Lecture 6: Equality

Łukasz Czajka

Definitional equality

Coq's <u>definitional equality</u> $\equiv$ includes:

Definitional equality

Coq's definitional equality $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered identical.

Definitional equality

Coq's <u>definitional equality</u> $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered <u>identical</u>.
- β-equality: $(\lambda x : \tau.t)s =_\beta t[s/x]$.

Definitional equality

Coq's <u>definitional equality</u> $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered <u>identical</u>.
- β-equality: $(\lambda x : \tau.t)s =_\beta t[s/x]$.
- η-equality: $\lambda x : \tau.tx =_\eta t$ if $x \notin \mathrm{FV}(t)$.

Definitional equality

Coq's definitional equality $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered identical.
- β-equality: $(\lambda x : \tau.t)s =_\beta t[s/x]$.
- η-equality: $\lambda x : \tau.tx =_\eta t$ if $x \notin \mathrm{FV}(t)$.
- ι-equality: generated by the reductions associated with fixpoints and matches.

Definitional equality

Coq's <u>definitional equality</u> $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered <u>identical</u>.
- β-equality: $(\lambda x : \tau.t)s =_\beta t[s/x]$.
- η-equality: $\lambda x : \tau.tx =_\eta t$ if $x \notin \mathrm{FV}(t)$.
- ι-equality: generated by the reductions associated with fixpoints and matches.
- δ-equality: a defined constant is definitionally equal to its definition (unfolding/folding a definition).

Definitional equality

Coq's definitional equality $\equiv$ includes:

- α-equality (implicitly): compatible renaming of bound variables, e.g., $\lambda x : \tau.t$ and $\lambda y : \tau.t[y/x]$ are considered identical.
- β-equality: $(\lambda x : \tau.t)s =_\beta t[s/x]$.
- η-equality: $\lambda x : \tau.tx =_\eta t$ if $x \notin \mathrm{FV}(t)$.
- ι-equality: generated by the reductions associated with fixpoints and matches.
- δ-equality: a defined constant is definitionally equal to its definition (unfolding/folding a definition).
- ζ-equality:

  $$(\texttt{let x := s in t}) =_\zeta \texttt{t[s/x]}$$

Conversion rule

Coq's conversion relation $\leq$ includes definitional equality and subtyping between universes.

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash \tau' : \mathcal{U} \quad \tau \leq \tau'}{\Gamma \vdash t : \tau'} \text{ (conv)}$$

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
  - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
    - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
    - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.
- Propositional equality $t =_\tau t'$ is a proposition in the logic which expresses that t is equal to t' in type τ.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
  - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.
- Propositional equality $t =_\tau t'$ is a proposition in the logic which expresses that t is equal to t' in type τ.
  - $t =_\tau t'$ is a type if $t, t' : \tau$.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
  - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.
- Propositional equality $t =_\tau t'$ is a proposition in the logic which expresses that t is equal to t' in type τ.
  - $t =_\tau t'$ is a type if $t, t' : \tau$. In particular, $t =_\tau t'$ may be assumed (may appear in the context).

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
  - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.
- Propositional equality $t =_\tau t'$ is a proposition in the logic which expresses that t is equal to t' in type τ.
  - $t =_\tau t'$ is a type if $t, t' : \tau$. In particular, $t =_\tau t'$ may be assumed (may appear in the context).
  - $=$ is defined in Coq's logic as an inductive predicate.

Propositional equality

- Definitional equality is a meta-level relation: it is not possible to state in the logic that two terms are definitionally equal.
  - $t \equiv t'$ is a (decidable) meta-level statement about the terms t, t' treated as syntactic objects.
  - It is not possible to write $t \equiv t'$ in the logic itself: $t \equiv t'$ is not a type.
- Propositional equality $t =_\tau t'$ is a proposition in the logic which expresses that t is equal to t' in type τ.
  - $t =_\tau t'$ is a type if $t, t' : \tau$. In particular, $t =_\tau t'$ may be assumed (may appear in the context).
  - $=$ is defined in Coq's logic as an inductive predicate.
  - if $t \equiv t'$ and $t, t' : \tau$ then $t =_\tau t'$ is inhabited (has an element/proof).

Propositional equality

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.

