OTT
Observational Equality
CIC
The Calculus of Inductive Constructions
Loïc Pujet, Nicolas Tabareau
Both Coq and Lean are based on the CIC

- Dependent type theory
- with an infinite universe hierarchy,
- an impredicative sort for propositions
- and a powerful scheme for inductive definitions
The Calculus of Inductive Constructions

Both Coq and Lean are based on the CIC

- Dependent type theory
- with a infinite universe hierarchy,
- an impredicative sort for propositions
- and a powerful scheme for inductive definitions

But difficulties with function extensionality and quotient types
Observational Equality

In observational type theories\(^1\) the inductive equality is replaced with the observational equality:

\[
\Gamma \vdash t : A \quad \Gamma \vdash u : A \\
\Gamma \vdash t \sim_A u : SProp
\]

---

\(^1\) Altenkirch, McBride, Swierstra '07 – Observational Equality, Now!
In observational type theories\(^1\) the inductive equality is replaced with the observational equality:

\[
\Gamma \vdash t : A \quad \Gamma \vdash u : A \\
\frac{}{\Gamma \vdash t \sim_A u : SProp}
\]

Observational equality is eliminated via typecasting:

\[
\Gamma \vdash e : A \sim B \quad \Gamma \vdash t : A \\
\frac{}{\Gamma \vdash \text{cast}(A, B, e, t) : B}
\]

which computes by case analysis on A and B.

\(^1\)Altenkirch, McBride, Swierstra '07 – Observational Equality, Now!
In ordinary dependent type theory each type former comes with
- a type formation rule
- introduction rules
- elimination rules
- computation rules
Observational Equality

In ordinary dependent type theory each type former comes with

- a type formation rule
- introduction rules
- elimination rules
- computation rules

In observational type theory, every type former is also equipped with

- a definition for the equality between inhabitants
- a definition for the equality between two instances of the type
- computation rules for type-casting
Observational Equality

Let us look at the example of (nondependent) function types:

\[ f \sim A \to B \iff \Pi(x : A). f x \sim B \]

\[ \text{former} \ (A \to B) \sim \text{Type} \ (C \to D) \iff \ (C \sim \text{Type} \ A) \land (B \sim \text{Type} \ D) \]

\[ \text{cast}(A \to B, C \to D, e, f) \equiv \lambda(x : C). \text{cast}(B, D, e, f \text{cast}(C, A, e, x)) \]
Observational Equality

Let us look at the example of (nondependent) function types:

- A definition for the equality between inhabitants
  \[ f \sim_{A \to B} g \leftrightarrow \Pi(x : A). f x \sim_B g x \]
Observational Equality

Let us look at the example of (nondependent) function types:

- A definition for the equality between inhabitants
  \[ f \sim_{A \rightarrow B} g \iff \prod(x : A). f \, x \sim_B g \, x \]

- A definition for the equality between two instances of the type former
  \[ (A \rightarrow B) \sim_{Type} (C \rightarrow D) \iff (C \sim_{Type} A) \land (B \sim_{Type} D) \]
Observational Equality

Let us look at the example of (nondependent) function types:

- A definition for the equality between inhabitants
  \[ f \sim_{A \to B} g \iff \Pi(x : A). f x \sim_B g x \]

- A definition for the equality between two instances of the type former
  \[ (A \to B) \sim_{\text{Type}} (C \to D) \iff (C \sim_{\text{Type}} A) \land (B \sim_{\text{Type}} D) \]

- A computation rule for type-casting
  \[
  \text{cast}(A \to B, C \to D, e, f) \\
  \equiv \lambda (x : C). \text{cast}(B, D, e_2, f \text{cast}(C, A, e_1, x))
  \]
Observational Inductives?

Observational type theory is compatible with...
Observational Inductives?

Observational type theory is compatible with:

- Dependent products
Observational Inductives?

Observational type theory is compatible with
- Dependent products
- Universes
Observational Inductives?

