# Foundations of Type theory for HoTT

Siddhartha Gadgil

Department of Mathematics
Indian Institute of Science

February 12, 2015

# Foundations overview

- We review the foundations of the type theory underlying homotopy type theory.

# Foundations overview

- We review the foundations of the type theory underlying homotopy type theory.
- This is a literate agda document.

# Foundations overview

- We review the foundations of the type theory underlying homotopy type theory.
- This is a literate agda document.
- We must include a module statement, matching the file name.

```
open import Base

module Foundations where
```

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function* $f : \mathbb{R} \to \mathbb{R}$ *is said to be continuous at* $x$ *if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R}(|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function $f:\mathbb{R}\to\mathbb{R}$ is said to be continuous at $x$ if*

$$\forall\epsilon > 0\exists\delta > 0\forall y\in\mathbb{R}(|y-x|\leq\delta \implies |f(y)-f(x)|<\epsilon).$$

- We however do not explicitly give rules saying why *a function $f:\mathbb{R}\to\mathbb{R}$ is said to be brown at $x$ if*

$$\forall\exists\delta\forall z\in\mathbb{R}(|y-f(x)|\leq\delta \implies |f(y)-f(x)|)$$

makes no sense.

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be continuous at $x$ if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R}(|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

- We however do not explicitly give rules saying why *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be brown at $x$ if*

$$\forall \exists \delta \forall z \in \mathbb{R}(|y - f(x)| \leq \delta \implies |f(y) - f(x)|)$$

  makes no sense.
- We thus do not give rules for

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be continuous at $x$ if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R}(|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

- We however do not explicitly give rules saying why *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be brown at $x$ if*

$$\forall \exists \delta \forall z \in \mathbb{R}(|y - f(x)| \leq \delta \implies |f(y) - f(x)|)$$

  makes no sense.
- We thus do not give rules for
  - What is a valid expression.

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function* $f : \mathbb{R} \to \mathbb{R}$ *is said to be continuous at* $x$ *if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R}(|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

- We however do not explicitly give rules saying why *a function* $f : \mathbb{R} \to \mathbb{R}$ *is said to be brown at* $x$ *if*

$$\forall \exists \delta \forall z \in \mathbb{R}(|y - f(x)| \leq \delta \implies |f(y) - f(x)|)$$

  makes no sense.
- We thus do not give rules for
  - What is a valid expression.
  - What it represents: term (real number, set etc) or formula (in definitions, theorems).

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function* $f : \mathbb{R} \to \mathbb{R}$ *is said to be continuous at* $x$ *if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R}(|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

- We however do not explicitly give rules saying why *a function* $f : \mathbb{R} \to \mathbb{R}$ *is said to be brown at* $x$ *if*

$$\forall \exists \delta \forall z \in \mathbb{R}(|y - f(x)| \leq \delta \implies |f(y) - f(x)|)$$

  makes no sense.
- We thus do not give rules for
  - What is a valid expression.
  - What it represents: term (real number, set etc) or formula (in definitions, theorems).
  - Rules for deduction.

# Axioms and Rules

- Usual *rigorous* mathematics is based on definitions and aioms, for example *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be continuous at $x$ if*

$$\forall \epsilon > 0 \exists \delta > 0 \forall y \in \mathbb{R} (|y - x| \leq \delta \implies |f(y) - f(x)| < \epsilon).$$

- We however do not explicitly give rules saying why *a function $f : \mathbb{R} \to \mathbb{R}$ is said to be brown at $x$ if*

$$\forall \exists \delta \forall z \in \mathbb{R} (|y - f(x)| \leq \delta \implies |f(y) - f(x)|)$$

  makes no sense.
- We thus do not give rules for
  - What is a valid expression.
  - What it represents: term (real number, set etc) or formula (in definitions, theorems).
  - Rules for deduction.
- We will instead give rules for what are valid expressions, and what are their types. We will need very few axioms.

# Contexts, terms, types, universes

- A context consists of a collection of *terms*.

# Contexts, terms, types, universes

- A context consists of a collection of *terms*.
- Each term has a *type*, mostly unique (denoted, for example, $a : A$).