Arguments eq {A}.
Arguments eq_refl {A x}, {A}.
(* we can write just `eq_refl' or `eq_refl y' *)

Notation "x = y :> A" := (@eq A x y) (at level 70).
Notation "x = y" := (eq x y) (at level 70).
```

Propositional equality

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.

Arguments eq {A}.
Arguments eq_refl {A x}, {A}.
(* we can write just `eq_refl' or `eq_refl y' *)

Notation "x = y :> A" := (@eq A x y) (at level 70).
Notation "x = y" := (eq x y) (at level 70).
```

- In the inductive definition of `eq`, the type `A` is an implicit parameter, the left side `x` of the equality is a parameter, the right side is an index.

Propositional equality

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.

Arguments eq {A}.
Arguments eq_refl {A x}, {A}.
(* we can write just `eq_refl' or `eq_refl y' *)

Notation "x = y :> A" := (@eq A x y) (at level 70).
Notation "x = y" := (eq x y) (at level 70).
```

- In the inductive definition of `eq`, the type `A` is an implicit parameter, the left side `x` of the equality is a parameter, the right side is an index.
- The constructor `eq_refl` forces the index to be identical to the parameter (modulo definitional equality).

Propositional equality

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.

Arguments eq {A}.
Arguments eq_refl {A x}, {A}.
(* we can write just `eq_refl' or `eq_refl y' *)

Notation "x = y :> A" := (@eq A x y) (at level 70).
Notation "x = y" := (eq x y) (at level 70).
```

- In the inductive definition of `eq`, the type `A` is an implicit parameter, the left side `x` of the equality is a parameter, the right side is an index.
- The constructor `eq_refl` forces the index to be identical to the parameter (modulo definitional equality).
- The full type of the constructor `eq_refl` states the reflexivity of equality:

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

Propositional equality

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq A x x.

Arguments eq {A}.
Arguments eq_refl {A x}, {A}.
(* we can write just `eq_refl' or `eq_refl y' *)

Notation "x = y :> A" := (@eq A x y) (at level 70).
Notation "x = y" := (eq x y) (at level 70).
```

- In the inductive definition of `eq`, the type `A` is an implicit parameter, the left side `x` of the equality is a parameter, the right side is an index.
- The constructor `eq_refl` forces the index to be identical to the parameter (modulo definitional equality).
- The full type of the constructor `eq_refl` states the reflexivity of equality:

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```
- `eq` is a small propositional inductive type, so equality proofs may be eliminated to create programs.

Propositional equality elimination

```
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (t : P x)
    (y : A) (e : x = y) =>
match e in @eq _ _ y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
```

Propositional equality elimination

```
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (t : P x)
    (y : A) (e : x = y) =>
match e in @eq _ _ y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
```

- Inside the match branch, the index variable `y'` is replaced with the parameter `x`:

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

Propositional equality elimination

```
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (t : P x)
    (y : A) (e : x = y) =>
match e in @eq _ _ y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
```

- Inside the match branch, the index variable `y'` is replaced with the parameter `x`:

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

  So `t` is required to have type `P x` inside the branch, which agrees with its actual type.

Propositional equality elimination

```
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (t : P x)
    (y : A) (e : x = y) =>
match e in @eq _ _ y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
```

- Inside the match branch, the index variable `y'` is replaced with the parameter `x`:

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

  So `t` is required to have type `P x` inside the branch, which agrees with its actual type.
- The type of the entire match expression is `P y`.

Propositional equality elimination

```
eq_ind =
fun (A : Type) (x : A) (P : A -> Prop) (t : P x)
    (y : A) (e : x = y) =>
match e in @eq _ _ y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
```

- Inside the match branch, the index variable `y'` is replaced with the parameter `x`:

  `eq_refl : forall (A : Type) (x : A), x = x`

  So `t` is required to have type `P x` inside the branch, which agrees with its actual type.
- The type of the entire match expression is `P y`.
- `eq_ind` computes on `eq_refl`:

  `eq_ind A a P t a (@eq_refl A a)` $\to_\iota$ `t`

Propositional equality elimination

Elimination into `Type` or `Set` is allowed for `eq`, because it is a small propositional inductive type.

