Prety much a summary of the 1st chapter

The “Homotopy Type Theory” book (a.k.a. The Univalent Foundations Program)

Andreas Avoukatos

Algorithms, Logic and Discrete Mathematics, DIT @ UOA

2025

Table of Contents

Table of Contents

Classic vs Construcivistic Mathematics

Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. [...]

Classic vs Construcivistic Mathematics

Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. [...]

Classic vs Construcivistic Mathematics

Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. [...] For, compared with the immense expanse of modern mathematics, what would the wretched remnants mean, the few isolated results, incomplete and unrelated, that the intuitionists have obtained?

David Hilbert, 1927

Classic vs Construcivistic Mathematics

> *Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. [...] For, compared with the immense expanse of modern mathematics, what would the wretched remnants mean, the few isolated results, incomplete and unrelated, that the intuitionists have obtained?*
>
> *David Hilbert, 1927*

- **All-or-nothing** approach: *only* constructive proofs are correct (and all others are *illusory*), or non-constructive proofs are valid, *occasionally* interesting / valuable, (but of **zero** philosophical importance)

Classic vs Construcivistic Mathematics

> *Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. [...] For, compared with the immense expanse of modern mathematics, what would the wretched remnants mean, the few isolated results, incomplete and unrelated, that the intuitionists have obtained?*
>
> *David Hilbert, 1927*

- **All-or-nothing** approach: *only* constructive proofs are correct (and all others are *illusory*), or non-constructive proofs are valid, *occasionally* interesting / valuable, (but of **zero** philosophical importance)
- No **productive interplay** between these camps

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967).

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both "camps" can be viewed as generalisation of his work

Exciting development

Erret Bishop writes “Foundations of Constructive Analysis” (1967). What’s new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both “camps” can be viewed as generalisation of his work

Exciting development

Erret Bishop writes “Foundations of Constructive Analysis” (1967). What’s new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both “camps” can be viewed as generalisation of his work

What followed:

- **intuitionistic set theories** got developed,

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both "camps" can be viewed as generalisation of his work

What followed:

- **intuitionistic set theories** got developed,
- Bishop's worked got **extended**:

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both "camps" can be viewed as generalisation of his work

What followed:

- **intuitionistic set theories** got developed,
- Bishop's worked got **extended**:
  - "Constructive Functional Analysis" [Bridges, 1979]

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both "camps" can be viewed as generalisation of his work

What followed:

- **intuitionistic set theories** got developed,
- Bishop's worked got **extended**:
  - "Constructive Functional Analysis" [Bridges, 1979]
  - "Varieties of Constructive Mathematics" [Bridges, Richman, 1981]

Exciting development

Erret Bishop writes "Foundations of Constructive Analysis" (1967). What's new:

- unlike previous attempts, **large swaths** of mathematics were **constructively developed**, with **minor changes** from classical theory
- worked in the **common ground**, s.t. both "camps" can be viewed as generalisation of his work

What followed:

- **intuitionistic set theories** got developed,
- Bishop's worked got **extended**:
  - "Constructive Functional Analysis" [Bridges, 1979]
  - "Varieties of Constructive Mathematics" [Bridges, Richman, 1981]
  - "A Course in Constructive Algebra" [Mines, Richman, Ruitenberg, 1988]

[...] Namely both of them [Hilbert and Brouwer] thought that if one took constructive mathematics seriously, it would be necessary to "give up" the most important parts of modern mathematics (such as, for example, measure theory or complex analysis).

[...] Namely both of them [Hilbert and Brouwer] thought that if one took constructive mathematics seriously, it would be necessary to "give up" the most important parts of modern mathematics (such as, for example, measure theory or complex analysis).

[...] Namely both of them [Hilbert and Brouwer] thought that if one took constructive mathematics seriously, it would be necessary to "give up" the most important parts of modern mathematics (such as, for example, measure theory or complex analysis). Bishop showed that this was simply false, and in addition that it is not necessary to introduce unusual assumptions that appear contradictory to the uninitiated. [...]

[...] Namely both of them [Hilbert and Brouwer] thought that if one took constructive mathematics seriously, it would be necessary to "give up" the most important parts of modern mathematics (such as, for example, measure theory or complex analysis). Bishop showed that this was simply false, and in addition that it is not necessary to introduce unusual assumptions that appear contradictory to the uninitiated. [...] One only had to proceed with a certain grace, instead of with Hilbert's "boxer's fists".

Michael Beeson

Table of Contents

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

There is **interplay** between the objects of the 2nd layer (**sets**) and the objects of the 1st layer (**propositions**).

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

There is **interplay** between the objects of the 2nd layer (**sets**) and the objects of the 1st layer (**propositions**).

Type theory is its own deductive system, and it consists of:

- **types**

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

There is **interplay** between the objects of the 2nd layer (**sets**) and the objects of the 1st layer (**propositions**).

Type theory is its own deductive system, and it consists of:

- **types**

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

There is **interplay** between the objects of the 2nd layer (**sets**) and the objects of the 1st layer (**propositions**).

Type theory is its own deductive system, and it consists of:

- **types**

Propositions are identified with **particular types**.

Main differences

Set theory consists of:

- the **deductive system** of First Order Logic
- the **axioms** of a theory, ZFC, (inside of the deductive system)

There is **interplay** between the objects of the 2nd layer (**sets**) and the objects of the 1st layer (**propositions**).

Type theory is its own deductive system, and it consists of:

- **types**

Propositions are identified with **particular types**.
Proving a theorem coincides with with **constructing** an object (an inhabitant) of a type.

Deductive systems

A collection of **rules**, for deriving **judgments**.

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Type Theory's judgments: "$a : A$", read as

- "the **term** *a* has type A" / "*a* is an element of type A"

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Type Theory's judgments: "$a : A$", read as

- "the **term** a has type A" / "a is an element of type A"
- "a is a point of the space A" (*in Homotopy Type Theory*)

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Type Theory's judgments: "$a : A$", read as

- "the **term** a has type A" / "a is an element of type A"
- "a is a point of the space A" (*in Homotopy Type Theory*)
- "a is a witness (or evidence of truth) of A" (*when A is a proposition*)

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Type Theory's judgments: "$a : A$", read as

- "the **term** a has type A" / "a is an element of type A"
- "a is a point of the space A" (*in Homotopy Type Theory*)
- "a is a witness (or evidence of truth) of A" (*when A is a proposition*)

Deductive systems

A collection of **rules**, for deriving **judgments**.

FOL's judgment: "a given proposition has a proof".
Example of a rule: "from A and B infer $A \wedge B$".

Type Theory's judgments: "$a : A$", read as

- "the **term** *a* has type A" / "*a* is an element of type A"
- "*a* is a point of the space A" (*in Homotopy Type Theory*)
- "*a* is a witness (or evidence of truth) of A" (*when A is a proposition*)

The judgement "$a : A$" is derivable **in type theory**, precisely **when** "A has a proof" is **derivable in FOL**.

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects “a” and “A”

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects "a" and "A"
- "let $x \in \mathbb{N}$" **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects "a" and "A"
- "let $x \in \mathbb{N}$" **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects “a” and “A”
- “let $x \in \mathbb{N}$” **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Type Theory:

- the statement “let $x : \mathbb{N}$” is **atomic**

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects “a” and “A”
- “let $x \in \mathbb{N}$” **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Type Theory:

- the statement “let $x : \mathbb{N}$” is **atomic**

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects “a” and “A”
- “let $x \in \mathbb{N}$” **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Type Theory:

- the statement “let $x : \mathbb{N}$” is **atomic**

Equality is a type, that is for $a, b : A$, we have a type $a =_A b$.

Membership and equality

Set theory:

- membership **may** (or **may not**) hold between two pre-existing objects "a" and "A"
- "let $x \in \mathbb{N}$" **introduces** an object x and **assumes** that $x \in \mathbb{N}$

Type Theory:

- the statement "let $x\colon \mathbb{N}$" is **atomic**

Equality is a type, that is for $a, b\colon A$, we have a type $a =_A b$. When $a =_A b$ is **inhabited**, we say that a and b are **(propositionally) equal**.

Judgmental vs propositional equality

The equality at the same level as "$x : A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

Judgmental vs propositional equality

The equality at the same level as "$x: A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a: A$".

Judgmental vs propositional equality

The equality at the same level as "$x : A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a : A$".

Example

Suppose we define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$,

Judgmental vs propositional equality

The equality at the same level as "$x : A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a : A$".

Example

Suppose we define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$,

Judgmental vs propositional equality

The equality at the same level as "$x: A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a: A$".

Example

Suppose we define $f: \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*.

Judgmental vs propositional equality

The equality at the same level as "$x : A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a : A$".

Example

Suppose we define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*. Imagine we have derived the **judgment** "$p : 3^2 = 9$".

Judgmental vs propositional equality

The equality at the same level as "$x : A$" is called **judgmental** or **defintional equality**,

$$a \equiv b : A$$

- we want to **control** the other form of judgement, "$a : A$".

Example

Suppose we define $f : \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*. Imagine we have derived the **judgment** "$p : 3^2 = 9$". What about "$\mathrm{f}(3) = 9$"?

Judgmental vs propositional equality

The equality at the same level as "$x \colon A$" is called **judgmental** or **defintional equality**,

$$a \equiv b \colon A$$

- we want to **control** the other form of judgement, "$a \colon A$".

Example

Suppose we define $f \colon \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*. Imagine we have derived the **judgment** "$p \colon 3^2 = 9$". What about "f(3) = 9"? Since $f(3)$ is 3^2 by definition, p should count as proof that $f(3) = 9$.

Judgmental vs propositional equality

The equality at the same level as "$x \colon A$" is called **judgmental** or **defintional equality**,

$$a \equiv b \colon A$$

- we want to **control** the other form of judgement, "$a \colon A$".

Example

Suppose we define $f \colon \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*. Imagine we have derived the **judgment** "$p \colon 3^2 = 9$". What about "f(3) = 9"? Since $f(3)$ is 3^2 by definition, p should count as proof that $f(3) = 9$.

As a rule

Judgmental vs propositional equality

The equality at the same level as “$x\colon A$” is called **judgmental** or **defintional equality**,

$$a \equiv b \colon A$$

- we want to **control** the other form of judgement, “$a\colon A$”.

Example

Suppose we define $f\colon \mathbb{N} \to \mathbb{N}$ by $f(x) = x^2$, then $f(3)$ is equal to 3^2 *by definition*. Imagine we have derived the **judgment** “$p\colon 3^2 = 9$”. What about “f(3) = 9”? Since $f(3)$ is 3^2 by definition, p should count as proof that $f(3) = 9$.

As a rule

- Given $a\colon A$ and $A \equiv B$, we derive $a\colon B$

Recap

There are **two forms** of judgment,

- $a : A$ (a is an **object** of **type** A)

Recap

There are **two forms** of judgment,

- $a : A$ (a is an **object** of **type** A)
- $a \equiv b : A$ (a and b are **definitionally equal** objects of type A)

Recap

There are **two forms** of judgment,

- $a : A$ (a is an **object** of **type** A)
- $a \equiv b : A$ (a and b are **definitionally equal** objects of type A)

Recap

There are **two forms** of judgment,

- $a : A$ (a is an **object** of **type** A)
- $a \equiv b : A$ (a and b are **definitionally equal** objects of type A)

(This) Type Theory:

- consists **entirely** of rules

Recap

There are **two forms** of judgment,

- $a : A$ (a is an **object** of **type** A)
- $a \equiv b : A$ (a and b are **definitionally equal** objects of type A)

(This) Type Theory:

- consists **entirely** of rules
- has **zero** axioms

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Type theory:

- The **rules** contain all the **information** (usually), with no axioms being necessary.

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Type theory:

- The **rules** contain all the **information** (usually), with no axioms being necessary.

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Type theory:

- The **rules** contain all the **information** (usually), with no axioms being necessary.

Pros:

- **rules are procedural**, which make possible (but *don't automatically* ensure) **good computational properties** of type theory, such as canonicity

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Type theory:

- The **rules** contain all the **information** (usually), with no axioms being necessary.

Pros:

- **rules are procedural**, which make possible (but *don't automatically* ensure) **good computational properties** of type theory, such as canonicity

Recap

Set theory:

- **Axioms** contain all the **information** about the behavior of sets

Type theory:

- The **rules** contain all the **information** (usually), with no axioms being necessary.