# Contexts, terms, types, universes

- A context consists of a collection of *terms*.
- Each term has a *type*, mostly unique (denoted, for example, $a : A$).
- The rules concerning a term are determined by its type.

# Contexts, terms, types, universes

- A context consists of a collection of *terms*.
- Each term has a *type*, mostly unique (denoted, for example, $a : A$).
- The rules concerning a term are determined by its type.
- Types are also terms.

# Contexts, terms, types, universes

- A context consists of a collection of *terms*.
- Each term has a *type*, mostly unique (denoted, for example, $a : A$).
- The rules concerning a term are determined by its type.
- Types are also terms.
- A *Universe* is a type $\mathcal{U}$ so that all terms with type $\mathcal{U}$ are themselves types.

# Types of rules

We have rules that:

- let us form terms from other terms.

# Types of rules

We have rules that:

- let us form terms from other terms.
- let us create a new context from a given context, by introducing new terms which can depend on the given context.

# Types of rules

We have rules that:

- let us form terms from other terms.
- let us create a new context from a given context, by introducing new terms which can depend on the given context.
- give the result of substituting one term for another (with the same type) in a given term.

# Types of rules

We have rules that:

- let us form terms from other terms.
- let us create a new context from a given context, by introducing new terms which can depend on the given context.
- give the result of substituting one term for another (with the same type) in a given term.
- allow us to make say that a term $a$ has a specified type $A$.

# Types of rules

We have rules that:

- let us form terms from other terms.
- let us create a new context from a given context, by introducing new terms which can depend on the given context.
- give the result of substituting one term for another (with the same type) in a given term.
- allow us to make say that a term $a$ has a specified type $A$.
- allow us to conclude that a pair of terms are equal (by definition).

# Types of rules

We have rules that:

- let us form terms from other terms.
- let us create a new context from a given context, by introducing new terms which can depend on the given context.
- give the result of substituting one term for another (with the same type) in a given term.
- allow us to make say that a term $a$ has a specified type $A$.
- allow us to conclude that a pair of terms are equal (by definition).
- give a collection of universes, which are present in all contexts.

# Universes

- There is a sequence of universes, $\mathcal{U}_0$, $U_1$, ...

# Universes

- There is a sequence of universes, $\mathcal{U}_0$, $U_1$, ...
- The universe $\mathcal{U}_i$ has type $\mathcal{U}_{i+1}$.

# Universes

- There is a sequence of universes, $\mathcal{U}_0$, $U_1$, $\ldots$
- The universe $\mathcal{U}_i$ has type $\mathcal{U}_{i+1}$.
- These are cumulative, with $U_i \subset U_{i+1}$.

# Universes

- There is a sequence of universes, $\mathcal{U}_0$, $U_1$, $\ldots$
- The universe $\mathcal{U}_i$ has type $\mathcal{U}_{i+1}$.
- These are cumulative, with $U_i \subset U_{i+1}$.
- If a type $T$ has type $\mathcal{U}_i$, it also has type $\mathcal{U}_{i+1}$.

# Function types

- If $A$ and $B$ are types, then we can form the function type $A \to B$.

# Function types

- If $A$ and $B$ are types, then we can form the function type $A \to B$.
- If $f : A \to B$ is a term of a function type, and $a : A$ is a term, then $f(a)$ is a term that has type $B$.

# Function types

- If $A$ and $B$ are types, then we can form the function type $A \to B$.
- If $f : A \to B$ is a term of a function type, and $a : A$ is a term, then $f(a)$ is a term that has type $B$.
- We can form terms of a type $A \to B$ by using a *lambda*-expression $a \mapsto b$.

# Function types

- If $A$ and $B$ are types, then we can form the function type $A \to B$.
- If $f : A \to B$ is a term of a function type, and $a : A$ is a term, then $f(a)$ is a term that has type $B$.
- We can form terms of a type $A \to B$ by using a *lambda*-expression $a \mapsto b$.
- Here $b$ is a term of type $B$ formed from the terms in the context together with a term $a$ we introduce and declare to have type $A$, using the usual rules for forming terms.