```
eq_rect =
fun (A : Type) (x : A) (P : A -> Type) (t : P x)
    (y : A) (e : x = y) =>
match e in _ = y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Type),
    P x -> forall y : A, x = y -> P y
```

Propositional equality elimination

Elimination into `Type` or `Set` is allowed for `eq`, because it is a small propositional inductive type.

```
eq_rect =
fun (A : Type) (x : A) (P : A -> Type) (t : P x)
    (y : A) (e : x = y) =>
match e in _ = y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Type),
    P x -> forall y : A, x = y -> P y
```

`eq_rect A a P t a (eq_refl a)` $\rightarrow_\iota$ `t`

Propositional equality elimination

Elimination into `Type` or `Set` is allowed for `eq`, because it is a small propositional inductive type.

```
eq_rect =
fun (A : Type) (x : A) (P : A -> Type) (t : P x)
    (y : A) (e : x = y) =>
match e in _ = y' return P y' with
| eq_refl => t
end
: forall (A : Type) (x : A) (P : A -> Type),
    P x -> forall y : A, x = y -> P y
```

`eq_rect A a P t a (eq_refl a)` $\to_\iota$ `t`

Used to implement type casts.

Symmetry of equality

```
eq_sym =
fun (A : Type) (x y : A) (H : x = y) =>
match H in _ = y' return y' = x with
| eq_refl => @eq_refl A x
end
: forall (A : Type) (x y : A), x = y -> y = x
```

Symmetry of equality

```
eq_sym =
fun (A : Type) (x y : A) (H : x = y) =>
match H in _ = y' return y' = x with
| eq_refl => @eq_refl A x
end
: forall (A : Type) (x y : A), x = y -> y = x
```

- Inside the match branch, the index variable `y'` is replaced with `x`, so `@eq_refl A x` is required to have type `x = x`.

Symmetry of equality

```
eq_sym =
fun (A : Type) (x y : A) (H : x = y) =>
match H in _ = y' return y' = x with
| eq_refl => @eq_refl A x
end
: forall (A : Type) (x y : A), x = y -> y = x
```

- Inside the match branch, the index variable `y'` is replaced with `x`, so `@eq_refl A x` is required to have type `x = x`.
- The entire match expression has type `y = x`.

Transitivity of equality

```
eq_trans =
fun (A : Type) (x y z : A) (H1 : x = y) (H2 : y = z) =>
match H2 in (_ = z') return (x = z') with
| eq_refl => H1
end
: forall (A : Type) (x y z : A), x = y -> y = z -> x = z
```

Transitivity of equality

```
eq_trans =
fun (A : Type) (x y z : A) (H1 : x = y) (H2 : y = z) =>
match H2 in (_ = z') return (x = z') with
| eq_refl => H1
end
: forall (A : Type) (x y z : A), x = y -> y = z -> x = z
```

- Inside the match branch, the index variable `z'` is replaced with `y`, so `H1` is required to have type `x = y`.

Transitivity of equality

```
eq_trans =
fun (A : Type) (x y z : A) (H1 : x = y) (H2 : y = z) =>
match H2 in (_ = z') return (x = z') with
| eq_refl => H1
end
: forall (A : Type) (x y z : A), x = y -> y = z -> x = z
```

- Inside the match branch, the index variable `z'` is replaced with `y`, so `H1` is required to have type `x = y`.
- The entire match expression has type `x = z`.

Compatibility of functions with equality

```
f_equal =
fun (A B : Type) (f : A -> B) (x y : A) (H : x = y) =>
match H in _ = y' return f x = f y' with
| eq_refl => @eq_refl B (f x)
end
: forall (A B : Type) (f : A -> B) (x y : A),
  x = y -> f x = f y
```

Compatibility of functions with equality

```
f_equal =
fun (A B : Type) (f : A -> B) (x y : A) (H : x = y) =>
match H in _ = y' return f x = f y' with
| eq_refl => @eq_refl B (f x)
end
: forall (A B : Type) (f : A -> B) (x y : A),
  x = y -> f x = f y
```

- Inside the match branch, the index variable `y'` is replaced with `x`, so `@eq_refl B (f x)` is required to have type `f x = f x`.

Compatibility of functions with equality

```
f_equal =
fun (A B : Type) (f : A -> B) (x y : A) (H : x = y) =>
match H in _ = y' return f x = f y' with
| eq_refl => @eq_refl B (f x)
end
: forall (A B : Type) (f : A -> B) (x y : A),
  x = y -> f x = f y
```

- Inside the match branch, the index variable `y'` is replaced with `x`, so `@eq_refl B (f x)` is required to have type `f x = f x`.
- The entire match expression has type `f x = f y`.

Equality tactics

- `reflexivity` is `apply eq_refl`.

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

Equality tactics

- `reflexivity` is `apply eq_refl`.

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```
- `symmetry` is `apply eq_sym`.

Equality tactics

- `reflexivity` is `apply eq_refl`.

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

- `symmetry` is `apply eq_sym`.