Observational type theory is compatible with

- Dependent products
- Universes
- Impredicative strict Prop
Observational Inductives?

Observational type theory is compatible with
- Dependent products
- Universes
- Impredicative strict Prop
- $\Sigma$-types, natural numbers
Observational Inductives?

Observational type theory is compatible with
- Dependent products
- Universes
- Impredicative strict Prop
- $\Sigma$-types, natural numbers

How do we fit the general inductive definitions of CIC into this picture?
First Example: Lists

Inductive list (A : Typeℓ) : Typeℓ :=
| nil : list A
| cons : A → list A → list A

When should two inhabitants of list A be equal?
The J eliminator already gives the correct answer!

When should list A and list B be equal types?
:list−eq : list A ∼ list B → A ∼ B.

How does type-casting compute?
cast (list A, list B, e, nil) ≡ nil
cast (list A, list B, e, cons a l) ≡
  cons (cast (A, B, list−eq e, a) (cast (list A, list B, e, l)))
First Example: Lists

*Inductive* list (A : Type_ℓ) : Type_ℓ :=
| nil : list A
| cons : A → list A → list A

▶ When should two inhabitants of *list A* be equal?
**Inductive list** \( (A : Type_\ell) : Type_\ell := \)

\[
\begin{align*}
nil : list A \\
cons : A \rightarrow list A \rightarrow list A
\end{align*}
\]

- When should two inhabitants of \( list A \) be equal?

The J eliminator already gives the correct answer!
First Example: Lists

Inductive list (A : Typeₜ) : Typeₜ :=
| nil : list A
| cons : A → list A → list A

- When should two inhabitants of \textit{list A} be equal?
  The J eliminator already gives the correct answer!
- When should \textit{list A} and \textit{list B} be equal types?
First Example: Lists

\[
\text{Inductive } \text{list} \ (A : \text{Type}_\ell) : \text{Type}_\ell := \\
| \text{nil} : \text{list} \ A \\
| \text{cons} : A \to \text{list} \ A \to \text{list} \ A
\]

- When should two inhabitants of \( \text{list} \ A \) be equal? The J eliminator already gives the correct answer!
- When should \( \text{list} \ A \) and \( \text{list} \ B \) be equal types?

\[
\text{list-eq} : \text{list} \ A \sim \text{list} \ B \to A \sim B.
\]
First Example: Lists

Inductive list (A : Type₀) : Type₀ :=
| nil : list A
| cons : A → list A → list A

► When should two inhabitants of list A be equal?
The J eliminator already gives the correct answer!

► When should list A and list B be equal types?
list-eq : list A ~ list B → A ~ B.

► How does type-casting compute?
First Example: Lists

\texttt{Inductive list (A : Type}_\ell\texttt{) : Type}_\ell\texttt{ :=}
\texttt{| nil : list A}
\texttt{| cons : A \to list A \to list A}

- When should two inhabitants of \texttt{list A} be equal? The J eliminator already gives the correct answer!
- When should \texttt{list A} and \texttt{list B} be equal types? \texttt{list\texttt{-}eq : list A \sim list B \to A \sim B}.
- How does type-casting compute?
\[
\texttt{cast(list A, list B, e, nil) } \equiv \texttt{ nil}
\]
\[
\texttt{cast(list A, list B, e, cons a l) } \equiv
\texttt{ cons cast(A, B, list\texttt{-}eq e, a) cast(list A, list B, e, l)}
\]
Second Example: Inductive Equality

\[
\text{Inductive } eq \ (A : Type_\ell)(x : A) : A \to Type_0 := \\
| eq\_refl : eq A x x
\]
Second Example: Inductive Equality

\[
\text{Inductive } \text{eq } (A : \text{Type}_\ell)(x : A) : A \to \text{Type}_0 := \\
| \text{eq}_\text{refl} : \text{eq } A x x
\]

- When should two inhabitants of \( \text{eq } A x y \) be equal?
Second Example: Inductive Equality

\[
\text{Inductive } eq \ (A : Type_\ell)(x : A) : A \to Type_0 := \\
\mid eq\_refl : eq A x x
\]

- When should two inhabitants of \(eq A x y\) be equal?