Pros:

- **rules are procedural**, which make possible (but *don't automatically* ensure) **good computational properties** of type theory, such as canonicity

Cons:

- we do not understand **how to formulate everything** we need. For homotopy type theory, we will **have to augment** the rules of type theory, notably the **univalence axiom**

Table of Contents

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**
- Type theory: **primitive** concept

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**
- Type theory: **primitive** concept

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**
- Type theory: **primitive** concept

Behaviour?

We explain the type by prescribing **what we can do** with its objects, **how to construct** them, what *equalities* they *induce* and so on.

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**
- Type theory: **primitive** concept

Behaviour?

We explain the type by prescribing **what we can do** with its objects, **how to construct** them, what *equalities* they *induce* and so on.

Function Types (1/3)

Given types A, B, we construct the type $A \to B$ of functions with domain A and codomain B.

- Set theory: functions are defined as **functional relations**
- Type theory: **primitive** concept

Behaviour?

We explain the type by prescribing **what we can do** with its objects, **how to construct** them, what *equalities* they *induce* and so on.

Usage

Given $f : A \to B$, $a : A$, we can **apply** the function to **obtain an element** of the codomain B, denoted $f(a)$, also written as $f\ a$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Example

Let f be of type $\mathbb{N} \to \mathbb{N}$, defined by $f(x) :\equiv x + x$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Example

Let f be of type $\mathbb{N} \to \mathbb{N}$, defined by $f(x) :\equiv x + x$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Example

Let f be of type $\mathbb{N} \to \mathbb{N}$, defined by $f(x) :\equiv x + x$. Then $f(2)$ is judgmentaly equal to $2 + 2$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Example

Let f be of type $\mathbb{N} \to \mathbb{N}$, defined by $f(x) :\equiv x + x$. Then $f(2)$ is judgmentaly equal to $2 + 2$. Similarly, $(\lambda(x : \mathbb{N}).x + x) : \mathbb{N} \to \mathbb{N}$.

Function Types, definitions (2/3)

Construction of elements of $A \to B$

- Direct definition: provide $f(x) :\equiv \Phi$ and check that $\Phi : B$, assuming $x : A$.
- Lambda abstraction: given an expression Φ of type B, which may use $x : A$, write $\lambda(x : A).\Phi$ to indicate the same function as $f(x) :\equiv \Phi$

Example

Let f be of type $\mathbb{N} \to \mathbb{N}$, defined by $f(x) :\equiv x + x$. Then $f(2)$ is judgmentaly equal to $2 + 2$. Similarly, $(\lambda(x : \mathbb{N}).x + x) : \mathbb{N} \to \mathbb{N}$.

Remark

In the lambda abstraction, we can **skip the domain** since it's **infered** in the type, that is $\lambda x.\Phi : A \to B$.

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured.

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured.

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured. (E.g.: given $f(x) :\equiv \lambda y.x + y$,

- $f(y) \equiv \lambda y.y + y$ - (not possible), $f(y) \equiv \lambda z.y + z$ - (acceptable)

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured. (E.g.: given $f(x) :\equiv \lambda y.x + y$,

- $f(y) \equiv \lambda y.y + y$ - (not possible), $f(y) \equiv \lambda z.y + z$ - (acceptable)

More inputs?

More functions: $f : A \times B \to C[?]$

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured. (E.g.: given $f(x) :\equiv \lambda y.x + y$,

- $f(y) \equiv \lambda y.y + y$ - (not possible), $f(y) \equiv \lambda z.y + z$ - (acceptable)

More inputs?

More functions: $f : A \times B \to C$[?]

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured. (E.g.: given $f(x) :\equiv \lambda y.x + y$,

- $f(y) \equiv \lambda y.y + y$ - (not possible), $f(y) \equiv \lambda z.y + z$ - (acceptable)

More inputs?

More functions: $f : A \times B \to C$[?] ✓$f : A \to (B \to C)$
$f(x)(y)$[?]

Computation rule (aka β-conversion / reduction)

$(\lambda x.\Phi)(a) \equiv \Phi'$, where Φ' is the expression Φ with x having been replaced by a. (E.g.: $(\lambda x.x + x)(2) \equiv 2 + 2$)

Uniqueness principle (aka η-conversion / expansion)

Given $f : A \to B$, we can construct a lambda $\lambda x.f(x)$ and we consider it definitionally equal to f: $f \equiv (\lambda x.f(x))$

Dummy variables (aka α-conversion)

As per usual we are careful about variables getting captured. (E.g.: given $f(x) :\equiv \lambda y.x + y$,

- $f(y) \equiv \lambda y.y + y$ - (not possible), $f(y) \equiv \lambda z.y + z$ - (acceptable)

More inputs?

More functions: $f : A \times B \to C$[?] ✓$f : A \to (B \to C)$
$f(x)(y)$[?]✓$f(x, y) :\equiv \Phi$, in λ-notation, $f :\equiv \lambda x.\lambda y.\Phi$

Universes (1/2)

A **universe** is a type whose **elements are types**.

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself?

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ \ldots,$

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$, and we assume that our universes are cumulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$.

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$, and we assume that our universes are cumulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$.

- **Convinient** (avoids Girard's paradox, compatibility with categorical semantics, (Grothendieck universes))

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$, and we assume that our universes are cumulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$.

- **Convinient** (avoids Girard's paradox, compatibility with categorical semantics, (Grothendieck universes))
- But, elements **no** longer have **unique types** (which complicates algorithms for inferring and checking types)

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ \ldots$, and we assume that our universes are cumulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$.

- **Convinient** (avoids Girard's paradox, compatibility with categorical semantics, (Grothendieck universes))
- But, elements **no** longer have **unique types** (which complicates algorithms for inferring and checking types)

Universes (1/2)

A **universe** is a type whose **elements are types**.
(Can there be a $\mathcal{U}_\infty$ that includes itself? As in set theory, no.)

To avoid this, we introduce a hierarchy $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$, and we assume that our universes are cumulative, i.e. if $A : \mathcal{U}_i$ then $A : \mathcal{U}_{i+1}$.

- **Convinient** (avoids Girard's paradox, compatibility with categorical semantics, (Grothendieck universes))
- But, elements **no** longer have **unique types** (which complicates algorithms for inferring and checking types)

When we say A is a type, we mean that it **inhabits** some **universe** $\mathcal{U}_i$.

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A : \mathcal{U}$ (meaning $A : \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A : \mathcal{U}$ (meaning $A : \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

- Convenient

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A : \mathcal{U}$ (meaning $A : \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

- Convenient
- But, it can be a bit dangerous

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A : \mathcal{U}$ (meaning $A : \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

- Convenient
- But, it can be a bit dangerous

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A : \mathcal{U}$ (meaning $A : \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

- Convenient
- But, it can be a bit dangerous

In case of **ambiguity** (eg during a proof that seemingly reproduces self-referencial arguments), the way to **check** is to try to **assign levels** consistently to **all** universes appearing in it.

Universes (2/2)

Typical Ambiguity: we omit i, and **assume** that levels can be **assigned** in a **consistent** way.

We may write $A\colon \mathcal{U}$ (meaning $A\colon \mathcal{U}_i$), $\mathcal{U} : \mathcal{U}$ ($\mathcal{U}_i : \mathcal{U}_{i+1}$).

- Convenient
- But, it can be a bit dangerous

In case of **ambiguity** (eg during a proof that seemingly reproduces self-referencial arguments), the way to **check** is to try to **assign levels** consistently to **all** universes appearing in it.

When **some universe** $\mathcal{U}$ is **assumed**, we may refer to the types belonging to $\mathcal{U}$ as **small types**.

Families (1/1)

What is a **family of types** (or dependent types) over a given type A?

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B: A \to \mathcal{U}$, whose **codomain** is a **universe**.

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B: A \to \mathcal{U}$, whose **codomain** is a **universe**.(Corresponding notion in Set Theory, families of sets.)

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B\colon A \to \mathcal{U}$, whose **codomain** is a **universe**.(Corresponding notion in Set Theory, families of sets.)

Examples

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B : A \to \mathcal{U}$, whose **codomain** is a **universe**.(Corresponding notion in Set Theory, families of sets.)

Examples

- $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, where $\mathrm{Fin}(n)$ is a type with exactly n elements

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B\colon A \to \mathcal{U}$, whose **codomain** is a **universe**.(Corresponding notion in Set Theory, families of sets.)

Examples

- $\mathrm{Fin}\colon \mathbb{N} \to \mathcal{U}$, where $\mathrm{Fin}(n)$ is a type with exactly n elements
- a constant type family, given $B\colon \mathcal{U}$, $(\lambda(x\colon A).B)\colon A \to \mathcal{U}$

Families (1/1)

What is a **family of types** (or dependent types) over a given type A? A **function** $B\colon A \to \mathcal{U}$, whose **codomain** is a **universe**.(Corresponding notion in Set Theory, families of sets.)

Examples

- $\mathrm{Fin}\colon \mathbb{N} \to \mathcal{U}$, where $\mathrm{Fin}(n)$ is a type with exactly n elements
- a constant type family, given $B\colon \mathcal{U}$, $(\lambda(x\colon A).B)\colon A \to \mathcal{U}$

Non-example

$(\lambda(i\colon \mathbb{N}).\mathcal{U}_i)$ - there is no universe large enough to be its codomain, we do not even identify the indices i with the naturals.

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**,

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**,

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied;

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Formation rule:

Given a type $A : \mathcal{U}$ and a family $B : A \to U$, we may construct the type $\prod_{(x:A)} B(x) : \mathcal{U}$. (If B is a constant family, $\prod_{(x:A)} B$ is just $A \to B$.)

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Formation rule:

Given a type $A : \mathcal{U}$ and a family $B : A \to U$, we may construct the type $\prod_{(x:A)} B(x) : \mathcal{U}$. (If B is a constant family, $\prod_{(x:A)} B$ is just $A \to B$.)

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Formation rule:

Given a type $A : \mathcal{U}$ and a family $B : A \to U$, we may construct the type $\prod_{(x:A)} B(x) : \mathcal{U}$. (If B is a constant family, $\prod_{(x:A)} B$ is just $A \to B$.)

Introduction rule (construction of dependent functions):

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Formation rule:

Given a type $A : \mathcal{U}$ and a family $B : A \to U$, we may construct the type $\prod_{(x:\,A)} B(x) : \mathcal{U}$. (If B is a constant family, $\prod_{(x:\,A)} B$ is just $A \to B$.)

Introduction rule (construction of dependent functions):

- explicitly: in order to define $f : \prod_{(x:\,A)} B(x)$ we need an expression $\Phi : B(x)$, and we write $f(x) :\equiv \Phi$, for $x : A$

Dependent function types ($\prod$-types), (1/3)

The **elements** of such a type are **functions**, whose **codomain type** can vary **depending** on the **element** of the **domain**, to which the function is applied; these are called **dependent functions**.

Formation rule:

Given a type $A : \mathcal{U}$ and a family $B : A \to U$, we may construct the type $\prod_{(x:\,A)} B(x) : \mathcal{U}$. (If B is a constant family, $\prod_{(x:\,A)} B$ is just $A \to B$.)

Introduction rule (construction of dependent functions):

- explicitly: in order to define $f : \prod_{(x:\,A)} B(x)$ we need an expression $\Phi : B(x)$, and we write $f(x) :\equiv \Phi$, for $x : A$
- λ-abstracton: $\lambda x.\Phi : \Pi(x : A).B(x)$

Dependent function types ($\prod$-types), (2/3)

Usage:

We can **apply** a dependent function $f : \Pi(x : A).B(x)$, to a **term** $a : A$, to **get the value** $f(a) : B(a)$. As previously, we have the computation rule and the uniqueness principle.

Dependent function types ($\prod$-types), (2/3)

Usage:

We can **apply** a dependent function $f : \Pi(x : A).B(x)$, to a **term** $a : A$, to **get the value** $f(a) : B(a)$. As previously, we have the computation rule and the uniqueness principle.

Example

Recall the family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, whose values are the **standard finite sets** with elements $0_n, 1_n, \ldots, (n-1)_n : \mathrm{Fin}(n)$.

Dependent function types ($\prod$-types), (2/3)

Usage:

We can **apply** a dependent function $f : \Pi(x : A).B(x)$, to a **term** $a : A$, to **get the value** $f(a) : B(a)$. As previously, we have the computation rule and the uniqueness principle.

Example

Recall the family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, whose values are the **standard finite sets** with elements $0_n, 1_n, \ldots, (n-1)_n : \mathrm{Fin}(n)$.