  ```
  eq_sym : forall (A : Type) (x y : A), x = y -> y = x
  ```

Equality tactics

- `reflexivity` is `apply eq_refl`.

```
eq_refl : forall (A : Type) (x : A), x = x
```

- `symmetry` is `apply eq_sym`.

```
eq_sym : forall (A : Type) (x y : A), x = y -> y = x
```

- `transitivity y` is `apply eq_trans with (y := y)`.

```
eq_trans : forall (A : Type) (x y z : A),
           x = y -> y = z -> x = z
```

Equality tactics

- `reflexivity` is `apply eq_refl`.

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

- `symmetry` is `apply eq_sym`.

  ```
  eq_sym : forall (A : Type) (x y : A), x = y -> y = x
  ```

- `transitivity y` is `apply eq_trans with (y := y).`

  ```
  eq_trans : forall (A : Type) (x y z : A),
             x = y -> y = z -> x = z
  ```

- `rewrite H` with `H : a = b` is `refine (eq_ind ..)` with appropriate arguments.

  ```
  eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
           P x -> forall y : A, x = y -> P y
  ```

Equality tactics

- `reflexivity` is `apply eq_refl`.

  ```
  eq_refl : forall (A : Type) (x : A), x = x
  ```

- `symmetry` is `apply eq_sym`.

  ```
  eq_sym : forall (A : Type) (x y : A), x = y -> y = x
  ```

- `transitivity y` is `apply eq_trans with (y := y)`.

  ```
  eq_trans : forall (A : Type) (x y z : A),
             x = y -> y = z -> x = z
  ```

- `rewrite H` with `H : a = b` is `refine (eq_ind ..)` with appropriate arguments.

  ```
  eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
           P x -> forall y : A, x = y -> P y
  ```

- E.g., if `H : a = b` and the goal is `P a` then `rewrite H` is `refine (eq_ind A b P _ a (eq_sym H))`.

Type casts

- `eq_rect` is used to implement type casts.

```
eq_rect : forall (A : Type) (x : A) (P : A -> Type),
    P x -> forall y : A, x = y -> P y
```

`eq_rect A a P t a (eq_refl a)` $\to_\iota$ `t`

Type casts

- `eq_rect` is used to implement type casts.

  ```
  eq_rect : forall (A : Type) (x : A) (P : A -> Type),
      P x -> forall y : A, x = y -> P y
  ```

  `eq_rect A a P t a (eq_refl a)` $\rightarrow_\iota$ `t`
- For `p : a = a`, shouldn't `eq_rect A a P t a p` be (propositionally) equal to `t`?

Type casts

- `eq_rect` is used to implement type casts.

  ```
  eq_rect : forall (A : Type) (x : A) (P : A -> Type),
      P x -> forall y : A, x = y -> P y
  ```

  `eq_rect A a P t a (eq_refl a)` $\to_\iota$ `t`
- For `p : a = a`, shouldn't `eq_rect A a P t a p` be (propositionally) equal to `t`?
- After all, `eq_refl` is the only constructor of `eq`, so we "should" have `p = eq_refl a`.

Type casts

- `eq_rect` is used to implement type casts.

  ```
  eq_rect : forall (A : Type) (x : A) (P : A -> Type),
      P x -> forall y : A, x = y -> P y
  ```

  `eq_rect A a P t a (eq_refl a)` $\to_\iota$ `t`
- For `p : a = a`, shouldn't `eq_rect A a P t a p` be (propositionally) equal to `t`?
- After all, `eq_refl` is the only constructor of `eq`, so we "should" have `p = eq_refl a`.
- This is indeed the case if `p` is closed (contains no free variables/axioms/opaque constants), because then `p` just computes to `eq_refl`.

Type casts

- `eq_rect` is used to implement type casts.