# Function types

- If $A$ and $B$ are types, then we can form the function type $A \to B$.
- If $f : A \to B$ is a term of a function type, and $a : A$ is a term, then $f(a)$ is a term that has type $B$.
- We can form terms of a type $A \to B$ by using a *lambda*-expression $a \mapsto b$.
- Here $b$ is a term of type $B$ formed from the terms in the context together with a term $a$ we introduce and declare to have type $A$, using the usual rules for forming terms.
- If $f = a \mapsto b : A \to B$, then for $a' : A$, $f(a')$ equals, by definition, the result of substituting $a$ by $a'$ in $b$.

# Π-types

- A *type family* is a function $B : A \to \mathcal{U}$, where $A$ is a type and $\mathcal{U}$ is a universe.

# Π-types

- A *type family* is a function $B : A \to \mathcal{U}$, where $A$ is a type and $\mathcal{U}$ is a universe.
- Given a type family $B : A \to \mathcal{U}$, we can form the type $\Pi_{a:A} B(a)$ of dependent functions.

# Π-types

- A *type family* is a function $B : A \to \mathcal{U}$, where $A$ is a type and $\mathcal{U}$ is a universe.
- Given a type family $B : A \to \mathcal{U}$, we can form the type $\Pi_{a:A} B(a)$ of dependent functions.
- Given a dependent function $f : \Pi_{a:A} B(a)$ and a term $a : A$, we can form the term $f(a)$ with type $B(a)$.

# Π-types

- A *type family* is a function $B : A \to \mathcal{U}$, where $A$ is a type and $\mathcal{U}$ is a universe.
- Given a type family $B : A \to \mathcal{U}$, we can form the type $\Pi_{a:A} B(a)$ of dependent functions.
- Given a dependent function $f : \Pi_{a:A} B(a)$ and a term $a : A$, we can form the term $f(a)$ with type $B(a)$.
- We can form terms of a type $\Pi_{a:A} B(a)$ by using a $\lambda$-expression $a \mapsto b$, with $b$ a term of type $B(a)$ formed from the terms in the context together with a term $a$ we introduce and declare to be of type $A$.

# Π-types

- A *type family* is a function $B : A \to \mathcal{U}$, where $A$ is a type and $\mathcal{U}$ is a universe.
- Given a type family $B : A \to \mathcal{U}$, we can form the type $\Pi_{a:A} B(a)$ of dependent functions.
- Given a dependent function $f : \Pi_{a:A} B(a)$ and a term $a : A$, we can form the term $f(a)$ with type $B(a)$.
- We can form terms of a type $\Pi_{a:A} B(a)$ by using a $\lambda$-expression $a \mapsto b$, with $b$ a term of type $B(a)$ formed from the terms in the context together with a term $a$ we introduce and declare to be of type $A$.
- If $f = a \mapsto b : \Pi_{a:A} B(a)$, then for $a' : A$, $f(a')$ equals, by definition, the result of substituting $a$ by $a'$ in $b$.

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.
- The constructors for $W$ are terms with specified types, which may depend on $W$.

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.
- The constructors for $W$ are terms with specified types, which may depend on $W$.
- For example, the type $\mathbb{N}$ is inductively generated by the constructors

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.
- The constructors for $W$ are terms with specified types, which may depend on $W$.
- For example, the type $\mathbb{N}$ is inductively generated by the constructors
  - $0 : \mathbb{N}$.

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.
- The constructors for $W$ are terms with specified types, which may depend on $W$.
- For example, the type $\mathbb{N}$ is inductively generated by the constructors
  - $0 : \mathbb{N}$.
  - $succ : \mathbb{N} \to \mathbb{N}$.

# Inductive types: a first look

- We can introduce into a context, simultaneously, a type $W$ *inductively generated* by given *constructors*, and its constructors.
- The constructors for $W$ are terms with specified types, which may depend on $W$.
- For example, the type $\mathbb{N}$ is inductively generated by the constructors
  - $0 : \mathbb{N}$.
  - $succ : \mathbb{N} \to \mathbb{N}$.
- In Agda, this is

```
data ℕ : Type where
  zero : ℕ
  succ : ℕ → ℕ
```

# Inductive types: Lists

- For each type $A : \mathcal{U}$, $List(A)$ is a type inductively defined by its constructors.

