, z)$.

Equality Rule

$$J_C(x, d, x, \mathrm{refl}(x)) = d$$

Heterogeneous Identity Type

$a =_p^B a'$ compares $a : B(x)$ and $a' : B(x')$ where $p : x =_A x'$

Identity Preservation

$$\mathbf{apd}_f : (p : x =_A x') \to f(x) =_p^B f(x')$$

Dependent Type Theory Background: Rules

Heterogeneous Identity Type

$a =_p^B a'$ compares $a : B(x)$ and $a' : B(x')$ where $p : x =_A x'$

Identity Preservation

$$\mathbf{apd}_f : (p : x =_A x') \to f(x) =_p^B f(x')$$

Combinatory Logic

- examples of combinators: S,I and K
- as 1-hit: CL is a type

$$Kxy = x$$

$$Ix = x$$

$$Sxyz = xz(yz)$$

Introduction Rules

Point Constructors

$$\mathsf{K}, \mathsf{S} : \mathsf{CL} \quad \text{and} \quad \mathbf{app} : \mathsf{CL} \to \mathsf{CL} \to \mathsf{CL}$$

Path Constructors

- $\mathsf{K}_{conv} : (x, y : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathsf{K}, x), y) =_{\mathsf{CL}} x$
- $\mathsf{S}_{conv} : (x, y, z : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathbf{app}(\mathsf{S}, x), y), z) =_{\mathsf{CL}} \mathbf{app}(\mathbf{app}(x, z), \mathbf{app}(y, z))$

Properties Derived from $=_{CL}$

- Reflexivity
- Transitivity
- Symmetry
- Application preserves equality

Introduction Rules

Point Constructors

$$\mathsf{K}, \mathsf{S} : \mathsf{CL} \quad \text{and} \quad \mathbf{app} : \mathsf{CL} \to \mathsf{CL} \to \mathsf{CL}$$

Path Constructors

- $\mathsf{K}_{conv} : (x, y : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathsf{K}, x), y) =_{\mathsf{CL}} x$
- $\mathsf{S}_{conv} : (x, y, z : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathbf{app}(\mathsf{S}, x), y), z) =_{\mathsf{CL}} \mathbf{app}(\mathbf{app}(x, z), \mathbf{app}(y, z))$

Properties Derived from $=_{CL}$

- Reflexivity
- Transitivity
- Symmetry
- Application preserves equality

Introduction Rules

Point Constructors

$$\mathsf{K}, \mathsf{S} : \mathsf{CL} \quad \text{and} \quad \mathbf{app} : \mathsf{CL} \to \mathsf{CL} \to \mathsf{CL}$$

Path Constructors

- $\mathsf{K}_{conv} : (x, y : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathsf{K}, x), y) =_{\mathsf{CL}} x$
- $\mathsf{S}_{conv} : (x, y, z : \mathsf{CL}) \to \mathbf{app}(\mathbf{app}(\mathbf{app}(\mathsf{S}, x), y), z) =_{\mathsf{CL}} \mathbf{app}(\mathbf{app}(x, z), \mathbf{app}(y, z))$

Properties Derived from $=_{CL}$

- Reflexivity
- Transitivity
- Symmetry
- Application preserves equality

Elimination Rules

Assumptions

- $\tilde{\mathsf{K}} : C(\mathsf{K})$
- $\tilde{\mathsf{S}} : C(\mathsf{S})$
- $\tilde{\mathbf{app}} : (x : \mathsf{CL}) \to C(x) \to (y : \mathsf{CL}) \to C(y) \to C(\mathbf{app}(x, y))$

Path Assumptions

- $\tilde{\mathsf{K}}_{conv} : (x, y : \mathsf{CL}) \to (\tilde{x} : C(x)) \to (\tilde{y} : C(y)) \to \tilde{\mathbf{app}}(\mathbf{app}(\mathsf{K}, x), \tilde{\mathbf{app}}(\mathsf{K}, \tilde{\mathsf{K}}, x, \tilde{x}), y, \tilde{y}) =^{C}_{\mathsf{K}_{conv}(x,y)} \tilde{x}$
- Analogous for $\tilde{\mathsf{S}}_{conv}$