The J eliminator does not seem to be sufficient to prove UIP 😞
Second Example: Inductive Equality

\[
\text{Inductive } \text{eq} (A : \text{Type}_\ell)(x : A) : A \to \text{Type}_0 := \begin{array}{l}
| \text{eq_refl} : \text{eq} A x x
\end{array}
\]

- When should two inhabitants of \( \text{eq} A x y \) be equal?
  The J eliminator does not seem to be sufficient to prove UIP 😞

- When should \( \text{eq} A x y \) and \( \text{eq} A' x' y' \) be equal types?
Second Example: Inductive Equality

\[
\text{Inductive } eq \ (A : Type_\ell)(x : A) : A \to Type_0 := \\
| eq\_refl : eq A x x
\]

- When should two inhabitants of \( eq A x y \) be equal?
  The J eliminator does not seem to be sufficient to prove UIP ☹

- When should \( eq A x y \) and \( eq A' x' y' \) be equal types?
  Equality of the parameters and indices?
Second Example: Inductive Equality

\[ \text{Inductive eq (A : Type}_\ell\text{)(x : A) : A → Type}_0\text{ :=} \]
\[ | \text{eq_refl : eq A x x} \]

- When should two inhabitants of \( \text{eq A x y} \) be equal?
  The J eliminator does not seem to be sufficient to prove UIP 😞

- When should \( \text{eq A x y} \) and \( \text{eq A' x' y'} \) be equal types?
  Equality of the parameters and indices?
  \( \rightarrow \) \( \text{eq} \) becomes an injective function from \( \text{Type}_\ell \) to \( \text{Type}_0 \)
Second Example: Inductive Equality

\[\text{Inductive } eq \ (A : Type_\ell) (x : A) : A \rightarrow Type_0 :=
\begin{align*}
| \text{eq_refl} & : eq A \times x \times x
\end{align*}\]

- When should two inhabitants of \(eq A \times y\) be equal?

The J eliminator does not seem to be sufficient to prove UIP 😞

- When should \(eq A \times y\) and \(eq A' \times x' \times y'\) be equal types?

Equality of the parameters and indices?
  \(\rightarrow eq\) becomes an injective function from \(Type_\ell\) to \(Type_0\)
  \(\rightarrow\) universe inconsistencies 😞
Second Example: Inductive Equality

\[
\text{Inductive } eq \ (A : Type_\ell)(x : A) : A \to Type_0 :=
\]
\[
\mid \text{eq_refl} : eq A \ x \ x
\]

- When should two inhabitants of \(eq A \ x \ y\) be equal?
  The J eliminator does not seem to be sufficient to prove UIP ☹

- When should \(eq A \ x \ y\) and \(eq A' \ x' \ y'\) be equal types?
  Equality of the parameters and indices?
  \(\to eq\) becomes an injective function from \(Type_\ell\) to \(Type_0\)
  \(\to\) universe inconsistencies ☹

- How does type-casting compute?
Second Example: Inductive Equality

\[
\text{Inductive } \text{eq } (A : \text{Type}_\ell)(x : A) : A \rightarrow \text{Type}_0 := \\
| \text{eq}_\text{refl} : \text{eq } A \times x \\
\]

- When should two inhabitants of \( \text{eq } A \times y \) be equal?
  
  The J eliminator does not seem to be sufficient to prove UIP 😞

- When should \( \text{eq } A \times y \) and \( \text{eq } A' \times x' \times y' \) be equal types?
  
  Equality of the parameters and indices?
  