Dependent function types ($\prod$-types), (2/3)

Usage:

We can **apply** a dependent function $f : \Pi(x : A).B(x)$, to a **term** $a : A$, to **get the value** $f(a) : B(a)$. As previously, we have the computation rule and the uniqueness principle.

Example

Recall the family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, whose values are the **standard finite sets** with elements $0_n, 1_n, \ldots, (n-1)_n : \mathrm{Fin}(n)$. We can then introduce

- $\mathrm{fmax} : \prod_{(n : \mathbb{N})} \mathrm{Fin}(n+1)$, which returns the "largest" element of each non-empty finite type, that is $\mathrm{fmax}(n) :\equiv n_{n+1}$

Dependent function types ($\prod$-types), (2/3)

Usage:

We can **apply** a dependent function $f : \Pi(x : A).B(x)$, to a **term** $a : A$, to **get the value** $f(a) : B(a)$. As previously, we have the computation rule and the uniqueness principle.

Example

Recall the family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, whose values are the **standard finite sets** with elements $0_n, 1_n, \ldots, (n-1)_n : \mathrm{Fin}(n)$. We can then introduce

- $\mathrm{fmax} : \prod_{(n : \mathbb{N})} \mathrm{Fin}(n+1)$, which returns the "largest" element of each non-empty finite type, that is $\mathrm{fmax}(n) :\equiv n_{n+1}$
- Similarly we can introduce $\mathrm{fmin}(n) :\equiv 0_{n+1}$.

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

- id: $\prod_{(A:\,\mathcal{U})} A \to A$, defined as id $:\equiv \lambda(A:\mathcal{U}).\lambda(x:A).x$

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

- id: $\prod_{(A:\,\mathcal{U})} A \to A$, defined as id $:\equiv \lambda(A:\mathcal{U}).\lambda(x:A).x$
- swap: $\prod_{(A:\,\mathcal{U})}\prod_{(B:\,\mathcal{U})}\prod_{(C:\,\mathcal{U})}(A \to B \to C) \to (B \to A \to C)$,

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

- id: $\prod_{(A:\,\mathcal{U})} A \to A$, defined as id $:\equiv \lambda(A:\mathcal{U}).\lambda(x:A).x$
- swap: $\prod_{(A:\,\mathcal{U})}\prod_{(B:\,\mathcal{U})}\prod_{(C:\,\mathcal{U})}(A \to B \to C) \to (B \to A \to C)$,

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

- id: $\prod_{(A:\,\mathcal{U})} A \to A$, defined as id $:\equiv \lambda(A:\mathcal{U}).\lambda(x:A).x$
- swap: $\prod_{(A:\,\mathcal{U})}\prod_{(B:\,\mathcal{U})}\prod_{(C:\,\mathcal{U})}(A \to B \to C) \to (B \to A \to C)$, defined as swap$(A, B, C, g) :\equiv \lambda b.\lambda a.g(a)(b)$.

Dependent function types ($\prod$-types), (3/3)

Another **class** of dependent function types, are those who are **polymorphic** over a given universe,that is, they take a **type** as one of its **arguments**, and then **acts on elements** of that type.

(Fancier) Examples

- id: $\prod_{(A:\mathcal{U})} A \to A$, defined as id $:\equiv \lambda(A:\mathcal{U}).\lambda(x:A).x$
- swap: $\prod_{(A:\mathcal{U})}\prod_{(B:\mathcal{U})}\prod_{(C:\mathcal{U})}(A \to B \to C) \to (B \to A \to C)$, defined as $\mathrm{swap}(A, B, C, g) :\equiv \lambda b.\lambda a.g(a)(b)$.
  We allow ourselves to write $\mathrm{swap}_{A,B,C}(g)(b, a) :\equiv g(a, b)$, and swap: $\prod_{(A,B,C:\mathcal{U})} \cdots$

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

- **Formation rules**: how to **form new types** of this kind (e.g.: if A, B are types then $A \to B$ is a type)

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

- **Formation rules**: how to **form new types** of this kind (e.g.: if A, B are types then $A \to B$ is a type)
- **Introduction rules**: how to **construct elements** of that type (e.g. λ-abstraction)

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

- **Formation rules**: how to **form new types** of this kind (e.g.: if A, B are types then $A \to B$ is a type)
- **Introduction rules**: how to **construct elements** of that type (e.g. λ-abstraction)
- **Elimination rules**: how to **use elements** of that type (function application)

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

- **Formation rules**: how to **form new types** of this kind (e.g.: if A, B are types then $A \to B$ is a type)
- **Introduction rules**: how to **construct elements** of that type (e.g. λ-abstraction)
- **Elimination rules**: how to **use elements** of that type (function application)
- **Computation rule**: how an **eliminator acts** on a **constructor** ($(\lambda x.\Phi)(a)$ is judgmentally equal to the substitution of a for x in Φ)

A helpful collection of rules

General **pattern** for **introducing** a **new kind** of type in type theory:

- **Formation rules**: how to **form new types** of this kind (e.g.: if A, B are types then $A \to B$ is a type)
- **Introduction rules**: how to **construct elements** of that type (e.g. λ-abstraction)
- **Elimination rules**: how to **use elements** of that type (function application)
- **Computation rule**: how an **eliminator acts** on a **constructor** ($(\lambda x.\Phi)(a)$ is judgmentally equal to the substitution of a for x in Φ)
- (Optionally) a **uniqueness principle** - expressing uniqueness of maps into or out of that type (f is judgmentally equal to the "expanded" function $\lambda x.f(x)$)

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$.

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$.

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a \colon A$ and $b \colon B$, we may form $(a, b) \colon A \times B$

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$
(There's a unique way to construct elements of $\mathbf{1}$, i.e. $\star : \mathbf{1}$)

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$
(There's a unique way to construct elements of $\mathbf{1}$, i.e. $\star : \mathbf{1}$)

Expectation

"Every element of $A \times B$ is a pair" (aka the uniqueness principle for products). We do not assert this as a rule, but we will prove it later on as a propositional equality.

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$
(There's a unique way to construct elements of $\mathbf{1}$, i.e. $\star : \mathbf{1}$)

Expectation

"Every element of $A \times B$ is a pair" (aka the uniqueness principle for products). We do not assert this as a rule, but we will prove it later on as a propositional equality.

Product types, formation, introduction (1/7)

Formation rules

Given types $A, B : \mathcal{U}$, we introduce the type $A \times B : \mathcal{U}$. A nullary version of the product type, called the unit type, is $\mathbf{1} : \mathcal{U}$.

Introduction rule (how to construct pairs)

Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$
(There's a unique way to construct elements of $\mathbf{1}$, i.e. $\star : \mathbf{1}$)

Expectation

"Every element of $A \times B$ is a pair" (aka the uniqueness principle for products). We do not assert this as a rule, but we will prove it later on as a propositional equality.

Product types, elimination (2/7)

Elimination rule

By providing $g : A \to B \to C$, we can define a function $f : A \times B \to C$ by $f((a, b)) :\equiv g(a)(b)$, (for any such g).

Product types, elimination (2/7)

Elimination rule

By providing $g\colon A \to B \to C$, we can define a function $f\colon A \times B \to C$ by $f((a, b)) :\equiv g(a)(b)$, (for any such g).

- Set theory: we would justify this by the fact that every element of $A \times B$ is an ordered pair, (it suffices to define f on such pairs).

Product types, elimination (2/7)

Elimination rule

By providing $g\colon A \to B \to C$, we can define a function $f\colon A \times B \to C$ by $f((a,b)) :\equiv g(a)(b)$, (for any such g).

- Set theory: we would justify this by the fact that every element of $A \times B$ is an ordered pair, (it suffices to define f on such pairs).
- Type theory: we assume that a function on a $A \times B$ is well-defined as soon as we specify its values on pairs, (this allows us to prove that every element of $A \times B$ is a pair).

Product types, recursor (3/7)

The projection functions,

- $\mathrm{pr}_1 : A \times B \to A$, defined as $\mathrm{pr}_1((a, b)) :\equiv a$

Product types, recursor (3/7)

The projection functions,

- $\mathsf{pr}_1 : A \times B \to A$, defined as $\mathsf{pr}_1((a, b)) :\equiv a$
- $\mathsf{pr}_2 : A \times B \to B$, defined as $\mathsf{pr}_2((a, b)) :\equiv b$

Product types, recursor (3/7)

The projection functions,

- $\mathrm{pr}_1 : A \times B \to A$, defined as $\mathrm{pr}_1((a, b)) :\equiv a$
- $\mathrm{pr}_2 : A \times B \to B$, defined as $\mathrm{pr}_2((a, b)) :\equiv b$

Product types, recursor (3/7)

The projection functions,

- $\mathrm{pr}_1 : A \times B \to A$, defined as $\mathrm{pr}_1((a, b)) :\equiv a$
- $\mathrm{pr}_2 : A \times B \to B$, defined as $\mathrm{pr}_2((a, b)) :\equiv b$

An **alternative** approach: **invoke** the principle **once** (in a universal case), and then simply **apply** the resulting function **in all other cases**.

Product types, recursor (3/7)

The projection functions,

- $\mathsf{pr}_1 : A \times B \to A$, defined as $\mathsf{pr}_1((a, b)) :\equiv a$
- $\mathsf{pr}_2 : A \times B \to B$, defined as $\mathsf{pr}_2((a, b)) :\equiv b$

An **alternative** approach: **invoke** the principle **once** (in a universal case), and then simply **apply** the resulting function **in all other cases**.

Recursor

We may define a function of type
$\mathsf{rec}_{AxB} : \prod_{(C:\mathcal{U})}(A \to B \to C) \to A \times B \to C$, with defining equation

$$\mathsf{rec}_{AxB}(C, g, (a, b)) :\equiv g(a)(b)$$

Product types, recursor (3/7)

The projection functions,

- $\mathrm{pr}_1 : A \times B \to A$, defined as $\mathrm{pr}_1((a, b)) :\equiv a$
- $\mathrm{pr}_2 : A \times B \to B$, defined as $\mathrm{pr}_2((a, b)) :\equiv b$

An **alternative** approach: **invoke** the principle **once** (in a universal case), and then simply **apply** the resulting function **in all other cases**.

Recursor

We may define a function of type
$\mathrm{rec}_{AxB} : \prod_{(C:\mathcal{U})}(A \to B \to C) \to A \times B \to C$, with defining equation

$$\mathrm{rec}_{AxB}(C, g, (a, b)) :\equiv g(a)(b)$$

Product types, recursor (3/7)

The projection functions,

- $\mathrm{pr}_1 : A \times B \to A$, defined as $\mathrm{pr}_1((a,b)) :\equiv a$
- $\mathrm{pr}_2 : A \times B \to B$, defined as $\mathrm{pr}_2((a,b)) :\equiv b$

An **alternative** approach: **invoke** the principle **once** (in a universal case), and then simply **apply** the resulting function **in all other cases**.

Recursor

We may define a function of type
$\mathrm{rec}_{AxB} : \prod_{(C:\mathcal{U})}(A \to B \to C) \to A \times B \to C$, with defining equation

$$\mathrm{rec}_{AxB}(C, g, (a,b)) :\equiv g(a)(b)$$

Then, the projections become

$$\mathrm{pr}_1 :\equiv \mathrm{rec}_{AxB}(A, \lambda a.\lambda b.a), \mathrm{pr}_2 :\equiv \mathrm{rec}_{AxB}(B, \lambda a.\lambda b.b)$$

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place.

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Exercise

Derive $\mathrm{rec}_{A\times B}$ from the projections and vice versa.

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Exercise

Derive $\mathrm{rec}_{A\times B}$ from the projections and vice versa.

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Exercise

Derive $\mathsf{rec}_{A\times B}$ from the projections and vice versa.

Recursor for the unit type

$\mathsf{rec}_{\mathbf{1}} : \prod_{(C\,:\,\mathcal{U})} C \to \mathbf{1} \to C$, with defining equation
$\mathsf{rec}_{\mathbf{1}}(C, c, *) :\equiv c$

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Exercise

Derive $\mathrm{rec}_{A\times B}$ from the projections and vice versa.

Recursor for the unit type

$\mathrm{rec}_{\mathbf{1}} : \prod_{(C\,:\,\mathcal{U})} C \to \mathbf{1} \to C$, with defining equation
$\mathrm{rec}_{\mathbf{1}}(C, c, *) :\equiv c$

Product types, recursor cont. (4/7)

The name recursor is a bit unfortunate, as no recursion is taking place. In inductive types (such as the product types), the recursor is used for defining functions out of a type, (and in types such as the naturals, it will be recursive).