```
eq_rect : forall (A : Type) (x : A) (P : A -> Type),
    P x -> forall y : A, x = y -> P y
```

  `eq_rect A a P t a (eq_refl a)` $\to_\iota$ `t`
- For `p : a = a`, shouldn't `eq_rect A a P t a p` be (propositionally) equal to `t`?
- After all, `eq_refl` is the only constructor of `eq`, so we "should" have `p = eq_refl a`.
- This is indeed the case if `p` is closed (contains no free variables/axioms/opaque constants), because then `p` just computes to `eq_refl`.
- But in general it is not possible to prove `p = eq_refl a`!

Uniqueness of identity proofs

- The Uniqueness of Identity Proofs (UIP) axiom:

```
Axiom UIP
: forall (A : Type) (x y : A) (p1 p2 : x = y), p1 = p2
```

Uniqueness of identity proofs

- The Uniqueness of Identity Proofs (UIP) axiom:

```
Axiom UIP
: forall (A : Type) (x y : A) (p1 p2 : x = y), p1 = p2
```

- The Uniqueness of Reflexive Identity Proofs (UIP-refl) axiom:

```
Axiom UIP_refl
: forall (A : Type) (x : A) (p : x = x), p = eq_refl x
```

Uniqueness of identity proofs

- The Uniqueness of Identity Proofs (UIP) axiom:

```
Axiom UIP
  : forall (A : Type) (x y : A) (p1 p2 : x = y), p1 = p2
```

- The Uniqueness of Reflexive Identity Proofs (UIP-refl) axiom:

```
Axiom UIP_refl
  : forall (A : Type) (x : A) (p : x = x), p = eq_refl x
```

These axioms are equivalent. They are not provable in Coq's logic but consistent with it.

Invariance by substitution of reflexive equality proofs

```
Axiom eq_rect_eq
: forall (A : Type) (a : A) (P : A -> Type)
    (t : P a) (p : a = a),
    t = eq_rect A a P t a p.
```

Invariance by substitution of reflexive equality proofs

```
Axiom eq_rect_eq
: forall (A : Type) (a : A) (P : A -> Type)
    (t : P a) (p : a = a),
    t = eq_rect A a P t a p.
```

- This axiom is equivalent to UIP.

Invariance by substitution of reflexive equality proofs

```
Axiom eq_rect_eq
: forall (A : Type) (a : A) (P : A -> Type)
    (t : P a) (p : a = a),
    t = eq_rect A a P t a p.
```

- This axiom is equivalent to UIP.
- This axiom is the one actually present as an axiom in Coq's standard library, with UIP and UIP-refl derived from it as theorems.

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.
- Compare the (definable) dependent eliminator for equality:

  ```
  eq_rect_dep
  : forall (A : Type) (x : A)
    (P : forall a : A, x = a -> Type),
    P x eq_refl -> forall (y : A) (e : x = y), P y e
  ```

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.
- Compare the (definable) dependent eliminator for equality:

  ```
  eq_rect_dep
  : forall (A : Type) (x : A)
    (P : forall a : A, x = a -> Type),
    P x eq_refl -> forall (y : A) (e : x = y), P y e
  ```

- Streicher's axiom K can be given a computational interpretation:

  `K A a P t (eq_refl a)` $\rightarrow_\iota$ `t`

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.
- Compare the (definable) dependent eliminator for equality:

  ```
  eq_rect_dep
  : forall (A : Type) (x : A)
    (P : forall a : A, x = a -> Type),
    P x eq_refl -> forall (y : A) (e : x = y), P y e
  ```

- Streicher's axiom K can be given a computational interpretation:

  `K A a P t (eq_refl a)` $\rightarrow_\iota$ `t`

- This rule holds definitionally in e.g. Agda, which makes working with dependent types a bit easier.

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.
- Compare the (definable) dependent eliminator for equality:

  ```
  eq_rect_dep
  : forall (A : Type) (x : A)
    (P : forall a : A, x = a -> Type),
    P x eq_refl -> forall (y : A) (e : x = y), P y e
  ```
- Streicher's axiom K can be given a computational interpretation:

  `K A a P t (eq_refl a)` $\rightarrow_\iota$ `t`
- This rule holds definitionally in e.g. Agda, which makes working with dependent types a bit easier.
- Agda's dependent pattern matching relies on Streicher's K.

Streicher's Axiom K