Result

$$f : (x : \mathsf{CL}) \to C(x)$$

Elimination Rules

Assumptions

- $\tilde{\mathsf{K}} : C(\mathsf{K})$
- $\tilde{\mathsf{S}} : C(\mathsf{S})$
- $\tilde{\mathbf{app}} : (x : \mathsf{CL}) \to C(x) \to (y : \mathsf{CL}) \to C(y) \to C(\mathbf{app}(x, y))$

Path Assumptions

- $\tilde{\mathsf{K}}_{conv} : (x, y : \mathsf{CL}) \to (\tilde{x} : C(x)) \to (\tilde{y} : C(y)) \to \tilde{\mathbf{app}}(\mathbf{app}(\mathsf{K}, x), \tilde{\mathbf{app}}(\mathsf{K}, \tilde{\mathsf{K}}, x, \tilde{x}), y, \tilde{y}) =^{C}_{\mathsf{K}_{conv}(x,y)} \tilde{x}$
- Analogous for $\tilde{\mathsf{S}}_{conv}$

Result

$$f : (x : \mathsf{CL}) \to C(x)$$

Elimination Rules

Assumptions

- $\tilde{\mathsf{K}} : C(\mathsf{K})$
- $\tilde{\mathsf{S}} : C(\mathsf{S})$
- $\tilde{\mathbf{app}} : (x : \mathsf{CL}) \to C(x) \to (y : \mathsf{CL}) \to C(y) \to C(\mathbf{app}(x, y))$

Path Assumptions

- $\tilde{\mathsf{K}}_{conv} : (x, y : \mathsf{CL}) \to (\tilde{x} : C(x)) \to (\tilde{y} : C(y)) \to \tilde{\mathbf{app}}(\mathbf{app}(\mathsf{K}, x), \tilde{\mathbf{app}}(\mathsf{K}, \tilde{\mathsf{K}}, x, \tilde{x}), y, \tilde{y}) =^{C}_{\mathsf{K}_{conv}(x,y)} \tilde{x}$
- Analogous for $\tilde{\mathsf{S}}_{conv}$

Result

$$f : (x : \mathsf{CL}) \to C(x)$$

Function Equality

- $f(\mathsf{K}) = \tilde{\mathsf{K}}$
- $f(\mathsf{S}) = \tilde{\mathsf{S}}$
- $f(\mathbf{app}(x, y)) = \tilde{\mathbf{app}}(x, f(x), y, f(y))$

Path Equality

- $\mathbf{apd}_f(\mathsf{K}_{conv}(x, y)) = \tilde{\mathsf{K}}_{conv}(x, y, f(x), f(y))$
- $\mathbf{apd}_f(\mathsf{S}_{conv}(x, y, z)) = \tilde{\mathsf{S}}_{conv}(x, y, z, f(x), f(y), f(z))$

Equality Rules

Function Equality

- $f(\mathsf{K}) = \tilde{\mathsf{K}}$
- $f(\mathsf{S}) = \tilde{\mathsf{S}}$
- $f(\mathbf{app}(x, y)) = \tilde{\mathbf{app}}(x, f(x), y, f(y))$

Path Equality

- $\mathbf{apd}_f(\mathsf{K}_{conv}(x, y)) = \tilde{\mathsf{K}}_{conv}(x, y, f(x), f(y))$
- $\mathbf{apd}_f(\mathsf{S}_{conv}(x, y, z)) = \tilde{\mathsf{S}}_{conv}(x, y, z, f(x), f(y), f(z))$

Setoids

Idea: dependent type theory with $(x : A) \to B(x)$, $a =_A a'$, and CL, has a setoid model.

Definition

A **setoid** is a set S equipped with an equivalence relation R.

- Intuitively, can be thought of as a set where elements are considered equivalent under a given relation.

Examples are:

- Modulo arithmetic: $(\mathbb{Z}, \equiv \mod n)$
- Rational numbers: $(a, b) \sim (c, d) \iff ad = bc$

Setoids

Idea: dependent type theory with $(x : A) \to B(x)$, $a =_A a'$, and CL, has a setoid model.

Definition

A **setoid** is a set S equipped with an equivalence relation R.