# Inductive types: Lists

- For each type $A : \mathcal{U}$, $List(A)$ is a type inductively defined by its constructors.
    - $[] : List(A)$.

# Inductive types: Lists

- For each type $A : \mathcal{U}$, $List(A)$ is a type inductively defined by its constructors.
  - $[] : List(A)$.
  - $cons : A \to List(A) \to List(A)$.

# Inductive types: Lists

- For each type $A : \mathcal{U}$, $List(A)$ is a type inductively defined by its constructors.
  - $[] : List(A)$.
  - $cons : A \to List(A) \to List(A)$.
- In Agda, this is

```
data List(A : Type) : Type where
  [] : List A
  _::_ : List A → List A → List A
```

# Inductive types: Lists

- For each type $A : \mathcal{U}$, $List(A)$ is a type inductively defined by its constructors.
  - $[] : List(A)$.
  - $cons : A \to List(A) \to List(A)$.
- In Agda, this is

```
data List(A : Type) : Type where
  [] : List A
  _::_ : List A → List A → List A
```

- We can view this as a *lambda*-expression with variable $A : \mathcal{U}$, with the right hand side given by the rules for constructing inductive types.

# Recursion and Induction : a first look

- We can define a function on an inductive type $W$ by defining it on each constructor.

# Recursion and Induction : a first look

- We can define a function on an inductive type $W$ by defining it on each constructor.
- To define it on a constructor, we give an expression like the right hand side of a $\lambda$-expression, except that if some argument $w$ to the constructor is of type $W$ (or a more general situation we shall see later), then we can use $f(w)$ in forming the right hand side.

# Recursion and Induction : a first look

- We can define a function on an inductive type $W$ by defining it on each constructor.
- To define it on a constructor, we give an expression like the right hand side of a $\lambda$-expression, except that if some argument $w$ to the constructor is of type $W$ (or a more general situation we shall see later), then we can use $f(w)$ in forming the right hand side.
- For instance, for $f : \mathbb{N} \to X$, we can define $f(succn) = m$, with $m$ a term formed using all the terms in the context, the term $n : \mathbb{N}$ and the term $f(n) : X$. Thus the data giving $f(succ\_)$ is a term of type $\mathbb{N} \to X \to X$, which we apply to $n$ and then $f(n)$.

# Recursion and Induction : a first look

- We can define a function on an inductive type $W$ by defining it on each constructor.
- To define it on a constructor, we give an expression like the right hand side of a $\lambda$-expression, except that if some argument $w$ to the constructor is of type $W$ (or a more general situation we shall see later), then we can use $f(w)$ in forming the right hand side.
- For instance, for $f : \mathbb{N} \to X$, we can define $f(succn) = m$, with $m$ a term formed using all the terms in the context, the term $n : \mathbb{N}$ and the term $f(n) : X$. Thus the data giving $f(succ\_)$ is a term of type $\mathbb{N} \to X \to X$, which we apply to $n$ and then $f(n)$.
- Similarly, for a type $A$, when defining $f : List(A) \to X$, we can define $f(cons(a)(l))$ in terms of $a$, $l$ and $f(l)$. Thus the data giving $f(cons(a)(l))$ is a term of type $A \to List(A) \to X \to X$, which we apply to $a$, then $l$ and finally $f(l)$.

# Recursion and Induction : a first look

- We can define a function on an inductive type $W$ by defining it on each constructor.
- To define it on a constructor, we give an expression like the right hand side of a $\lambda$-expression, except that if some argument $w$ to the constructor is of type $W$ (or a more general situation we shall see later), then we can use $f(w)$ in forming the right hand side.
- For instance, for $f : \mathbb{N} \to X$, we can define $f(succ n) = m$, with $m$ a term formed using all the terms in the context, the term $n : \mathbb{N}$ and the term $f(n) : X$. Thus the data giving $f(succ\_)$ is a term of type $\mathbb{N} \to X \to X$, which we apply to $n$ and then $f(n)$.
- Similarly, for a type $A$, when defining $f : List(A) \to X$, we can define $f(cons(a)(l))$ in terms of $a$, $l$ and $f(l)$. Thus the data giving $f(cons(a)(l))$ is a term of type $A \to List(A) \to X \to X$, which we apply to $a$, then $l$ and finally $f(l)$.
- The analogue of recursive definitions for defining dependnet functions are called inductive definitions.