  → \( \text{eq} \) becomes an injective function from \( \text{Type}_\ell \) to \( \text{Type}_0 \)
  
  → universe inconsistencies 😞

- How does type-casting compute?

\[
\text{cast}(\text{eq } A \times x, \text{eq } A' \times x' \times y', e, \text{eq}_\text{refl}) \equiv \text{eq}_\text{refl}
\]
Second Example: Inductive Equality

\textbf{Inductive} \textit{eq} (A : \textit{Type}_\ell)(x : A) : A \rightarrow \textit{Type}_0 := \ \\
\mid \textit{eq\_refl} : \textit{eq} A \times x

\begin{itemize}
\item When should two inhabitants of \textit{eq} A x y be equal?
The J eliminator does not seem to be sufficient to prove UIP ☹
\item When should \textit{eq} A x y and \textit{eq} A' x' y' be equal types?
Equality of the parameters and indices?
\rightarrow \textit{eq} becomes an injective function from \textit{Type}_\ell to \textit{Type}_0
\rightarrow \text{universe inconsistencies ☹}
\item How does type-casting compute?
\begin{align*}
\text{cast}(\textit{eq} A x x, \textit{eq} A' x' y', e, \textit{eq\_refl}) & \equiv \textit{eq\_refl}
\end{align*}
\end{itemize}

Does not typecheck ☹
Observational Inductives?

Not so simple!
The universe inconsistency shows up because the size of an inductive is determined by the types of its constructor arguments, not parameters or indices.

\[
\text{Inductive Small } (A : \text{Type}_ℓ) : \text{Type}_0 := \\
| \text{small} : \mathbb{N} \rightarrow \text{Small } A
\]

\[
\text{Inductive Large } (A : \text{Type}_ℓ) : \text{Type}_ℓ := \\
| \text{large} : A \rightarrow \text{Large } A
\]
Constructor arguments, not parameters!

The universe inconsistency shows up because the size of an inductive is determined by the types of its constructor arguments, not parameters or indices.

\[
\text{Inductive } \text{Small} \ (A : \text{Type}_\ell) : \text{Type}_0 := \\
| \text{small} : \mathbb{N} \rightarrow \text{Small} \ A \\
\text{Inductive } \text{Large} \ (A : \text{Type}_\ell) : \text{Type}_\ell := \\
| \text{large} : A \rightarrow \text{Large} \ A
\]
Constructor arguments, not parameters!

Lazy way out: bump up the universe levels of the inductives according to their parameters and indices.
Constructor arguments, not parameters!

Lazy way out: bump up the universe levels of the inductives according to their parameters and indices.

- reasonable for indices (cf HoTT)
Constructor arguments, not parameters!

Lazy way out: bump up the universe levels of the inductives according to their parameters and indices.

- reasonable for indices (cf HoTT)
- unacceptable for parameters!
Constructor arguments, not parameters!

Better way out: equality of inductive types should imply the equality of the types of the constructor arguments.
Better way out: equality of inductive types should imply the equality of the types of the constructor arguments.

\[
\text{Inductive } \text{Small} \ (A : \text{Type}_\ell) : \text{Type}_0 := \\
\mid \text{small} : \mathbb{N} \to \text{Small} A
\]

\[
eq\text{-Small} : \text{Small} A \sim \text{Small} B \to \mathbb{N} \sim \mathbb{N}
\]
Better way out: equality of inductive types should imply the equality of the types of the constructor arguments.