- Intuitively, can be thought of as a set where elements are considered equivalent under a given relation.

Examples are:

- Modulo arithmetic: $(\mathbb{Z}, \equiv \mod n)$
- Rational numbers: $(a, b) \sim (c, d) \iff ad = bc$

Setoids

Idea: dependent type theory with $(x : A) \to B(x)$, $a =_A a'$, and CL, has a setoid model.

Definition

A **setoid** is a set S equipped with an equivalence relation R.

- Intuitively, can be thought of as a set where elements are considered equivalent under a given relation.

Examples are:

- Modulo arithmetic: $(\mathbb{Z}, \equiv \mod n)$
- Rational numbers: $(a, b) \sim (c, d) \iff ad = bc$

Setoid Model

1. Interpret **type as a setoid** A, consisting of a set A_0 together with an equivalence relation R.
   - R is represented as a binary family of sets $(A_1(x,x'))_{x,x' \in A_0}$ such that $A_1(x,x')$ is inhabitated $\iff$ $R(x,x')$ holds.
2. CL is a setoid $(\mathrm{CL}_0, \mathrm{CL}_1)$ where CL_0 is an inductive type generated by K, S and **app** and CL_1 is an inductive family generated by K_{conv} and S_{conv} and the constructors for transitivity, reflexivity, symmetry and preservation of equality by **app**.

Constructing a General Schema

Use the schema in the style of inductive families:

- point constructor $\leftarrow$ constructor for an inductive type
- path constructor $\leftarrow$ constructor for a binary inductive family

General form for an inductive type $\mathbf{H}$:

$$
\begin{aligned}
(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1}))& \\
\to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots& \\
\to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}&
\end{aligned}
$$

- $A_m(x_1, \ldots, x_{m-1}))$ is a type if $(x_1 : A_1, \ldots, x_{m-1} : A_{m-1}(x_1, \ldots, x_{m-2}))$;
- $B_1(\ldots)$ and $B_n(\ldots)$ are types if $(x_1 : A_1, \ldots, x_m : A_m(x_1, \ldots, x_{m-1}))$.
- A_i and B_j do not depend on $\mathbf{H}$.

Constructing a General Schema

Use the schema in the style of inductive families:

- point constructor $\leftarrow$ constructor for an inductive type
- path constructor $\leftarrow$ constructor for a binary inductive family

General form for an inductive type $\mathbf{H}$:

$$
\begin{aligned}
(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1})) \\
\to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots \\
\to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}
\end{aligned}
$$

- $A_m(x_1, \ldots, x_{m-1})$ is a type if $(x_1 : A_1, \ldots, x_{m-1} : A_{m-1}(x_1, \ldots, x_{m-2}))$;
- $B_1(\ldots)$ and $B_n(\ldots)$ are types if $(x_1 : A_1, \ldots, x_m : A_m(x_1, \ldots, x_{m-1}))$.
- A_i and B_j do not depend on $\mathbf{H}$.

Introduction
Problem Set-Up
Combinatory Logic as a 1-Hit
Dependent Type Theory Background
Combinatory Logic
Setoid Model
A Schema for 1-Hits
Point Constructors
Path Constructors
Simplified Form for Point and Path Constructors
Elimination and Equality Rules
Lifting Function
Examples

Constructing a General Schema

Use the schema in the style of inductive families:

- point constructor $\leftarrow$ constructor for an inductive type
- path constructor $\leftarrow$ constructor for a binary inductive family

General form for an inductive type $\mathbf{H}$:

$$
\begin{aligned}
(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1})) \\
\to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots \\
\to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}
\end{aligned}
$$

- $A_m(x_1, \ldots, x_{m-1}))$ is a type if $(x_1 : A_1, \ldots, x_{m-1} : A_{m-1}(x_1, \ldots, x_{m-2}))$;
- $B_1(\ldots)$ and $B_n(\ldots)$ are types if $(x_1 : A_1, \ldots, x_m : A_m(x_1, \ldots, x_{m-1}))$.
- A_i and B_j do not depend on $\mathbf{H}$.