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.
- We also have identities saying that function built from $rec_{W,X}$, when applied to the data for a constructor, has the appropriate value.

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.
- We also have identities saying that function built from $rec_{W,X}$, when applied to the data for a constructor, has the appropriate value.
- For instance, for $W = \mathbb{N}$ and a type $X$, the data for the constructor 0 is $f(0) : X$, while the data for the constructor *succ* is $\mathbb{N} \to X \to X$ (as we have seen).

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.
- We also have identities saying that function built from $rec_{W,X}$, when applied to the data for a constructor, has the appropriate value.
- For instance, for $W = \mathbb{N}$ and a type $X$, the data for the constructor 0 is $f(0) : X$, while the data for the constructor *succ* is $\mathbb{N} \to X \to X$ (as we have seen).
- Thus, $rec_{\mathbb{N},X}$ has type $X \to (\mathbb{N} \to X \to X) \to (\mathbb{N} \to X)$.

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.
- We also have identities saying that function built from $rec_{W,X}$, when applied to the data for a constructor, has the appropriate value.
- For instance, for $W = \mathbb{N}$ and a type $X$, the data for the constructor 0 is $f(0) : X$, while the data for the constructor *succ* is $\mathbb{N} \to X \to X$ (as we have seen).
- Thus, $rec_{\mathbb{N},X}$ has type $X \to (\mathbb{N} \to X \to X) \to (\mathbb{N} \to X)$.
- For the constructor applied to 0, we get the identity $rec_{\mathbb{N},X}(z)(g)(0) \equiv z$.

# Recursion functions

- In Homotopy Type Theory, recursive definitions are formalized by giving rules for forming a function $rec_{W,X}$ for an inductive type $W$ and a type $X$, which when applied to the data for recursive definition for each constructor gives a function $W \to X$.
- We also have identities saying that function built from $rec_{W,X}$, when applied to the data for a constructor, has the appropriate value.
- For instance, for $W = \mathbb{N}$ and a type $X$, the data for the constructor 0 is $f(0) : X$, while the data for the constructor *succ* is $\mathbb{N} \to X \to X$ (as we have seen).
- Thus, $rec_{\mathbb{N},X}$ has type $X \to (\mathbb{N} \to X \to X) \to (\mathbb{N} \to X)$.
- For the constructor applied to 0, we get the identity $rec_{\mathbb{N},X}(z)(g)(0) \equiv z$.
- For a term $n : \mathbb{N}$, the constructor applied to $succ(n)$ gives the identity $rec_{\mathbb{N},X}(z)(g)(succ(n)) \equiv g(n)(rec_{\mathbb{N},X}(z)(g)(n))$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
  - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
  - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.
- We shall call the type of a famiy of $W$ a family-type for $W$. Family types are types of the form:

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
  - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.
- We shall call the type of a famiy of $W$ a family-type for $W$. Family types are types of the form:
  - $W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
  - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.
- We shall call the type of a famiy of $W$ a family-type for $W$. Family types are types of the form:
  - $W$.
  - $A \to W'$ where $W'$ is a family-type $W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
    - a term $w : W$.
    - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
    - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.
- We shall call the type of a famiy of $W$ a family-type for $W$. Family types are types of the form:
    - $W$.
    - $A \to W'$ where $W'$ is a family-type $W$.
    - $\Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family-type for $W$.