```
Axiom K : forall (A : Type) (x : A) (P : x = x -> Type),
          P (eq_refl x) -> forall p : x = x, P p.
```

- Streicher's axiom K is also equivalent to UIP.
- Compare the (definable) dependent eliminator for equality:

  ```
  eq_rect_dep
  : forall (A : Type) (x : A)
    (P : forall a : A, x = a -> Type),
    P x eq_refl -> forall (y : A) (e : x = y), P y e
  ```
- Streicher's axiom K can be given a computational interpretation:

  K A a P t (eq_refl a) $\rightarrow_\iota$ t
- This rule holds definitionally in e.g. Agda, which makes working with dependent types a bit easier.
- Agda's dependent pattern matching relies on Streicher's K.
- Streicher's axiom K is incompatible with some recent developments in type theory (univalence, Homotopy Type Theory).

UIP for types with decidable equality

- A type `A` has decidable equality if:

```
forall x y : A, {x = y} + {x <> y}
```

UIP for types with decidable equality

- A type `A` has decidable equality if:

  ```
  forall x y : A, {x = y} + {x <> y}
  ```

- In Coq, UIP is provable for types with decidable equality:

  ```
  Theorem UIP_dec
  : forall A : Type,
      (forall x y : A, {x = y} + {x <> y}) ->
      forall (x y : A) (p1 p2 : x = y), p1 = p2
  ```

Heterogeneous equality

- Propositional equality `eq` can be used to compare only elements of the same type.

Heterogeneous equality

- Propositional equality `eq` can be used to compare only elements of the same type.
- Equality between two elements a, b of two different types A, B cannot be stated in terms of `eq`.

Heterogeneous equality

- Propositional equality `eq` can be used to compare only elements of the same type.
- Equality between two elements a, b of two different types A, B cannot be stated in terms of `eq`. Not even when A is propositionally equal to B!

Heterogeneous equality

```
vapp : forall {A n m},
         vector A n -> vector A m -> vector A (n + m).

Lemma lem_vapp_nil {A} :
  forall n (v : vector A n), vapp v vnil = v.
```

Heterogeneous equality

```
vapp : forall {A n m},
         vector A n -> vector A m -> vector A (n + m).

Lemma lem_vapp_nil {A} :
  forall n (v : vector A n), vapp v vnil = v.
Error:
In environment
A : Type
n : nat
v : vector A n
The term "v" has type "vector A n" while it
is expected to have type "vector A (n + 0)".
```

John Major equality

```
Inductive JMeq (A : Type) (x : A)
  : forall B : Type, B -> Prop :=
| JMeq_refl : JMeq A x A x.

Arguments JMeq [A] _ [B].
Arguments JMeq_refl {A x}, [A] _.

Notation "x ~= y" := (JMeq x y) (at level 70).
```

John Major equality

```
Inductive JMeq (A : Type) (x : A)
  : forall B : Type, B -> Prop :=
| JMeq_refl : JMeq A x A x.

Arguments JMeq [A] _ [B].
Arguments JMeq_refl {A x}, [A] _.

Notation "x ~= y" := (JMeq x y) (at level 70).
```

- John Major equality enables us to state equality between elements in two different types.

John Major equality

```
Inductive JMeq (A : Type) (x : A)
  : forall B : Type, B -> Prop :=
| JMeq_refl : JMeq A x A x.

Arguments JMeq [A] _ [B].
Arguments JMeq_refl {A x}, [A] _.

Notation "x ~= y" := (JMeq x y) (at level 70).
```

- John Major equality enables us to state equality between elements in two different types.
- However, we may use John Major equality only when the two types are actually the same:

```
JMeq_ind : forall (A : Type) (x : A) (P : A -> Prop),
                P x -> forall y : A, x ~= y -> P y
```

John Major equality

```
Inductive JMeq (A : Type) (x : A)
  : forall B : Type, B -> Prop :=
| JMeq_refl : JMeq A x A x.

Arguments JMeq [A] _ [B].
Arguments JMeq_refl {A x}, [A] _.