Constructing a General Schema

Use the schema in the style of inductive families:

- point constructor $\leftarrow$ constructor for an inductive type
- path constructor $\leftarrow$ constructor for a binary inductive family

General form for an inductive type **H**:

$$\begin{aligned}(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1}))\\ \to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots\\ \to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}\end{aligned}$$

- $A_m(x_1, \ldots, x_{m-1}))$ is a type if $(x_1 : A_1, \ldots, x_{m-1} : A_{m-1}(x_1, \ldots, x_{m-2}))$;
- $B_1(\ldots)$ and $B_n(\ldots)$ are types if $(x_1 : A_1, \ldots, x_m : A_m(x_1, \ldots, x_{m-1}))$.
- A_i and B_j do not depend on **H**.

Constructing a General Schema

Use the schema in the style of inductive families:

- point constructor $\leftarrow$ constructor for an inductive type
- path constructor $\leftarrow$ constructor for a binary inductive family

General form for an inductive type $\mathbf{H}$:

$$\begin{aligned}(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1}))\\ \to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots\\ \to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}\end{aligned}$$

- $A_m(x_1, \ldots, x_{m-1}))$ is a type if $(x_1 : A_1, \ldots, x_{m-1} : A_{m-1}(x_1, \ldots, x_{m-2}))$;
- $B_1(\ldots)$ and $B_n(\ldots)$ are types if $(x_1 : A_1, \ldots, x_m : A_m(x_1, \ldots, x_{m-1}))$.
- A_i and B_j do not depend on $\mathbf{H}$.

Restriction to Finitary HITs

$$\begin{aligned}(x_1 : A_1) \to \cdots \to (x_m : A_m(x_1, \ldots, x_{m-1}))\\ \to (B_1(x_1, \ldots, x_m) \to \mathbf{H}) \to \cdots\\ \to (B_n(x_1, \ldots, x_m) \to \mathbf{H}) \to \mathbf{H}\end{aligned}$$

If B_j is empty $\to$ **finitary inductive definition**.

Point Constructors

$$
\begin{aligned}
c_0 : (x_1 : A_1) \to \ldots \to (x_m : A_m(x_1, \ldots, x_{m-1})) \\
\to \mathbf{H} \to \ldots \to \mathbf{H} \to \mathbf{H}
\end{aligned}
$$

Path Constructors

$$
\begin{aligned}
c_1 : {} & (x_1 : B_1) \to \cdots \to (x_n : B_n(x_1, \ldots, x_n)) \\
& \to (y_1 : \mathbf{H}) \to \cdots \to (y_{n'} : \mathbf{H}) \\
& \to p_1(x_1, \ldots, x_n, y_1, \ldots, y_{n'}) =_{\mathbf{H}} q_1(x_1, \ldots, x_n, y_1, \ldots, y_{n'}) \\
& \vdots \\
& \to p_{n''}(x_1, \ldots, x_n, y_1, \ldots, y_{n'}) =_{\mathbf{H}} q_{n''}(x_1, \ldots, x_n, y_1, \ldots, y_{n'}) \\
& \to p'(x_1, \ldots, x_n, y_1, \ldots, y_{n'}) =_{\mathbf{H}} q'(x_1, \ldots, x_n, y_1, \ldots, y_{n'})
\end{aligned}
$$

- $y_j : \mathbf{H}$ is an **inductive premise**;
- $p ::= y \mid c_0(a_1, \ldots, a_m, p_1, \ldots, p_k)$ - syntax for **point constructor patterns**

Simplified Form for Point and Path Constructors

One point constructor with $m = 1$:

$$c_0 : A \to \mathbf{H} \to \mathbf{H}$$

One path constructor with $n = n' = n'' = 1$:

$$\begin{aligned} c_1 : (x : B) \to (y : \mathbf{H}) \to p(x, y) &=_{\mathbf{H}} q(x, y) \\ \to p'(x, y) &=_{\mathbf{H}} q'(x, y) \end{aligned}$$

Simplified Form for Point and Path Constructors

One point constructor with $m = 1$:

$$c_0 : A \to \mathbf{H} \to \mathbf{H}$$

One path constructor with $n = n' = n'' = 1$:

$$\begin{aligned} c_1 : (x : B) \to (y : \mathbf{H}) &\to p(x, y) =_{\mathbf{H}} q(x, y) \\ &\to p'(x, y) =_{\mathbf{H}} q'(x, y) \end{aligned}$$