# Families

- For a type $W$, a *family* of terms of $W$ is one of:
  - a term $w : W$.
  - a function $\varphi : A \to W'$ where, for each $a : A$, $\varphi(a) : W'$ is a family of terms of $W$.
  - a dependent function $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $\varphi(a) : W'(a)$ is a family of terms of $W$.
- We shall call the type of a famiy of $W$ a family-type for $W$. Family types are types of the form:
  - $W$.
  - $A \to W'$ where $W'$ is a family-type $W$.
  - $\Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family-type for $W$.
- We can recursively define a *member of a family*, which is a term of type $W$.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
  - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
  - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.
  - for $\varphi : A \to W'$ where $W'$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
  - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.
  - for $\varphi : A \to W'$ where $W'$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
  - for $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
  - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.
  - for $\varphi : A \to W'$ where $W'$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
  - for $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
- This gives functions, or dependent functions, on any given family-types.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
    - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.
    - for $\varphi : A \to W'$ where $W'$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
    - for $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
- This gives functions, or dependent functions, on any given family-types.
- We can use the same definition for dependent functions $f$.

# Induced functions on families

- Suppose $f : W \to X$ is a function, and $\varphi$ is a family of terms of $W$, then we can define $f_*(\varphi)$ as follows.
    - for $\varphi = w$ with $w : W$, $f_*(\varphi) = f(w)$.
    - for $\varphi : A \to W'$ where $W'$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
    - for $\varphi : \Pi_{a:A} W'(a)$ where, for each $a : A$, $W'(a)$ is a family of terms of $W$, define $f_*(\varphi) = (a : A) \mapsto f_*(\varphi(a))$.
- This gives functions, or dependent functions, on any given family-types.
- We can use the same definition for dependent functions $f$.
- In all cases, the type of the induced function on a family-type $W'$ depends only on the type $F$ of $f$. We denote this $Ind_F W'$

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.
  - $T = \Pi_{a:A} T'(a)$, each $T'(a)$ a constructor type for $W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.
  - $T = \Pi_{a:A} T'(a)$, each $T'(a)$ a constructor type for $W$.
  - $T = \Pi_{w:W} T'(w)$, each $T'(w)$ a constructor type for $W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.
  - $T = \Pi_{a:A} T'(a)$, each $T'(a)$ a constructor type for $W$.
  - $T = \Pi_{w:W} T'(w)$, each $T'(w)$ a constructor type for $W$.
  - $T = W' \to T'$, $W'$ a family-type for $W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.
  - $T = \Pi_{a:A} T'(a)$, each $T'(a)$ a constructor type for $W$.
  - $T = \Pi_{w:W} T'(w)$, each $T'(w)$ a constructor type for $W$.
  - $T = W' \to T'$, $W'$ a family-type for $W$.
  - $T = \Pi_{w:W'} T'(w)$, $W'$ family-type for $W$.

# Constructor types for an inductive type

- The constructors of an inductive type $W$ must (and can) be terms with type $T$ a so-called *Constructor type* for $W$, which is one of the following:
  - $T = W$.
  - $T = A \to T'$, where $T'$ is a constructor-type for $W$ and $A$ is a type can be formed from the terms in the context not including $W$.
  - $T = W \to T'$, $T'$ as above.
  - $T = \Pi_{a:A} T'(a)$, each $T'(a)$ a constructor type for $W$.
  - $T = \Pi_{w:W} T'(w)$, each $T'(w)$ a constructor type for $W$.
  - $T = W' \to T'$, $W'$ a family-type for $W$.
  - $T = \Pi_{w:W'} T'(w)$, $W'$ family-type for $W$.
- We call a term of a constructor type for $W$ a quasi-constructor for $W$.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
  - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
  - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{a:A} R_{W,X}(\varphi(a))$.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
    - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.
    - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{a:A} R_{W,X}(\varphi(a))$.
    - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{w:W}(X \to R_{W,X}(\varphi(a)))$.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
  - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{a:A} R_{W,X}(\varphi(a))$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{w:W}(X \to R_{W,X}(\varphi(a)))$.
  - If $\varphi : \Pi_{a:A} W'$, with $W'$ a family-type for $W$, then $R_{W,X}(\varphi) = \Pi_{w:W'}(Ind_{W \to X}(W') \to R_{W,X}(\varphi(a)))$.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
  - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{a:A} R_{W,X}(\varphi(a))$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{w:W}(X \to R_{W,X}(\varphi(a)))$.
  - If $\varphi : \Pi_{a:A} W'$, with $W'$ a family-type for $W$, then $R_{W,X}(\varphi) = \Pi_{w:W'}(Ind_{W\to X}(W') \to R_{W,X}(\varphi(a)))$.
- Domains of Induction are similar.