**Inductive Small** $(A : Type_\ell) : Type_0 :=$
| $small : \mathbb{N} \rightarrow Small A$

$eq\text{-}Small : Small A \sim Small B \rightarrow \mathbb{N} \sim \mathbb{N}$

**Inductive Large** $(A : Type_\ell) : Type_\ell :=$
| $large : A \rightarrow Large A$

$eq\text{-}Large : Large A \sim Large B \rightarrow A \sim B$
Better way out: equality of inductive types should imply the equality of the types of the constructor arguments.

\[\text{Inductive Small } (A : \text{Type}_\ell) : \text{Type}_0 :=
\]
\[| \text{small} : \mathbb{N} \rightarrow \text{Small} A\]

\[\text{eq-Small} : \text{Small} A \sim \text{Small} B \rightarrow \mathbb{N} \sim \mathbb{N}\]

\[\text{Inductive Large } (A : \text{Type}_\ell) : \text{Type}_\ell :=
\]
\[| \text{large} : A \rightarrow \text{Large} A\]

\[\text{eq-Large} : \text{Large} A \sim \text{Large} B \rightarrow A \sim B\]

\[\text{cast(Small} A, \text{Small} B, e, \text{small} n) \equiv \text{small cast(}\mathbb{N}, \mathbb{N}, \text{eq-Small} e, n)\]

\[\text{cast(Large} A, \text{Large} B, e, \text{large} x) \equiv \text{large cast(A,B, eq-Large} e, x)\]
With this technique, we can smoothly handle all inductive definitions without indices.

**Inductive** `eq (A : Type_ℓ)(x : A) : Π (x : A). Type₀ := ℎeq_refl : eq A x x`
Observational Inductives?

With this technique, we can smoothly handle all inductive definitions without indices.

\[
\text{Inductive } \text{eq } (A : \text{Type}) (x : A) : \prod (x : A). \text{Type}_0 := |
\text{eq\_refl} : \text{eq } A x x
\]

Treating indices will require a few more tricks.
No Canonicity for Indices

Remember our failed attempt at a computation rule

\[
\text{Inductive \ } eq \ (A : Type_\ell)(x : A) : A \to Type_0 :=
\]
\[
\mid eq\_refl : eq A x x
\]

\[
\text{cast}(eq A x x, eq A' x' y', e, eq\_refl) \equiv eq\_refl
\]
No Canonicity for Indices

Remember our failed attempt at a computation rule

\[
\text{Inductive } eq \ (A : Type_\ell)(x : A) : A \to Type_0 := \\
| \text{eq_refl} : eq A \ x \ x
\]

\[
\text{cast}(eq A \ x \ x, eq A' \ x' \ y', e, eq_refl) \equiv eq\_refl
\]

We cannot simplify casts on indices in general...
No Canonicity for Indices

Remember our failed attempt at a computation rule

\[
\text{Inductive } \text{eq } (A : \text{Type}_\ell)(x : A) : A \to \text{Type}_0 := \\
\mid \text{eq_refl} : \text{eq } A \times x
\]

\[
\text{cast}(\text{eq } A \times x, \text{eq } A' \times y', e, \text{eq_refl}) \equiv \text{eq_refl}
\]

We cannot simplify casts on indices in general...

...but we can encode them away with observational equality
"You can pick any colour, as long as it is black"

Henry Ford's trick\(^2\) can be used to encode indices with equalities on parameters:

---

\(^2\)Altenkirch, McBride '06 – Towards Observational Type Theory
"You can pick any colour, as long as it is black"

Henry Ford's trick\textsuperscript{2} can be used to encode indices with equalities on parameters:

\begin{verbatim}
Inductive vector (A : Type) : \IN \to Type :=
| vnil : vector A 0
| vcons : \Pi (m : \IN). A \to vector A m \to vector A (S m)
\end{verbatim}

\textsuperscript{2}Altenkirch, McBride '06 – Towards Observational Type Theory
"You can pick any colour, as long as it is black"