Point assumption

$$\tilde{c}_0 : (x : A) \to (y : A) \to C(y) \to C(c_0(x, y))$$

Path assumption

$$\tilde{c}_1 : (x : B) \to (y : \mathbf{H}) \to (\tilde{y} : C(y)) \to (z : p =_{\mathbf{H}} q) \to T_0(p) =^C_z T_0(q) \to T_0(p') =^C_{c_1(x,y,z)} T_0(q')$$

Result: a function $f : (x : \mathbf{H}) \to C(x)$.

Equality rules

$$f(c_0(x, y)) = \tilde{c}_0(x, y, f(y))$$

$$\mathbf{apd}_f(c_1(x, y, z)) = \tilde{c}_1(x, y, f(y), z, \mathbf{apd}_f(z))$$

Elimination and equality rules

Introduction
Problem Set-Up
Combinatory Logic as a 1-Hit
Dependent Type Theory Background
Combinatory Logic
Setoid Model
A Schema for 1-Hits
Point Constructors
Path Constructors
Simplified Form for Point and Path Constructors
Elimination and Equality Rules
Lifting Function
Examples

Point assumption

$$\tilde{c}_0 : (x : A) \to (y : A) \to C(y) \to C(c_0(x, y))$$

Path assumption

$$\tilde{c}_1 : (x : B) \to (y : \mathbf{H}) \to (\tilde{y} : C(y)) \to (z : p =_{\mathbf{H}} q) \to$$
$$T_0(p) =^C_z T_0(q) \to T_0(p') =^C_{c_1(x,y,z)} T_0(q')$$

Result: a function $f : (x : \mathbf{H}) \to C(x)$.

Equality rules

$$f(c_0(x, y)) = \tilde{c}_0(x, y, f(y))$$

$$\mathbf{apd}_f(c_1(x, y, z)) = \tilde{c}_1(x, y, f(y), z, \mathbf{apd}_f(z))$$

The lifting function: example

- The lifting of $p : \mathbf{H}$ is denoted $T_0(p) : C(p)$
- The idea is that $T_0(p) = f(p)$ for the resulting function f

Example ($\tilde{K}_{conv}$):

$T_0(app(app(K, x), y)) = \tilde{app}(app(K, x), \tilde{app}(K, \tilde{K}, x, \tilde{x}), y, \tilde{y})$
$T_0(x) = \tilde{x}$

This lifting function is defined by:

$T_0(x) = \tilde{x}$, $T_0(y) = \tilde{y}$, $T_0(K) = \tilde{K}$, $T_0(S) = \tilde{S}$ and
$T_0(app(t, t')) = \tilde{app}(t, T_0(t), t', T_0(t'))$

The lifting function: example

Introduction
Problem Set-Up
Combinatory Logic as a 1-Hit
Dependent Type Theory Background
Combinatory Logic
Setoid Model
A Schema for 1-Hits
Point Constructors
Path Constructors
Simplified Form for Point and Path Constructors
Elimination and Equality Rules
Lifting Function
Examples

- The lifting of $p : \mathbf{H}$ is denoted $T_0(p) : C(p)$
- The idea is that $T_0(p) = f(p)$ for the resulting function f

Example ($\tilde{K}_{conv}$):

$T_0(app(app(K, x), y)) = \tilde{app}(app(K, x), \tilde{app}(K, \tilde{K}, x, \tilde{x}), y, \tilde{y})$
$T_0(x) = \tilde{x}$

This lifting function is defined by:

$T_0(x) = \tilde{x}$, $T_0(y) = \tilde{y}$, $T_0(K) = \tilde{K}$, $T_0(S) = \tilde{S}$ and
$T_0(app(t, t')) = \tilde{app}(t, T_0(t), t', T_0(t'))$

The lifting function: example

- The lifting of $p : \mathbf{H}$ is denoted $T_0(p) : C(p)$
- The idea is that $T_0(p) = f(p)$ for the resulting function f