# Domains of recursion

- We shall associate to any quasi-constructor $\varphi$ for $W$ a type $R_{W,X}(\varphi)$ which we call the domain of recursion.
- This can be defined in all cases for the type of $\varphi$. We give only the dependent function cases below.
  - If $\varphi : W$, then $R_{W,X}(\varphi) = X$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{a:A} R_{W,X}(\varphi(a))$.
  - If $\varphi : \Pi_{a:A} W$, then $R_{W,X}(\varphi) = \Pi_{w:W}(X \to R_{W,X}(\varphi(a)))$.
  - If $\varphi : \Pi_{a:A} W'$, with $W'$ a family-type for $W$, then $R_{W,X}(\varphi) = \Pi_{w:W'}(Ind_{W \to X}(W') \to R_{W,X}(\varphi(a)))$.
- Domains of Induction are similar.
- We now see examples.

# Recursion functions

- The types and identities of recursion functions can be built from the domains of recursion of the constructors.

# Recursion functions

- The types and identities of recursion functions can be built from the domains of recursion of the constructors.
- Namely, if an inductive type has constructors $g_1, g_2, \ldots, g_k$, then $rec_{W,X}$ has type

$$R_{W,X}(g_1) \to R_{W,X}(g_2) \to \cdots \to R_{W,X}(g_n) \to (W \to X).$$

# Recursion functions

- The types and identities of recursion functions can be built from the domains of recursion of the constructors.
- Namely, if an inductive type has constructors $g_1, g_2, \dots, g_k$, then $rec_{W,X}$ has type

$$R_{W,X}(g_1) \to R_{W,X}(g_2) \to \cdots \to R_{W,X}(g_n) \to (W \to X).$$

- We get identities for each constructor recursively.

# Inductive type families

- A type family $\tilde{W}$ is a family of terms of a universe $\mathcal{U}$.

# Inductive type families

- A type family $\tilde{W}$ is a family of terms of a universe $\mathcal{U}$.
- We can define constructor types for $\tilde{W}$ analogous to those for a type $W$, except that we replace all instances of $W$ by members of the family $\tilde{W}$.

# Inductive type families

- A type family $\tilde{W}$ is a family of terms of a universe $\mathcal{U}$.
- We can define constructor types for $\tilde{W}$ analogous to those for a type $W$, except that we replace all instances of $W$ by members of the family $\tilde{W}$.
- Recursion, induction etc. are similar.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and Π-types

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and $\Pi$-types
- Can apply (dependent) functions to arguments of the right type.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and $\Pi$-types
- Can apply (dependent) functions to arguments of the right type.
- Can define (dependent) functions using $\lambda$-expressions.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and Π-types
- Can apply (dependent) functions to arguments of the right type.
- Can define (dependent) functions using $\lambda$-expressions.
- Can define inductive types and inductive type families by listing constructors, which must be of the appropriate constructor type.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and $\Pi$-types
- Can apply (dependent) functions to arguments of the right type.
- Can define (dependent) functions using $\lambda$-expressions.
- Can define inductive types and inductive type families by listing constructors, which must be of the appropriate constructor type.
- For an inductive type $W$ and a type $X$ (or type family on $W$), we have recursion/induction functions.

# Rules for forming terms

We now can list all the rules for forming terms.

- Universes: given in advance.
- Can form function types and Π-types
- Can apply (dependent) functions to arguments of the right type.
- Can define (dependent) functions using $\lambda$-expressions.
- Can define inductive types and inductive type families by listing constructors, which must be of the appropriate constructor type.
- For an inductive type $W$ and a type $X$ (or type family on $W$), we have recursion/induction functions.
- Finally, we can simply introduce a term with a given type as an *axiom*.