Notation "x ~= y" := (JMeq x y) (at level 70).
```

- John Major equality enables us to state equality between elements in two different types.
- However, we may use John Major equality only when the two types are actually the same:

  ```
  JMeq_ind : forall (A : Type) (x : A) (P : A -> Prop),
                    P x -> forall y : A, x ~= y -> P y
  ```

- `JMeq_ind` is defined using `JMeq_eq`:

  ```
  JMeq_eq : forall (A : Type) (x y : A), x ~= y -> x = y
  ```

John Major equality

```
Inductive JMeq (A : Type) (x : A)
  : forall B : Type, B -> Prop :=
| JMeq_refl : JMeq A x A x.

Arguments JMeq [A] _ [B].
Arguments JMeq_refl {A x}, [A] _.

Notation "x ~= y" := (JMeq x y) (at level 70).
```

- John Major equality enables us to state equality between elements in two different types.
- However, we may use John Major equality only when the two types are actually the same:

  ```
  JMeq_ind : forall (A : Type) (x : A) (P : A -> Prop),
                  P x -> forall y : A, x ~= y -> P y
  ```

- `JMeq_ind` is defined using `JMeq_eq`:

  ```
  JMeq_eq : forall (A : Type) (x y : A), x ~= y -> x = y
  ```

- `JMeq_eq` is an axiom equivalent to UIP.

John Major equality

This works:

```
vapp : forall {A n m},
         vector A n -> vector A m -> vector A (n + m).

Lemma lem_vapp_nil {A} :
  forall n (v : vector A n), vapp v vnil ~= v.
```

John Major equality

> *John Major's "classless society" widened people's aspirations to equality, but also the gap between rich and poor. (...) In much the same way, `JMeq` forms equations between members of any type, but they cannot be treated as equals (i.e. substituted) unless they are of the same type. Just as before, each thing is only equal to itself.*

Conor McBride, "Dependently Typed Functional Programs and their Proofs", PhD thesis, 1999

Dependent equality

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
| eq_dep_intro : eq_dep U P p x p x

eq_dep_ind =
fun (U : Type) (P : U -> Type) (p : U) (x : P p)
  (Q : forall q : U, P q -> Prop) (f : Q p x) (q : U)
  (y : P q) (e : eq_dep U P p x q y) =>
match e in eq_dep _ _ _ _ q' y' return Q q' y' with
| eq_dep_intro _ _ _ _ => f
end
: forall (U : Type) (P : U -> Type) (p : U) (x : P p)
    (Q : forall q : U, P q -> Prop),
    Q p x -> forall (q : U) (y : P q),
    eq_dep U P p x q y -> Q q y
```

Dependent equality

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
| eq_dep_intro : eq_dep U P p x p x

eq_dep_ind =
fun (U : Type) (P : U -> Type) (p : U) (x : P p)
  (Q : forall q : U, P q -> Prop) (f : Q p x) (q : U)
  (y : P q) (e : eq_dep U P p x q y) =>
match e in eq_dep _ _ _ _ q' y' return Q q' y' with
| eq_dep_intro _ _ _ _ => f
end
: forall (U : Type) (P : U -> Type) (p : U) (x : P p)
    (Q : forall q : U, P q -> Prop),
    Q p x -> forall (q : U) (y : P q),
    eq_dep U P p x q y -> Q q y
```

The eliminator `eq_dep_ind` does not depend on any axioms. We may rewrite dependent equalities without UIP.

Dependent equality

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
| eq_dep_intro : eq_dep U P p x p x
```

To convert `eq_dep` to `eq` we need the axiom

```
eq_dep_eq
: forall (U : Type) (P : U -> Type) (p : U) (x y : P p),
    eq_dep U P p x p y -> x = y
```

which is equivalent to UIP.

Dependent equality

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
| eq_dep_intro : eq_dep U P p x p x
```

- JMeq is equivalent to eq_dep Type (fun X => X).

Dependent equality

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
| eq_dep_intro : eq_dep U P p x p x
```

- `JMeq` is equivalent to `eq_dep Type (fun X => X)`.
- `eq_dep` is strictly finer than `JMeq`:

```
forall U P p q (x : P p) (y : P q),
  eq_dep U P p x q y -> x ~= y.

exists U P p q (x : P p) (y : P q),
  x ~= y /\ ~ eq_dep U P p x q y.
```