Exercise

Derive $\mathrm{rec}_{A\times B}$ from the projections and vice versa.

Recursor for the unit type

$\mathrm{rec}_{\mathbf{1}} : \prod_{(C:\mathcal{U})} C \to \mathbf{1} \to C$, with defining equation
$\mathrm{rec}_{\mathbf{1}}(C, c, *) :\equiv c$

What would a generalisation of the recursor be?

Product types, dependent functions (5/7)

Dependent functions over the product type

Given $C\colon A \times B \to \mathcal{U}$, we may define a $f\colon \prod_{(x\colon A\times B)} C(x)$, by providing a $g\colon \prod_{(x\colon A)} \prod_{(y\colon B)} C((x,y))$ with defining equation

$$f((x,y)) :\equiv g(x)(y)$$

Product types, dependent functions (5/7)

Dependent functions over the product type

Given $C \colon A \times B \to \mathcal{U}$, we may define a $f \colon \prod_{(x \colon A \times B)} C(x)$, by providing a $g \colon \prod_{(x \colon A)} \prod_{(y \colon B)} C((x, y))$ with defining equation

$$f((x, y)) :\equiv g(x)(y)$$

Product types, dependent functions (5/7)

Dependent functions over the product type

Given $C \colon A \times B \to \mathcal{U}$, we may define a $f \colon \prod_{(x \colon A \times B)} C(x)$, by providing a $g \colon \prod_{(x \colon A)} \prod_{(y \colon B)} C((x, y))$ with defining equation

$$f((x, y)) :\equiv g(x)(y)$$

We can begin the search of an element of the type $\mathsf{uniq}_{A \times B} \colon \prod_{(x \colon A \times B)}((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A \times B} x)$

Product types, dependent functions (5/7)

Dependent functions over the product type

Given $C\colon A \times B \to \mathcal{U}$, we may define a $f\colon \prod_{(x\colon A\times B)} C(x)$, by providing a $g\colon \prod_{(x\colon A)} \prod_{(y\colon B)} C((x,y))$ with defining equation

$$f((x,y)) :\equiv g(x)(y)$$

We can begin the search of an element of the type $\mathsf{uniq}_{A\times B}\colon \prod_{(x\colon A\times B)}((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x)$(aka the propositional uniqueness principle)

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\, A\times B)}((\mathrm{pr}_1(x), \mathrm{pr}_2(x)) =_{A\times B} x))$
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathrm{refl}_x : x =_A x$, for any $x : A$)

How to define $\mathrm{uniq}_{A\times B}((a, b))$?

In the case that $x :\equiv (a, b)$,

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\, A\times B)}((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x)$)
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathsf{refl}_x : x =_A x$, for any $x\colon A$)

How to define $\mathsf{uniq}_{A\times B}((a, b))$?

In the case that $x :\equiv (a, b)$,

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\,A\times B)}((\mathrm{pr}_1(x),\mathrm{pr}_2(x)) =_{A\times B} x))$
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathrm{refl}_x : x =_A x$, for any $x : A$)

How to define $\mathrm{uniq}_{A\times B}((a,b))$?

In the case that $x :\equiv (a,b)$, we can calculate

$$(\mathrm{pr}_1((a,b)),\mathrm{pr}_2((a,b))) \equiv (a,b),$$

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\, A\times B)}((\mathrm{pr}_1(x), \mathrm{pr}_2(x)) =_{A\times B} x))$
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathrm{refl}_x : x =_A x$, for any $x : A$)

How to define $\mathrm{uniq}_{A\times B}((a, b))$?

In the case that $x :\equiv (a, b)$, we can calculate

$$(\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) \equiv (a, b),$$

therefore,

$$\mathrm{refl}_{(a,b)} : (\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) = (a, b)$$

is well-typed,

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\, A\times B)}((\mathrm{pr}_1(x), \mathrm{pr}_2(x)) =_{A\times B} x)$)
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathrm{refl}_x : x =_A x$, for any $x : A$)

How to define $\mathrm{uniq}_{A\times B}((a, b))$?

In the case that $x :\equiv (a, b)$, we can calculate

$$(\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) \equiv (a, b),$$

therefore,

$$\mathrm{refl}_{(a,b)} : (\mathrm{pr}_1((a, b)), \mathrm{pr}_2((a, b))) = (a, b)$$

is well-typed, since both sides are judgmentally equal.

Product types, uniqueness principle (6/7)

(Looking for an element of $\prod_{(x:\,A\times B)}((\mathrm{pr}_1(x),\mathrm{pr}_2(x)) =_{A\times B} x))$
(What we need to know regarding the **identity type**: there is a **reflexivitiy element** $\mathrm{refl}_x : x =_A x$, for any $x : A$)

How to define $\mathrm{uniq}_{A\times B}((a,b))$?

In the case that $x :\equiv (a,b)$, we can calculate

$$(\mathrm{pr}_1((a,b)),\mathrm{pr}_2((a,b))) \equiv (a,b),$$

therefore,

$$\mathrm{refl}_{(a,b)} : (\mathrm{pr}_1((a,b)),\mathrm{pr}_2((a,b))) = (a,b)$$

is well-typed, since both sides are judgmentally equal.Hence, it suffices to define $\mathrm{uniq}_{A\times B}((a,b)) :\equiv \mathrm{refl}_{(a,b)}$.

Product types, induction principle (7/7)

As previously, let's **apply the principle once** (in the **universal** case).

Product types, induction principle (7/7)

As previously, let's **apply the principle once** (in the **universal** case). We call the resulting function **induction for product types**:

Induction

Given $A, B : \mathcal{U}$, we have

$\mathsf{ind}_{A\times B} : \prod_{(C:\, A\times B \to \mathcal{U})} \left(\prod_{(x:\, A)} \prod_{(y:\, B)} C((x,y))\right) \to \prod_{(z:\, A\times B)} C(z)$, with the defining equation

$$\mathsf{ind}_{A\times B}(C, g, (a,b)) :\equiv g(a)(b)$$

Product types, induction principle (7/7)

As previously, let's **apply the principle once** (in the **universal** case). We call the resulting function **induction for product types**:

Induction

Given $A, B : \mathcal{U}$, we have
$\mathsf{ind}_{A\times B} : \prod_{(C : A\times B \to \mathcal{U})} \left(\prod_{(x : A)} \prod_{(y : B)} C((x, y))\right) \to \prod_{(z : A\times B)} C(z)$, with the defining equation

$$\mathsf{ind}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction, for the unit type

$\mathsf{ind}_{\mathbf{1}} : \prod_{(C : \mathbf{1}\to\mathcal{U})} C(*) \to \prod_{(x : \mathbf{1})} C(x)$, with defining equation $\mathsf{ind}_{\mathbf{1}}(C, c, *) :\equiv c$

Product types, induction principle (7/7)

As previously, let's **apply the principle once** (in the **universal** case). We call the resulting function **induction for product types**:

Induction

Given $A, B : \mathcal{U}$, we have

$\mathrm{ind}_{A\times B} : \prod_{(C:\, A\times B\,\to\,\mathcal{U}\,)} \left(\prod_{(x:\, A)} \prod_{(y:\, B)} C((x,y))\right) \to \prod_{(z:\, A\times B)} C(z)$, with the defining equation

$$\mathrm{ind}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction, for the unit type

$\mathrm{ind}_{\mathbf{1}} : \prod_{(C:\, \mathbf{1}\to\mathcal{U})} C(*) \to \prod_{(x:\, \mathbf{1})} C(x)$, with defining equation $\mathrm{ind}_{\mathbf{1}}(C, c, *) :\equiv c$

Product types, induction principle (7/7)

As previously, let's **apply the principle once** (in the **universal** case). We call the resulting function **induction for product types**:

Induction

Given $A, B : \mathcal{U}$, we have
$\mathrm{ind}_{A\times B} : \prod_{(C : A\times B \to \mathcal{U})} (\prod_{(x : A)} \prod_{(y : B)} C((x,y))) \to \prod_{(z : A\times B)} C(z)$, with the defining equation

$$\mathrm{ind}_{A\times B}(C, g, (a,b)) :\equiv g(a)(b)$$

Induction, for the unit type

$\mathrm{ind}_{\mathbf{1}} : \prod_{(C : \mathbf{1}\to\mathcal{U})} C(*) \to \prod_{(x : \mathbf{1})} C(x)$, with defining equation $\mathrm{ind}_{\mathbf{1}}(C, c, *) :\equiv c$

The propositional uniqueness principle for $\mathbf{1}$, $\mathrm{uniq}_{\mathbf{1}} : \prod_{(x : \mathbf{1})} x = \star$, with the defining equation $\mathrm{uniq}_1(\star) :\equiv \mathrm{refl}_\star$,
or via induction $\mathrm{uniq}_1 :\equiv \mathrm{ind}_1(\lambda x. x = \star, \mathrm{refl}_\star)$

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**,

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**.

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**. This is called a dependent pair type (or $\sum$-type).

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**. This is called a dependent pair type (or $\sum$-type). (Corresponds to an indexed sum over a given index, in Set Theory).

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**. This is called a dependent pair type (or $\sum$-type). (Corresponds to an indexed sum over a given index, in Set Theory).

Formation

Given type $A\colon \mathcal{U}$, a family $B\colon A \to \mathcal{U}$, their dependent pair type is $\sum_{(x\colon A)} B(x)$. (If B is constant, then $\sum_{(x\colon A)} B \equiv A \times B$)

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**. This is called a dependent pair type (or $\sum$-type). (Corresponds to an indexed sum over a given index, in Set Theory).

Formation

Given type $A \colon \mathcal{U}$, a family $B \colon A \to \mathcal{U}$, their dependent pair type is $\sum_{(x \colon A)} B(x)$. (If B is constant, then $\sum_{(x \colon A)} B \equiv A \times B$)

Dependent pair types ($\sum$-types), (1/4)

A **generalisation** of **product types**, that **allows the type** of the **second component** of a pair to **vary depending** on the choice of the **first component**. This is called a dependent pair type (or $\sum$-type). (Corresponds to an indexed sum over a given index, in Set Theory).

Formation

Given type $A : \mathcal{U}$, a family $B : A \to \mathcal{U}$, their dependent pair type is $\sum_{(x:\,A)} B(x)$. (If B is constant, then $\sum_{(x:\,A)} B \equiv A \times B$)
The way to construct an element of a dependent pair type, is by pairing.

Introduction

Given $a : A$ and $b : B(a)$, we may construct $(a, b) : \sum_{(x:\,A)} B(x)$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:\,A)} B(x) \to C$,

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x : A)} B(x) \to C$,

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x : A)} B(x) \to C$, we provide a function $g : \prod_{(x : A)} B(x) \to C$,

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:\,A)} B(x) \to C$, we provide a function $g : \prod_{(x:\,A)} B(x) \to C$, and then we can define f via

$$f((a,b)) :\equiv g(a)(b)$$

Example, projections

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:A)} B(x) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via

$$f((a,b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:A} B(x)) \to A$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:A)} B(x) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via

$$f((a,b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:A} B(x)) \to A$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:\, A)} B(x) \to C$, we provide a function $g : \prod_{(x:\, A)} B(x) \to C$, and then we can define f via

$$f((a,b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:\, A} B(x)) \to A$
  - $\mathrm{pr}_1((a,b)) :\equiv a$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:A)} B(x) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via

$$f((a,b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:A} B(x)) \to A$
  - $\mathrm{pr}_1((a,b)) :\equiv a$
- $\mathrm{pr}_2 : \prod_{p : \sum_{x:A} B(x)} B(\mathrm{pr}_1(p))$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:\, A)} B(x) \to C$, we provide a function $g : \prod_{(x:\, A)} B(x) \to C$, and then we can define f via

$$f((a, b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:\, A} B(x)) \to A$
  - $\mathrm{pr}_1((a, b)) :\equiv a$
- $\mathrm{pr}_2 : \prod_{p:\, \sum_{x:\, A} B(x)} B(\mathrm{pr}_1(p))$

Dependent pair types $\sum$-types, recursion principle (2/4)

Recursion principle

In order to define a non-dependent function ouf of a $\sum$-type $f : \sum_{(x:\, A)} B(x) \to C$, we provide a function $g : \prod_{(x:\, A)} B(x) \to C$, and then we can define f via

$$f((a, b)) :\equiv g(a)(b)$$

Example, projections

- $\mathrm{pr}_1 : (\sum_{x:\, A} B(x)) \to A$
  - $\mathrm{pr}_1((a, b)) :\equiv a$
- $\mathrm{pr}_2 : \prod_{p:\, \sum_{x:\, A} B(x)} B(\mathrm{pr}_1(p))$
  - ? (we need the induction principle)