Henry Ford's trick\(^2\) can be used to encode indices with equalities on parameters:

\[
\text{Inductive vector } (A : \text{Type}_\ell) : \mathbb{N} \to \text{Type}_\ell :=
\begin{align*}
| \text{vnil} &: \text{vector } A \ 0 \\
| \text{vcons} &: \Pi (m : \mathbb{N}). A \to \text{vector } A \ m \to \text{vector } A \ (S \ m)
\end{align*}
\]

becomes

\[
\text{Inductive vector}_F \ (A : \text{Type}_\ell)(n : \mathbb{N}) : \text{Type}_\ell :=
\begin{align*}
| \text{vnil}_F &: (n \sim 0) \to \text{vector}_F A \ n \\
| \text{vcons}_F &: \Pi (m : \mathbb{N}). A \to \text{vector}_F A \ m \to (n \sim S \ m) \to \text{vector}_F A \ n
\end{align*}
\]

\(^2\)Altenkirch, McBride '06 – Towards Observational Type Theory
Henry Ford's trick\textsuperscript{2} can be used to encode indices with equalities on parameters:

\[
\text{Inductive vector } (A : Type_\ell) : \mathbb{N} \to Type_\ell := \\
| \text{vnil} : \text{vector } A 0 \\
| \text{vcons} : \Pi (m : \mathbb{N}). A \to \text{vector } A m \to \text{vector } A (S m)
\]

becomes

\[
\text{Inductive vector}_F (A : Type_\ell)(n : \mathbb{N}) : Type_\ell := \\
| \text{vnil}_F : (n \sim 0) \to \text{vector}_F A n \\
| \text{vcons}_F : \Pi (m : \mathbb{N}). A \to \text{vector}_F A m \to (n \sim S m) \to \text{vector}_F A n
\]

and now we can used our recipe for inductives without indices.

\textsuperscript{2}Altenkirch, McBride '06 – Towards Observational Type Theory
"You can pick any colour, as long as it is black"

In the case of the inductive equality, Henry Ford's encoding produces:

\[
\text{Inductive } eq_F (A : Type) (x : A) (y : A) : Type_0 := \\
\mid eq_{\text{refl}}_F : x \sim_A y \rightarrow eq_F A x y
\]
"You can pick any colour, as long as it is black"

In the case of the inductive equality, Henry Ford's encoding produces:

\[
\text{Inductive } eq_F (A : Type_\ell)(x : A)(y : A) : Type_0 := \\
| \text{eq\_refl}_F : x \sim_A y \to eq_F A x y \\
\]

→ an inhabitant of the inductive equality packs a hidden proof of the observational equality!
In the case of the inductive equality, Henry Ford's encoding produces:

\[
\text{Inductive } eq_F \ (A : Type_\ell)(x : A)(y : A) : Type_0 :=
| eq\_refl_F : x \sim_A y \rightarrow eq_F A x y
\]

→ an inhabitant of the inductive equality packs a hidden proof of the observational equality!

Our strategy: present \( eq \) to the user but elaborate everything to \( eq_F \) under the hood.

\[
\begin{align*}
  eq & \rightsquigarrow eq_F \\
  eq\_refl & \rightsquigarrow eq\_refl_F \ refl \\
  \ldots
\end{align*}
\]
"You can pick any colour, as long as it is black"

eq\_elim \leadsto ... 

We can write a term with the expected type using \textit{cast} and \textit{eqF\_elim}
"You can pick any colour, as long as it is black"

\[ eq_{\text{elim}} \rightsquigarrow \ldots \]

We can write a term with the expected type using \textit{cast} and \textit{eq}_{F\_elim}

However, the \texttt{computation rule} is not preserved: \textit{cast} only computes on closed types, while \textit{eq}_{\text{elim}} can compute even when the return type is open.
"You can pick any colour, as long as it is black"

\[ \text{eq\_elim} \leadsto \ldots \]

We can write a term with the expected type using \textit{cast} and \textit{eq\_elim}

However, the \textit{computation rule} is not preserved: \textit{cast} only computes on closed types, while \textit{eq\_elim} can compute even when the return type is open.

The missing ingredient is the computation rule for \textit{cast} on reflexivity:

\[ \text{cast}(A, A, \text{refl}, t) \equiv t \]
The missing rule

Our goal: adding \( \text{cast}(A, A, \text{refl}, t) \equiv t \) as a **definitional** equality
Our goal: adding \( \text{cast}(A, A, \text{refl}, t) \equiv t \) as a \textit{definitional} equality.