Example ($\tilde{K}_{conv}$):

$$T_0(app(app(K,x),y)) = a\tilde{p}p(app(K,x), a\tilde{p}p(K,\tilde{K},x,\tilde{x}),y,\tilde{y})$$
$$T_0(x) = \tilde{x}$$

This lifting function is defined by:

$T_0(x) = \tilde{x}$, $T_0(y) = \tilde{y}$, $T_0(K) = \tilde{K}$, $T_0(S) = \tilde{S}$ and
$T_0(app(t,t')) = a\tilde{p}p(t, T_0(t), t', T_0(t'))$

The lifting function: general

General definition of $(T_0(p(x,y)) : C(p(x,y)))$:

$$T_0(y) = \tilde{y}$$

$$T_0(c_0(a,p)) = \tilde{c_0}(a, p, T_0(p))$$

Hence, $x : B, y : \mathbf{H}, \tilde{y} : C(y) \vdash T_0(p(x,y)) : C(p(x,y))$ and $T_0(p)(x, y, f(y)) = f(p(x,y))$

The lifting function: general

General definition of ($T_0(p(x,y)) : C(p(x,y))$):

$$T_0(y) = \tilde{y}$$

$$T_0(c_0(a,p)) = \tilde{c}_0(a,p,T_0(p))$$

Hence, $x : B, y : \mathbf{H}, \tilde{y} : C(y) \vdash T_0(p(x,y)) : C(p(x,y))$ and $T_0(p)(x,y,f(y)) = f(p(x,y))$

Example: The circle again

```
Inductive circle : Type :=
| base : circle
| loop : base == base.
```

$$\frac{\tilde{base} : C(\text{base}) \quad \tilde{loop} : T_0(\text{base}) =^{C}_{loop} T_0(\text{base})}{f : (x : \text{circle}) \to C(x)}$$

- $f(base) = \tilde{base}$
- $apd_f(loop) = \tilde{loop}$

Example: Natlist

```
Inductive Natlist: Type :=
| Nil : Natlist
| Cons : nat -> Natlist -> Natlist
| NilEq: Nil == Nil
| ConsEq : forall n1, n2 : nat ->
        forall l1, l2 : Natlist ->
        n1 == n2 -> l1 == l2 ->
        Cons(n1,l1) == Cons(n2,l2)
```

Example: Natlist (elimination rule)

Point constructors:

$$\tilde{Nil} : C(Nil)$$

$$\tilde{Cons} : (n : nat) \to (l : Natlist) \to C(l) \to C(cons(n, l))$$

Path constructors:

$$ni\tilde{l}Eq : T_0(Nil) =^{C}_{Nileq} T_0(Nil)$$

$$\begin{aligned} Co\tilde{ns}Eq : \quad & (n1, n2 : \text{nat}) \to (l1, l2 : \text{Natlist}) \to (\tilde{l1} : C(l1)) \\ & \to (\tilde{l2} : C(l2)) \to (x : n1 =_{\text{Natlist}} n2) \to (y : l1 =_{\text{Natlist}} l2) \\ & T_0(n1) =^{C}_{x} T_0(n2) \to T_0(l1) =^{C}_{y} T_0(l2) \to \\ & T_0(\text{Cons}(n1, l1)) =^{C}_{\text{ConsEq}(n1,n2,l1,l2,x,y)} T_0(\text{Cons}(n2, l2)) \end{aligned}$$

Example: Natlist (equality rule)

$$
\begin{aligned}
f(\mathrm{Nil}) &= \widetilde{\mathrm{Nil}} \\
f(\mathrm{Cons}(n, l)) &= \widetilde{\mathrm{Cons}}(n, l, f(l)) \\
\mathbf{apd}_f(\mathrm{NilEq}) &= \widetilde{\mathrm{NilEq}} \\
\mathbf{apd}_f(\mathrm{ConsEq}(n_1, n_2, l_1, l_2, x, y)) &= \widetilde{\mathrm{ConsEq}}(n_1, n_2, l_1, l_2, \\
&\quad f(l_1), f(l_2), z_1, z_2, \\
&\quad \mathbf{apd}_f(z_1), \mathbf{apd}_f(z_2)).
\end{aligned}
$$

End of presentation

Questions?