Dependent pair types $\sum$-types, induction principle (3/4)

Induction principle

In order to construct a dependent function out of a $\sum$-type into a family $C : (\sum_{x:A} B(x)) \to \mathcal{U}$, we need a function

$$g : \prod_{x:A} \prod_{b:B(a)} C((a,b))$$

in order to derive a function

$$f : \prod_{p : \sum_{x:A} B(x)} C(p)$$

with defining equation

$$f((a,b)) :\equiv g(a)(b)$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:A} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C \right) \to \left(\sum_{x:A} B(x) \right) \to C$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:A} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C \right) \to \left(\sum_{x:A} B(x) \right) \to C$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:A} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C \right) \to \left(\sum_{x:A} B(x) \right) \to C$$

with defining equation

$$\mathrm{rec}_{\sum_{x:A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:\,A} B(x)} : \prod_{C:\,\mathcal{U}} \left(\prod_{x:\,A} B(x) \to C \right) \to \left(\sum_{x:\,A} B(x) \right) \to C$$

with defining equation

$$\mathrm{rec}_{\sum_{x:\,A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction operator

$$\mathrm{ind}_{\sum_{x:\,A} B(x)} : \prod_{C:\,(\sum_{x:\,A} B(x)) \to \mathcal{U}} \left(\prod_{a:\,A} \prod_{b:\,B(a)} C((a, b)) \right) \to \prod_{p:\,\sum_{x:\,A} B(x)} C(p)$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:A} B(x)} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C \right) \to \left(\sum_{x:A} B(x) \right) \to C$$

with defining equation

$$\mathrm{rec}_{\sum_{x:A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction operator

$$\mathrm{ind}_{\sum_{x:A} B(x)} : \prod_{C:(\sum_{x:A} B(x)) \to \mathcal{U}} \left(\prod_{a:A} \prod_{b:B(a)} C((a,b)) \right) \to \prod_{p:\sum_{x:A} B(x)} C(p)$$

Dependent pair types $\sum$-types, packaging (4/4)

Recursor

$$\mathrm{rec}_{\sum_{x:\,A} B(x)} : \prod_{C:\,\mathcal{U}} \left(\prod_{x:\,A} B(x) \to C \right) \to \left(\sum_{x:\,A} B(x) \right) \to C$$

with defining equation

$$\mathrm{rec}_{\sum_{x:\,A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Induction operator

$$\mathrm{ind}_{\sum_{x:\,A} B(x)} : \prod_{C:\,(\sum_{x:\,A} B(x)) \to \mathcal{U}} \left(\prod_{a:\,A} \prod_{b:\,B(a)} C((a, b)) \right) \to \prod_{p:\,\sum_{x:\,A} B(x)} C(p)$$

with the defining equation

$$\mathrm{ind}_{\sum_{x:\,A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$.

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$.

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

- $\mathsf{inl}(a) : A + B$ for $a : A$

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

- $\mathsf{inl}(a) : A + B$ for $a : A$
- $\mathsf{inr}(b) : A + B$ for $b : B$

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

- $\mathsf{inl}(a) : A + B$ for $a : A$
- $\mathsf{inr}(b) : A + B$ for $b : B$
- (No ways to construct elements of the empty type)

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

- $\mathrm{inl}(a) : A + B$ for $a : A$
- $\mathrm{inr}(b) : A + B$ for $b : B$
- (No ways to construct elements of the empty type)

Coproduct types, (1/3)

Formation rule

Given $A, B : \mathcal{U}$, we introduce their coproduct type $A + B : \mathcal{U}$. (A nullary version: the empty type $\mathbf{0} : \mathcal{U}$.)

Introduction rule

Two ways of constructing elements of $A + B$.

- $\mathsf{inl}(a) : A + B$ for $a : A$
- $\mathsf{inr}(b) : A + B$ for $b : B$
- (No ways to construct elements of the empty type)

Functions $f : A + B \to C$

Given $g_0 : A \to C$, $g_1 : B \to C$, we have the defining equations

$$f(\mathsf{inl}(a)) :\equiv g_0(a), f(\mathsf{inr}(b)) :\equiv g_1(b)$$

Coproduct types, (2/3)

Recursor

We have $\mathrm{rec}_{A+B} : \prod_{(C:\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$
with defining equations

Coproduct types, (2/3)

Recursor

We have $\mathsf{rec}_{A+B} : \prod_{(C\,:\,\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$ with defining equations

- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a)$

Coproduct types, (2/3)

Recursor

We have $\mathrm{rec}_{A+B} : \prod_{(C\,:\,\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$
with defining equations

- $\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inl}(a)) :\equiv g_0(a)$
- $\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inr}(b)) :\equiv g_1(b)$

Coproduct types, (2/3)

Recursor

We have $\mathsf{rec}_{A+B} : \prod_{(C:\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$ with defining equations

- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a)$
- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) :\equiv g_1(b)$

Coproduct types, (2/3)

Recursor

We have $\mathsf{rec}_{A+B} : \prod_{(C\,:\,\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$ with defining equations

- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a)$
- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) :\equiv g_1(b)$

We can always construct a function $f : \mathbf{0} \to C$ (without any defining equation),

Coproduct types, (2/3)

Recursor

We have $\mathsf{rec}_{A+B} : \prod_{(C:\mathcal{U})}(A \to C) \to (B \to C) \to A + B \to C$ with defining equations

- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a)$
- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) :\equiv g_1(b)$

We can always construct a function $f : \mathbf{0} \to C$ (without any defining equation),thus $\mathsf{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U})} \mathbf{0} \to C$.

Coproduct types, (2/3)

Recursor

We have $\mathsf{rec}_{A+B} : \prod_{(C:\mathcal{U})} (A \to C) \to (B \to C) \to A + B \to C$ with defining equations

- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a)$
- $\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) :\equiv g_1(b)$

We can always construct a function $f : \mathbf{0} \to C$ (without any defining equation),thus $\mathsf{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U})} \mathbf{0} \to C$. (This corresponds to the principle ex falso quodlibet, principle of explosion.)

Coproduct types, (3/3)

Dependent function $f : \prod_{z : A+B} C(z)$

Given family $C : A + B \to \mathcal{U}$,

Coproduct types, (3/3)

Dependent function $f : \prod_{z : A+B} C(z)$

Given family $C : A + B \to \mathcal{U}$,

Coproduct types, (3/3)

Dependent function $f\colon \prod_{z\colon A+B} C(z)$

Given family $C\colon A+B \to \mathcal{U}$, we require $g_0\colon \prod_{(a\colon A)} C(\mathrm{inl}(a))$, $g_1\colon \prod_{(b\colon B)} C(\mathrm{inr}(b))$,

Coproduct types, (3/3)

Dependent function $f\colon \prod_{z\colon A+B} C(z)$

Given family $C\colon A+B \to \mathcal{U}$, we require $g_0\colon \prod_{(a\colon A)} C(\mathsf{inl}(a))$, $g_1\colon \prod_{(b\colon B)} C(\mathsf{inr}(b))$, in order to produce a functtion f via the defining equations

$$f(\mathsf{inl}(a)) :\equiv g_0(a), f(\mathsf{inr}(b)) :\equiv g_1(b)$$

Coproduct types, (3/3)

Dependent function $f : \prod_{z : A+B} C(z)$

Given family $C : A + B \to \mathcal{U}$, we require $g_0 : \prod_{(a:\,A)} C(\mathrm{inl}(a))$, $g_1 : \prod_{(b:\,B)} C(\mathrm{inr}(b))$, in order to produce a functtion f via the defining equations

$$f(\mathrm{inl}(a)) :\equiv g_0(a), f(\mathrm{inr}(b)) :\equiv g_1(b)$$

In a nice package (induction principle):

$$\mathrm{ind}_{A+B} : \prod_{(C:\,(A+B)\to\mathcal{U})} \prod_{(a:\,A)} C(\mathrm{inl}(a)) \to \prod_{(b:\,B)} C(\mathrm{inr}(b)) \to \prod_{(x:\,A+B)} C(x)$$

For the empty type, $\mathrm{ind}_{\mathbf{0}} : \prod_{(C:\,\mathbf{0}\to\mathcal{U})} \prod_{(z:\,\mathbf{0})} C(z)$

The type of booleans, (1/3)

We introduce $\mathbf{2} : \mathcal{U}$, which is intended to have exactly two elements, $0_2, 1_2 : \mathbf{2}$. (Alternative definitions?)

The type of booleans, (1/3)

We introduce $\mathbf{2} : \mathcal{U}$, which is intended to have exactly two elements, $0_2, 1_2 : \mathbf{2}$. (Alternative definitions?)

Functions $f : \mathbf{2} \to C$

We require $c_0, c_1 : C$, to define a function f via the defining equations $f(0_2) :\equiv c0, f(1_2) :\equiv c1$

The type of booleans, (1/3)

We introduce $\mathbf{2} : \mathcal{U}$, which is intended to have exactly two elements, $0_2, 1_2 : \mathbf{2}$. (Alternative definitions?)

Functions $f : \mathbf{2} \to C$

We require $c_0, c_1 : C$, to define a function f via the defining equations $f(0_2) :\equiv c0, f(1_2) :\equiv c1$

The type of booleans, (1/3)

We introduce $\mathbf{2} : \mathcal{U}$, which is intended to have exactly two elements, $0_2, 1_2 : \mathbf{2}$. (Alternative definitions?)

Functions $f : \mathbf{2} \to C$

We require $c_0, c_1 : C$, to define a function f via the defining equations $f(0_2) :\equiv c0, f(1_2) :\equiv c1$

Recursion principle

Is a term $\mathrm{rec}_{\mathbf{2}} : \prod_{(C:\mathcal{U})} C \to C \to \mathbf{2} \to C$, with defining equations,

$$\mathrm{rec}_2(C, c_0, c_1, 0_2) :\equiv c_0, \mathrm{rec}_2(C, c_0, c_1, 1_2) :\equiv c_1$$

The type of booleans, (2/3)

Dependent functions $f : \prod_{(x:\mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$,

The type of booleans, (2/3)

Dependent functions $f : \prod_{(x:\mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$,

The type of booleans, (2/3)

Dependent functions $f : \prod_{(x : \mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$, we require elements $c_0 : C(0_2), c_1 : C(1_2)$,

The type of booleans, (2/3)

Dependent functions $f\colon \prod_{(x\colon \mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$, we require elements $c_0\colon C(0_2), c_1\colon C(1_2)$, in order to derive a dep function $f\colon \prod_{(x\colon \mathbf{2})} C(x)$,

The type of booleans, (2/3)

Dependent functions $f : \prod_{(x : \mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$, we require elements $c_0 : C(0_2), c_1 : C(1_2)$, in order to derive a dep function $f : \prod_{(x : \mathbf{2})} C(x)$, via defining equations $f(0_2) :\equiv c_0, f(1_2) :\equiv c_1$

The type of booleans, (2/3)

Dependent functions $f\colon \prod_{(x:\,\mathbf{2})} C(x)$

Given family $C : \mathbf{2} \to \mathcal{U}$, we require elements $c_0\colon C(0_2), c_1\colon C(1_2)$, in order to derive a dep function $f\colon \prod_{(x:\,\mathbf{2})} C(x)$, via defining equations $f(0_2) :\equiv c_0, f(1_2) :\equiv c_1$

In a nice packaging (induction principle)

We have $\mathrm{ind}_{\mathbf{2}}\colon \prod_{(C:\,\mathbf{2}\to\mathcal{U})} C(0_2) \to C(1_2) \to \prod_{(x:\,\mathbf{2})} C(x)$, via the defining equations

$$\mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 0_2) :\equiv c_0$$
$$\mathrm{ind}_{\mathbf{2}}(C, c_0, c_1, 1_2) :\equiv c_1$$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

The type of booleans, (3/3)

Is it true that $\prod_{(x:\, \mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?
  - $\mathsf{inl}(\mathsf{refl}_{0_2}) : C(0_2)$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\, \mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?
  - $\mathsf{inl}(\mathsf{refl}_{0_2}) : C(0_2)$
  - $\mathsf{inr}(\mathsf{refl}_{1_2}) : C(1_2)$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?
  - $\mathsf{inl}(\mathsf{refl}_{0_2}) : C(0_2)$
  - $\mathsf{inr}(\mathsf{refl}_{1_2}) : C(1_2)$

The type of booleans, (3/3)

Is it true that $\prod_{(x:\,\mathbf{2})}(x = 0_2) + (x = 1_2)$?

- Let's define the family $C(x) :\equiv x = 0_2 + x = 1_2$
  - $C(0_2) \equiv 0_2 = 0_2 + 0_2 = 1_2$
  - $C(1_2) \equiv 1_2 = 0_2 + 1_2 = 1_2$
- Can we find elements for each case?
  - $\mathrm{inl}(\mathrm{refl}_{0_2}) : C(0_2)$
  - $\mathrm{inr}(\mathrm{refl}_{1_2}) : C(1_2)$

Lastly, we derive,

$$\mathrm{ind}_{\mathbf{2}}(\lambda x.(x = 0_2 + x = 1_2), \mathrm{inl}(\mathrm{refl}_{0_2}), \mathrm{inr}(\mathrm{refl}_{1_2})) : \prod_{(x:\,\mathbf{2})} x = 0_1 + x = 1_2$$

The natural numbers, (1/7)

Introduction rules

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$

The natural numbers, (1/7)

Introduction rules

- zero : $\mathbb{N}$
- succ : $\mathbb{N} \to \mathbb{N}$

The natural numbers, (1/7)

Introduction rules

- zero : $\mathbb{N}$
- succ : $\mathbb{N} \to \mathbb{N}$

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \mathsf{zero}, 1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), \ldots$

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \mathsf{zero}, 1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), \ldots$

Recursion principle

In order to construct $f : \mathbb{N} \to C$, we need

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \mathsf{zero}, 1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), \ldots$

Recursion principle

In order to construct $f : \mathbb{N} \to C$, we need

- a starting point $c_0 : C$,

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \mathsf{zero}, 1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), \ldots$

Recursion principle

In order to construct $f : \mathbb{N} \to C$, we need

- a starting point $c_0 : C$,

The natural numbers, (1/7)

Introduction rules

- $\mathsf{zero} : \mathbb{N}$
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \mathsf{zero}, 1 :\equiv \mathsf{succ}(0), 2 :\equiv \mathsf{succ}(1), \ldots$

Recursion principle

In order to construct $f : \mathbb{N} \to C$, we need

- a starting point $c_0 : C$, a next step func $c_s : \mathbb{N} \to C \to C$

The natural numbers, (1/7)

Introduction rules

- zero: $\mathbb{N}$
- succ: $\mathbb{N} \to \mathbb{N}$

Usual notation: $0 :\equiv \text{zero}, 1 :\equiv \text{succ}(0), 2 :\equiv \text{succ}(1), \ldots$

Recursion principle

In order to construct $f : \mathbb{N} \to C$, we need

- a starting point $c_0 : C$, a next step func $c_s : \mathbb{N} \to C \to C$

These give rise to f, with the defining equations

$$f(0) :\equiv c_0, \qquad f(\text{succ}(n)) :\equiv c_s(n, f(n))$$

The natural numbers, (2/7)

Example

Define double: $\mathbb{N} \to \mathbb{N}$ which doubles its input.

The natural numbers, (2/7)

Example

Define double: $\mathbb{N} \to \mathbb{N}$ which doubles its input.

- $c_0 :\equiv 0,$ $c_s(n, y) :\equiv \mathrm{succ}(\mathrm{succ}(y))$

The natural numbers, (2/7)

Example

Define double: $\mathbb{N} \to \mathbb{N}$ which doubles its input.

- $c_0 :\equiv 0, \qquad c_s(n, y) :\equiv \mathsf{succ}(\mathsf{succ}(y))$
- $\mathsf{double}(0) :\equiv 0, \qquad \mathsf{double}(\mathsf{succ}(n)) :\equiv \mathsf{succ}(\mathsf{succ}(\mathsf{double}(n)))$

The natural numbers, (2/7)

Example

Define double: $\mathbb{N} \to \mathbb{N}$ which doubles its input.

- $c_0 :\equiv 0,$ $\quad c_s(n, y) :\equiv \mathrm{succ}(\mathrm{succ}(y))$
- $\mathrm{double}(0) :\equiv 0,$ $\quad \mathrm{double}(\mathrm{succ}(n)) :\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(n)))$

Calculation

$$\begin{aligned}
\mathrm{double}(2) &\equiv \mathrm{double}(\mathrm{succ}(\mathrm{succ}(0))) \equiv c_s(\mathrm{succ}(0), \mathrm{double}(\mathrm{succ}(0))) \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{double}(\mathrm{succ}(0)))) \equiv \mathrm{succ}(\mathrm{succ}(c_s(0, \mathrm{double}(0)))) \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{double}(0)))))[\equiv \mathrm{succ}^4(c_0)] \\
&\equiv \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0)))) \equiv 4
\end{aligned}$$

The natural numbers, multivariable functions (3/7)

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define $\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define add : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define $\text{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following “starting point” and “next step” data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$
- $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), c_s(m, g)(n) :\equiv \mathsf{succ}(g(n))$

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define add : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$
- $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), c_s(m, g)(n) :\equiv \mathrm{succ}(g(n))$

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define $\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$
- $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), c_s(m, g)(n) :\equiv \mathrm{succ}(g(n))$

That is, we have the following defining equations:

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define $\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$
- $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), c_s(m, g)(n) :\equiv \mathrm{succ}(g(n))$

That is, we have the following defining equations:

$$\mathrm{add}(0, n) :\equiv n$$
$$\mathrm{add}(\mathrm{succ}(m), n) :\equiv \mathrm{succ}(\mathrm{add}(m, n))$$

The natural numbers, multivariable functions (3/7)

(Just allow C to be a function type.)

Example

Define add : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, with the following "starting point" and "next step" data:

- $c_0 : \mathbb{N} \to \mathbb{N}, c_0(n) :\equiv n$
- $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}), c_s(m, g)(n) :\equiv \mathrm{succ}(g(n))$

That is, we have the following defining equations:

$$\mathrm{add}(0, n) :\equiv n$$
$$\mathrm{add}(\mathrm{succ}(m), n) :\equiv \mathrm{succ}(\mathrm{add}(m, n))$$

Calculation

$$\mathrm{add}(1, 2) \equiv \mathrm{add}(\mathrm{succ}(0), 2) \equiv \mathrm{succ}(\mathrm{add}(0, 2))$$
$$\equiv \mathrm{succ}(2) \equiv 3$$

The natural numbers, recursor (4/7)

Recursor

$\mathrm{rec}_{\mathbb{N}} : \prod_{C : \mathcal{U}} C \to (C \to \mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to C)$ with defining equations,

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n))$$

The natural numbers, recursor (4/7)

Recursor

$\mathrm{rec}_{\mathbb{N}} : \prod_{C : \mathcal{U}} C \to (C \to \mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to C)$ with defining equations,

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n))$$

The natural numbers, recursor (4/7)

Recursor

$\mathrm{rec}_{\mathbb{N}} : \prod_{C : \mathcal{U}} C \to (C \to \mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to C)$ with defining equations,

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n))$$

This way,

$$\mathrm{double} :\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, 0, \lambda n.\lambda y.\,\mathrm{succ}(\mathrm{succ}(y)))$$

$$\mathrm{add} :\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda n.\lambda g.\lambda m.\,\mathrm{succ}(g(m)))$$

The natural numbers, induction principle (5/7)

Induction principle

Assuming a family $f : \mathbb{N} \to \mathcal{U}$, an element $c_0 : C(0)$, and a function $c_s : \prod_{(n:\,\mathbb{N})} C(n) \to C(\mathrm{succ}(n))$, we can construct $f : \prod_{(n:\,\mathbb{N})} C(n)$ with the defining equations:

$$f(0) :\equiv c_0, \qquad f(\mathrm{succ}(n)) :\equiv c_s(n, f(n))$$

The natural numbers, induction principle (5/7)

Induction principle

Assuming a family $f : \mathbb{N} \to \mathcal{U}$, an element $c_0 : C(0)$, and a function $c_s : \prod_{(n : \mathbb{N})} C(n) \to C(\mathrm{succ}(n))$, we can construct $f : \prod_{(n : \mathbb{N})} C(n)$ with the defining equations:

$$f(0) :\equiv c_0, \qquad f(\mathrm{succ}(n)) :\equiv c_s(n, f(n))$$

- in nice packaging

We can construct $\mathrm{ind}_{\mathbb{N}} : \prod_{C : \mathbb{N} \to \mathcal{U}} C(0) \to (\prod_{n : \mathbb{N}} C(n) \to C(\mathrm{succ}(n))) \to \prod_{n : \mathbb{N}} C(n)$ with defining equations

$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$
$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}(n)) :\equiv c_s(n, \mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, n))$$

The natural numbers, induction principle, ex (6/7)

Example

Construct an element assoc: $\prod_{i,j,k\,:\,\mathbb{N}} i + (j + k) = (i + j) + k$

The natural numbers, induction principle, ex (6/7)

Example

Construct an element assoc: $\prod_{i,j,k\,:\,\mathbb{N}} i + (j + k) = (i + j) + k$

The natural numbers, induction principle, ex (6/7)

Example

Construct an element $\mathsf{assoc} : \prod_{i,j,k\,:\,\mathbb{N}} i + (j + k) = (i + j) + k$

By induction, it suffices to supply,

$$\mathsf{assoc}_0 : \prod_{j,k\,:\,\mathbb{N}} 0 + (j + k) = (0 + j) + k, \text{ and}$$

$$\mathsf{assoc}_s : \prod_{i\,:\,\mathbb{N}} \prod_{j,k\,:\,\mathbb{N}} i + (j + k) = (i + j) + k \to \prod_{j,k\,:\,\mathbb{N}} \mathsf{succ}(i) + (j + k) = (\mathsf{succ}(i) + j) + k$$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$
- Regarding assoc_s, notice that,

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathsf{succ}(i) + (j + k) \equiv \mathsf{succ}(i + (j + k))$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathrm{assoc}_0(j, k) :\equiv \mathrm{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathrm{succ}(i) + (j + k) \equiv \mathrm{succ}(i + (j + k))$
  - $(\mathrm{succ}(i) + j) + k \equiv \mathrm{succ}((i + j) + k)$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathsf{succ}(i) + (j + k) \equiv \mathsf{succ}(i + (j + k))$
  - $(\mathsf{succ}(i) + j) + k \equiv \mathsf{succ}((i + j) + k)$
  - The needed type for $\mathsf{assoc}_s(i, p, j, k)$ is equivalently $\mathsf{succ}(i + (j + k)) \equiv \mathsf{succ}((i + j) + k)$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathsf{succ}(i) + (j + k) \equiv \mathsf{succ}(i + (j + k))$
  - $(\mathsf{succ}(i) + j) + k \equiv \mathsf{succ}((i + j) + k)$
  - The needed type for $\mathsf{assoc}_s(i, p, j, k)$ is equivalently $\mathsf{succ}(i + (j + k)) \equiv \mathsf{succ}((i + j) + k)$
  - We are given $p(j, k) \colon i + (j + k) \equiv (i + j) + k$ (the "inductive hypothesis")

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathrm{assoc}_0(j, k) :\equiv \mathrm{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathrm{succ}(i) + (j + k) \equiv \mathrm{succ}(i + (j + k))$
  - $(\mathrm{succ}(i) + j) + k \equiv \mathrm{succ}((i + j) + k)$
  - The needed type for $\mathrm{assoc}_s(i, p, j, k)$ is equivalently $\mathrm{succ}(i + (j + k)) \equiv \mathrm{succ}((i + j) + k)$
  - We are given $p(j, k) : i + (j + k) \equiv (i + j) + k$ (the "inductive hypothesis")
- Invoke: if two naturals are equal, then their successors are. Provable in HoTT, we call this

$$\mathrm{ap}_{\mathrm{succ}} : (m =_{\mathbb{N}} n) \to (\mathrm{succ}(m) =_{\mathbb{N}} \mathrm{succ}(n))$$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathsf{assoc}_0(j, k) :\equiv \mathsf{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathsf{succ}(i) + (j + k) \equiv \mathsf{succ}(i + (j + k))$
  - $(\mathsf{succ}(i) + j) + k \equiv \mathsf{succ}((i + j) + k)$
  - The needed type for $\mathsf{assoc}_s(i, p, j, k)$ is equivalently $\mathsf{succ}(i + (j + k)) \equiv \mathsf{succ}((i + j) + k)$
  - We are given $p(j, k) : i + (j + k) \equiv (i + j) + k$ (the "inductive hypothesis")
- Invoke: if two naturals are equal, then their successors are. Provable in HoTT, we call this

$$\mathsf{ap}_{\mathsf{succ}} : (m =_{\mathbb{N}} n) \to (\mathsf{succ}(m) =_{\mathbb{N}} \mathsf{succ}(n))$$

The natural numbers, induction principle, ex cont (7/7)

- Calculate $0 + (j + k) \equiv j + k \equiv (0 + j) + k$, define $\mathrm{assoc}_0(j, k) :\equiv \mathrm{refl}_{j+k}$
- Regarding assoc_s, notice that,
  - $\mathrm{succ}(i) + (j + k) \equiv \mathrm{succ}(i + (j + k))$
  - $(\mathrm{succ}(i) + j) + k \equiv \mathrm{succ}((i + j) + k)$
  - The needed type for $\mathrm{assoc}_s(i, p, j, k)$ is equivalently $\mathrm{succ}(i + (j + k)) \equiv \mathrm{succ}((i + j) + k)$
  - We are given $p(j, k) : i + (j + k) \equiv (i + j) + k$ (the "inductive hypothesis")
- Invoke: if two naturals are equal, then their successors are. Provable in HoTT, we call this

$$\mathrm{ap}_{\mathrm{succ}} : (m =_{\mathbb{N}} n) \to (\mathrm{succ}(m) =_{\mathbb{N}} \mathrm{succ}(n))$$

Hence, $\mathrm{assoc}_s(i, p, j, k) :\equiv \mathrm{ap}_{\mathrm{succ}}(p(j, k))$

Table of Contents

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f \colon A + B \to C$ in two ways,

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f : A + B \to C$ in two ways,

1. via the recursor $f :\equiv \mathrm{rec}_{A+B}(C, g_0, g_1)$

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f : A + B \to C$ in two ways,

1. via the recursor $f :\equiv \mathsf{rec}_{A+B}(C, g_0, g_1)$
2. by the defining eqs $f(\mathsf{inl}(a)) :\equiv g_0(a), f(\mathsf{inr}(b)) :\equiv g_1(b)$

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f\colon A + B \to C$ in two ways,

1. via the recursor $f :\equiv \mathrm{rec}_{A+B}(C, g_0, g_1)$
2. by the defining eqs $f(\mathrm{inl}(a)) :\equiv g_0(a), f(\mathrm{inr}(b)) :\equiv g_1(b)$

Relation between the two?

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f : A + B \to C$ in two ways,

1. via the recursor $f :\equiv \mathrm{rec}_{A+B}(C, g_0, g_1)$
2. by the defining eqs $f(\mathrm{inl}(a)) :\equiv g_0(a), f(\mathrm{inr}(b)) :\equiv g_1(b)$

Relation between the two?

- $1 \Rightarrow 2$: use the computation rules of rec

Pattern matching and recursion, an observation (1/3)

Reminder

We are able to define a function $f\colon A+B \to C$ in two ways,

1. via the recursor $f :\equiv \mathsf{rec}_{A+B}(C, g_0, g_1)$
2. by the defining eqs $f(\mathsf{inl}(a)) :\equiv g_0(a), f(\mathsf{inr}(b)) :\equiv g_1(b)$

Relation between the two?

- $1 \Rightarrow 2$: use the computation rules of rec
- $2 \Rightarrow 1$: we're given $f(\mathsf{inl}(a)) :\equiv F_0, f(\mathsf{inr}(b)) :\equiv F_1$, thus

$$f :\equiv \mathsf{rec}_{A+B}(C, \lambda a.F_0, \lambda b.F_1)$$

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Solution

Read "double(n)" as the result of the recursive call. (Given double $:\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$, that's the second argument of c_s.)

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Solution

Read "double(n)" as the result of the recursive call. (Given double $:\equiv \mathsf{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$, that's the second argument of c_s.)

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Solution

Read "double(n)" as the result of the recursive call. (Given double $:\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$, that's the second argument of c_s.)

If we have an $f : \mathbb{N} \to C$ given as $f(0) :\equiv \Phi_0$, $f(\mathrm{succ}(n)) :\equiv \Phi_s$, where Φ_s may infolve n and the symbol "$f(n)$",

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Solution

Read "double(n)" as the result of the recursive call. (Given double $:\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$, that's the second argument of c_s.)

If we have an $f : \mathbb{N} \to C$ given as $f(0) :\equiv \Phi_0$, $f(\mathrm{succ}(n)) :\equiv \Phi_s$, where Φ_s may infolve n and the symbol "$f(n)$", we may translate it to $f :\equiv \mathrm{rec}_{\mathbb{N}}(C, F_0, \lambda n.\lambda r.\Phi'_s)$,

Pattern matching and recursion, problems? (2/3)

What if the defining eq involves the function itself in the definiens?

Solution

Read "double(n)" as the result of the recursive call. (Given double $:\equiv \mathrm{rec}_{\mathbb{N}}(\mathbb{N}, c_0, c_s)$, that's the second argument of c_s.)

If we have an $f : \mathbb{N} \to C$ given as $f(0) :\equiv \Phi_0$, $f(\mathrm{succ}(n)) :\equiv \Phi_s$, where Φ_s may infolve n and the symbol "$f(n)$", we may translate it to $f :\equiv \mathrm{rec}_{\mathbb{N}}(C, F_0, \lambda n.\lambda r.\Phi'_s)$, where Φ'_s is Φ_s with "$f(n)$" being replaced by r.

Pattern matching and recursion, problems? (3/3)

Definition by pattern matching

occurs when one conviniently constructs a function via defining equations (by recursion), or a dependent function (via induction).

Pattern matching and recursion, problems? (3/3)

Definition by pattern matching

occurs when one conviniently constructs a function via defining equations (by recursion), or a dependent function (via induction).

A restriction on the recursive calls

In order for a definition to be re-expressible using the recursive principle, the defined function can appear in the body of $f(\mathrm{succ}(n))$ as part of the symbol "$f(n)$".

Pattern matching and recursion, problems? (3/3)

Definition by pattern matching

occurs when one conviniently constructs a function via defining equations (by recursion), or a dependent function (via induction).

A restriction on the recursive calls

In order for a definition to be re-expressible using the recursive principle, the defined function can appear in the body of $f(\mathrm{succ}(n))$ as part of the symbol "$f(n)$".

Bad example

Defying the aforementioned can lead to
$f(0) :\equiv 0, f(n) :\equiv f(\mathrm{succ}(\mathrm{succ}(n)))$, which doesn't compute for all $n : \mathbb{N}$.

Propositions as types, (1/4)

refers to the following translation of logical connectives, into type-forming operations:

English	Type Theory
True	$\mathbf{1}$
False	$\mathbf{0}$
A and B	$A \times B$
If A then B	$A \to B$
A iff B	$(A \to B) \times (B \to A)$
not A	$A \to \mathbf{0}$
For all $x \colon A$, $P(x)$ holds	$\prod_{(x \colon A)} P(x)$
There exists $x \colon A$, such that $P(x)$	$\sum_{(x \colon A)} P(x)$

Propositions as types, some comments (2/4)

- Type $\mathbf{0}$ corresponds to falsity: an inhabitant of $\mathbf{0}$ is a contradiction and there is no basic way to prove a contradiction.

Propositions as types, some comments (2/4)

- Type $\mathbf{0}$ corresponds to falsity: an inhabitant of $\mathbf{0}$ is a contradiction and there is no basic way to prove a contradiction.
- We define the negation of A as $A \to \mathbf{0}$. A witness of $\neg A$ is a function $A \to \mathbf{0}$, which we may construct assuming $x : A$ and deriving an element of $\mathbf{0}$.

Propositions as types, some comments (2/4)

- Type $\mathbf{0}$ corresponds to falsity: an inhabitant of $\mathbf{0}$ is a contradiction and there is no basic way to prove a contradiction.
- We define the negation of A as $A \to \mathbf{0}$. A witness of $\neg A$ is a function $A \to \mathbf{0}$, which we may construct assuming $x : A$ and deriving an element of $\mathbf{0}$.
- This "proof by contradiction" is constractively valid. The invalid "PBC" is assuming $\neg A$, to derive A. Constructively, such an argument would only allow to conclude $\neg\neg A$, and there is no obvious way to get $\neg\neg A \to A$.

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"

(one of) De Morgan's Law(s)

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"
(one of) De Morgan's Law(s)

The proposition, in Type Theory

The corresponding type is $(A \to 0) \times (B \to 0) \to (A + B) \to 0$

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"

(one of) De Morgan's Law(s)

The proposition, in Type Theory

The corresponding type is $(A \to 0) \times (B \to 0) \to (A + B) \to 0$

Its proof, in Type Theory

(A recursion principle out of $(A \to \mathbf{0}) \times (B \to \mathbf{0})$, such that)

$$f((x, y)) :\equiv \square : A + B \to \mathbf{0},$$

for $(x, y) : (A \to \mathbf{0}) \times (B \to \mathbf{0})$)

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"
(one of) De Morgan's Law(s)

The proposition, in Type Theory

The corresponding type is $(A \to 0) \times (B \to 0) \to (A + B) \to 0$

Its proof, in Type Theory

(A recursion principle out of $(A \to \mathbf{0}) \times (B \to \mathbf{0})$, such that)

$$f((x, y)) :\equiv \square : A + B \to \mathbf{0},$$

for $(x, y) : (A \to \mathbf{0}) \times (B \to \mathbf{0})$)

- Letting $z : A + B$, we need $f((x, y))(z) :\equiv \square : \mathbf{0}$

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"

(one of) De Morgan's Law(s)

The proposition, in Type Theory

The corresponding type is $(A \to 0) \times (B \to 0) \to (A + B) \to 0$

Its proof, in Type Theory

(A recursion principle out of $(A \to \mathbf{0}) \times (B \to \mathbf{0})$, such that)

$$f((x, y)) :\equiv \square : A + B \to \mathbf{0},$$

for $(x, y) : (A \to \mathbf{0}) \times (B \to \mathbf{0})$)

- Letting $z : A + B$, we need $f((x, y))(z) :\equiv \square : \mathbf{0}$
- There are two cases,

$$f((x, y))(\mathrm{inl}(a)) :\equiv \square : \mathbf{0} \text{ and } f((x, y))(\mathrm{inr}(b)) :\equiv \square : \mathbf{0}$$

Propositions as types, an example (3/4)

"If not A and not B, then not (A or B)"
(one of) De Morgan's Law(s)

The proposition, in Type Theory

The corresponding type is $(A \to 0) \times (B \to 0) \to (A + B) \to 0$

Its proof, in Type Theory

(A recursion principle out of $(A \to \mathbf{0}) \times (B \to \mathbf{0})$, such that)

$$f((x, y)) :\equiv \square : A + B \to \mathbf{0},$$

for $(x, y) : (A \to \mathbf{0}) \times (B \to \mathbf{0})$)

- Letting $z : A + B$, we need $f((x, y))(z) :\equiv \square : \mathbf{0}$
- There are two cases,
  $$f((x, y))(\mathrm{inl}(a)) :\equiv \square : \mathbf{0} \text{ and } f((x, y))(\mathrm{inr}(b)) :\equiv \square : \mathbf{0}$$
- Hence,
  $$f((x, y))(\mathrm{inl}(a)) :\equiv x(a) : \mathbf{0} \text{ and } f((x, y))(\mathrm{inr}(b)) :\equiv y(b) : \mathbf{0}$$

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

Propositions as types, another example (4/4)

> *"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"*

What's the type?

$$(\prod_{(x\colon A)} P(x) \times Q(x)) \to (\prod_{(x\colon A)} P(x)) \times (\prod_{(x\colon A)} Q(x))$$

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

What's the type?

$$(\prod_{(x:\,A)} P(x) \times Q(x)) \to (\prod_{(x:\,A)} P(x)) \times (\prod_{(x:\,A)} Q(x))$$

- Supposing $p\colon \prod_{(x:\,A)} P(x) \times Q(x)$, we're looking for

Λογική και Διακριτά Μαθηματικά 2016 Μεταπτυχιακό Πρόγραμμα «Αλγόριθμοι, ∝∧μ∀

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

What's the type?

$$\left(\prod_{(x\colon A)} P(x) \times Q(x)\right) \to \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$$

- Supposing $p\colon \prod_{(x\colon A)} P(x) \times Q(x)$, we're looking for
- $f(p) :\equiv \square\colon \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

What's the type?

$$\left(\prod_{(x\colon A)} P(x) \times Q(x)\right) \to \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$$

- Supposing $p\colon \prod_{(x\colon A)} P(x) \times Q(x)$, we're looking for
- $f(p) :\equiv \square \colon \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv \left(\square \colon \prod_{(x\colon A)} P(x), \square \colon \prod_{(x\colon A)} Q(x)\right)$

Propositions as types, another example (4/4)

> *"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"*

What's the type?

$$\left(\prod_{(x\colon A)} P(x)\times Q(x)\right)\to\left(\prod_{(x\colon A)} P(x)\right)\times\left(\prod_{(x\colon A)} Q(x)\right)$$

- Supposing $p\colon \prod_{(x\colon A)} P(x)\times Q(x)$, we're looking for
- $f(p) :\equiv \Box\colon \left(\prod_{(x\colon A)} P(x)\right)\times\left(\prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv \left(\Box\colon \prod_{(x\colon A)} P(x), \Box\colon \prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv (\lambda x.(\Box\colon P(x)), \Box\colon \prod_{(x\colon A)} Q(x))$

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

What's the type?

$$\left(\prod_{(x\colon A)} P(x) \times Q(x)\right) \to \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$$

- Supposing $p\colon \prod_{(x\colon A)} P(x) \times Q(x)$, we're looking for
- $f(p) :\equiv \square\colon \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv \left(\square\colon \prod_{(x\colon A)} P(x), \square\colon \prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv (\lambda x.(\square\colon P(x)), \square\colon \prod_{(x\colon A)} Q(x))$

Propositions as types, another example (4/4)

"If for all $x : A$, $P(x)$ and $Q(x)$, then for all $x : A, P(x)$ and for all $x : A, Q(x)$"

What's the type?

$$\left(\prod_{(x:\,A)} P(x) \times Q(x)\right) \to \left(\prod_{(x:\,A)} P(x)\right) \times \left(\prod_{(x:\,A)} Q(x)\right)$$

- Supposing $p : \prod_{(x:\,A)} P(x) \times Q(x)$, we're looking for
- $f(p) :\equiv \square : \left(\prod_{(x:\,A)} P(x)\right) \times \left(\prod_{(x:\,A)} Q(x)\right)$
- $f(p) :\equiv \left(\square : \prod_{(x:\,A)} P(x), \square : \prod_{(x:\,A)} Q(x)\right)$
- $f(p) :\equiv (\lambda x.(\square : P(x)), \square : \prod_{(x:\,A)} Q(x))$
  - We have that $p(x) : P(x) \times Q(x)$, hence $\mathrm{pr}_1(p(x)) : P(x)$ and $f(p) :\equiv (\lambda x.\, \mathrm{pr}_1(p(x)), \square : \prod_{(x:\,A)} Q(x))$

Propositions as types, another example (4/4)

"If for all $x\colon A$, $P(x)$ and $Q(x)$, then for all $x\colon A, P(x)$ and for all $x\colon A, Q(x)$"

What's the type?

$$\left(\prod_{(x\colon A)} P(x) \times Q(x)\right) \to \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$$

- Supposing $p\colon \prod_{(x\colon A)} P(x) \times Q(x)$, we're looking for
- $f(p) :\equiv \square\colon \left(\prod_{(x\colon A)} P(x)\right) \times \left(\prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv \left(\square\colon \prod_{(x\colon A)} P(x), \square\colon \prod_{(x\colon A)} Q(x)\right)$
- $f(p) :\equiv (\lambda x.(\square\colon P(x)), \square\colon \prod_{(x\colon A)} Q(x))$
  - We have that $p(x)\colon P(x) \times Q(x)$, hence $\mathrm{pr}_1(p(x))\colon P(x)$ and $f(p) := (\lambda x.\, \mathrm{pr}_1(p(x)), \square\colon \prod_{(x\colon A)} Q(x))$
- Lastly,

$$f(p) :\equiv (\lambda x.\, \mathrm{pr}_1(p(x)), \lambda x.\, \mathrm{pr}_2(p(x)))$$

The "natural" propositions-as-types logic confines itself to effective and computationally meaningful constructions.

The “natural” propositions-as-types logic confines itself to effective and computationally meaningful constructions. LEM has no effective procedure for deciding whether a proposition is true or false. Pros:

The “natural” propositions-as-types logic confines itself to effective and computationally meaningful constructions. LEM has no effective procedure for deciding whether a proposition is true or false. Pros:

- there’s an intrinsic computational meaning

Thus, type theory enriches, rather than constrains, convential mathematical practice.

The “natural” propositions-as-types logic confines itself to effective and computationally meaningful constructions. LEM has no effective procedure for deciding whether a proposition is true or false. Pros:

- there’s an intrinsic computational meaning
- axiomatic freedom: there’s no construction witnessing LEM,

Thus, type theory enriches, rather than constrains, convential mathematical practice.

The "natural" propositions-as-types logic confines itself to effective and computationally meaningful constructions. LEM has no effective procedure for deciding whether a proposition is true or false. Pros:

- there's an intrinsic computational meaning
- axiomatic freedom: there's no construction witnessing LEM,
- the logic is compatible with the existence of on (type theory does not deny LEM)

Thus, type theory enriches, rather than constrains, convential mathematical practice.

$n \leq m :\equiv \sum_{k:\mathbb{N}} n + k = m$
$n < m :\equiv \sum_{(k:\mathbb{N})} n + succ(k) = m$ $n < m :\equiv (n \leq m) \times \neg(n = m)$
—— we can use the universes in typoe theory to represent "higher order logic" - ie quantify over all propositions or over all predicates for example, we can represent the prop "for all prop P: A -¿ U, if P(a) then P(b)" $Pi(P : A- > U).P(a) \to P(b)$
where A : U, a,b : A. Apriori, this lives in a different, higher, universe than the props we are quantifying over, ie
$(Pi(P : A \to U_i)... : U_i + 1$

—

Proof relevance, what is (1/2)

We described “proof-relevant” translation of propositions, where the proofs of disjuctions and existential statements carry some information:

Proof relevance, what is (1/2)

We described "proof-relevant" translation of propositions, where the proofs of disjuctions and existential statements carry some information:

- an inhabitant of $A + B$ (regarded as a witness of "A or B"), points to whether it came from A or from B

Proof relevance, what is (1/2)

We described “proof-relevant” translation of propositions, where the proofs of disjuctions and existential statements carry some information:

- an inhabitant of $A + B$ (regarded as a witness of “A or B”), points to whether it came from A or from B
- an inhabitant $\sum_{x\colon A} P(x)$, informs us at what x is; (the first projection of the inhabitant)

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

Proof relevance, consequences (2/2)

An observation

We can have "*A* iff *B*", with *A* and *B* exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

so far

"A iff B" tells us that A and B are **logically** equivalent - which differs from equivalence of types.

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

so far

"A iff B" tells us that A and B are **logically** equivalent - which differs from equivalence of types.

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

so far

"A iff B" tells us that A and B are **logically** equivalent - which differs from equivalence of types. $\mathbb{N}$ and $\mathbf{1}$ are logically equivalent but not equivalent types.

Proof relevance, consequences (2/2)

An observation

We can have "A iff B", with A and B exhibiting different behaviour: $\mathbb{N}$ iff $\mathbf{1}$ $\quad (\mathbb{N} \to \mathbf{1}) \times (\mathbf{1} \to \mathbb{N})$

Explanation

This equivalence tells us only that, when regarded as a mere propositions, $\mathbb{N}$ represents the same proposition as $\mathbf{1}$ (the true proposition)

so far

"A iff B" tells us that A and B are **logically** equivalent - which differs from equivalence of types. $\mathbb{N}$ and $\mathbf{1}$ are logically equivalent but not equivalent types.

Foreshadowing: there is class of types called "**mere propositions**", where logical and type equivalence **coincide**.

Identity types, (1/3)

Formation rule

Given type $A : \mathcal{U}$ and two elements $a, b : A$, we form the type

$$(a =_A b) : \mathcal{U}, \qquad [\text{Typically}, \mathrm{Id}_A(a, b)]$$

Identity types, (1/3)

Formation rule

Given type $A : \mathcal{U}$ and two elements $a, b : A$, we form the type

$$(a =_A b) : \mathcal{U}, \qquad [\text{Typically}, \mathsf{Id}_A(a, b)]$$

Introduction rule

The basic way to construct an element of $a =_A b$ is to know that a and b are the same.

Identity types, (1/3)

Formation rule

Given type $A : \mathcal{U}$ and two elements $a, b : A$, we form the type

$$(a =_A b) : \mathcal{U}, \qquad\qquad [\text{Typically}, \mathrm{Id}_A(a, b)]$$

Introduction rule

The basic way to construct an element of $a =_A b$ is to know that a and b are the same.

Identity types, (1/3)

Formation rule

Given type $A : \mathcal{U}$ and two elements $a, b : A$, we form the type

$$(a =_A b) : \mathcal{U}, \qquad [\text{Typically}, \mathsf{Id}_A(a, b)]$$

Introduction rule

The basic way to construct an element of $a =_A b$ is to know that a and b are the same. Thus,

$$\mathsf{refl} : \prod_{a : A} a =_A a$$

Identity types, (1/3)

Formation rule

Given type $A\colon \mathcal{U}$ and two elements $a, b\colon A$, we form the type

$$(a =_A b)\colon \mathcal{U}, \qquad\qquad [\text{Typically}, \mathsf{Id}_A(a,b)]$$

Introduction rule

The basic way to construct an element of $a =_A b$ is to know that a and b are the same. Thus,

$$\mathsf{refl}\colon \prod_{a\colon A} a =_A a$$

In particular, if $a \equiv b$, then we also have an element $\mathsf{refl}_a\colon a =_A b$

Identity types, equals may be substituted for equals, induction (2/3)

Indiscernibility of identicals (a consequence of ind princ)

For every family $C: A \to \mathcal{U}$ there is a function

$$f : \prod_{(x,y:\,A)} \prod_{(p:\,x=_A y)} C(x) \to C(y)$$

such that $f(x, x, \mathsf{refl}_x) :\equiv \mathsf{id}_{C(x)}$

Identity types, equals may be substituted for equals, induction (2/3)

Indiscernibility of identicals (a consequence of ind princ)

For every family $C\colon A \to \mathcal{U}$ there is a function

$$f\colon \prod_{(x,y\colon A)} \prod_{(p\colon x=_A y)} C(x) \to C(y)$$

such that $f(x,x,\mathsf{refl}_x) :\equiv \mathsf{id}_{C(x)}$

Path induction

Given a family $C\colon \prod_{x,y\colon A}(x =_A y) \to \mathcal{U}$ and a function $c\colon \prod_{x\colon A} C(x,x,\mathsf{refl}_x)$ there is a function

$$f\colon \prod_{(x,y\colon A)} \prod_{(p\colon x=_A y)} C(x,y,p)$$

such that $f(x,x,\mathsf{refl}_x) :\equiv c(x)$.

Identity types, disequality (3/3)

Definition

is the negation of equality: $(x \neq_A y) :\equiv \neg(x =_A y)$

Table of Contents

Homotopy Type Theory

Homotopy Type Theory (HoTT) extends MLTT by changing the interpretation of equality and incorporating ideas from homotopy theory and higher category theory:

- Identity types: HoTT views equality as a path in a space, leading to a richer structure.

Homotopy Type Theory

Homotopy Type Theory (HoTT) extends MLTT by changing the interpretation of equality and incorporating ideas from homotopy theory and higher category theory:

- Identity types: HoTT views equality as a path in a space, leading to a richer structure.
- Univalence Axiom (UA): Introduced by Vladimir Voevodsky, states that equivalent types are identifiable (i.e., they are equal in the type-theoretic sense). Formally, for a universe $\mathcal{U}$, there is an equivalence: $(A \simeq B) \simeq (A =_{\mathcal{U}} B)$

Homotopy Type Theory

Homotopy Type Theory (HoTT) extends MLTT by changing the interpretation of equality and incorporating ideas from homotopy theory and higher category theory:

- Identity types: HoTT views equality as a path in a space, leading to a richer structure.
- Univalence Axiom (UA): Introduced by Vladimir Voevodsky, states that equivalent types are identifiable (i.e., they are equal in the type-theoretic sense). Formally, for a universe $\mathcal{U}$, there is an equivalence: $(A \simeq B) \simeq (A =_{\mathcal{U}} B)$
- Higher inductive types (HITs): generalisation of inductive types, (allows the introduction of paths and higher paths)