Because of proof irrelevance, it should apply whenever the two endpoints of the cast are convertible:

\[
\frac{\Gamma \vdash A \equiv B}{\Gamma \vdash \text{cast}(A, B, e, t) \equiv t : B x}
\]
The missing rule

Our goal: adding \( \text{cast}(A, A, \text{refl}, t) \equiv t \) as a \text{definitional} equality

Because of proof irrelevance, it should apply whenever the two endpoints of the cast are convertible:

\[
\Gamma \vdash A \equiv B \\
\Gamma \vdash \text{cast}(A, B, e, t) \equiv t : B \times
\]

→ nonlinear reduction rule which specifies \text{reduction} mutually with \text{conversion checking}
Is this déjà vu?

This idea is reminiscent of Lean's treatment of the J eliminator:

\[
P \ a \equiv P \ b \\
\frac{P \ a \equiv P \ b}{J(A, a, P, t, b, e) \equiv t}
\]
Is this déjà vu?

This idea is reminiscent of Lean's treatment of the J eliminator:

\[
P\ a \equiv P\ b \\
\frac{P\ a \equiv P\ b}{J(A, a, P, t, b, e) \equiv t}
\]

⚠ Abel, Coquand '19 – Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality
This idea is reminiscent of Lean's treatment of the J eliminator:

\[ \begin{align*}
  P \ a & \equiv P \ b \\
  \_ & \\
  J(A, a, P, t, b, e) & \equiv t
\end{align*} \]

⚠️ Abel, Coquand '19 – Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality

This is not an undecidability result, though

Does the addition of Werner’s rule, while destroying proof normalization, preserve decidability of conversion and type checking? (Since proofs are irrelevant for equality, they need not be normalized during type checking.)
The conversion checking algorithm

Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.
Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.

- we can implement it as a nonlinear reduction rule.
Because cast reduces on type constructors, this rule only plays a role for **relevant neutral terms**.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.
Because cast reduces on type constructors, this rule only plays a role for \textit{relevant neutral terms}.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.

\textbf{input: $t$ and $u$ (neutral terms)}
The conversion checking algorithm

Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.

input: t and u (neutral terms) → head-reduce t and u is one of them a cast?

no

yes

yes
The conversion checking algorithm

Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.

input: t and u (neutral terms) → head-reduce t and u is one of them a cast?
                      no    proceed as usual with t and u
Because cast reduces on type constructors, this rule only plays a role for **relevant neutral terms**.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.

**The conversion checking algorithm**

- **input**: \( t \) and \( u \) (neutral terms)
- **head-reduce** \( t \) and \( u \)
  - is one of them a cast?
    - **yes**: is \( A \) convertible to \( B \)? (recursive call)
    - **no**: proceed as usual with \( t \) and \( u \)
The conversion checking algorithm

Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.

▶ we can implement it as a nonlinear reduction rule
▶ or we can offload it to the conversion checker for neutral terms.

input: t and u (neutral terms) → head-reduce t and u

is one of them a cast? yes → is A convertible to B? (recursive call) yes → recursive call on t' and u

no → proceed as usual with t and u
The conversion checking algorithm

Because cast reduces on type constructors, this rule only plays a role for relevant neutral terms.

- we can implement it as a nonlinear reduction rule
- or we can offload it to the conversion checker for neutral terms.

```
input: t and u (neutral terms)
head-reduce t and u is one of them a cast?
  yes
  is A convertible to B? (recursive call)
    yes
    recursive call on t' and u
    no
  no
  proceed as usual with t and u

output no
```
Decidability proof

Using the second approach, we can add it on top of our logical relation model for $\text{TT}_{\text{obs}} / \text{CC}_{\text{obs}}$.

---

³P, Tabareau ’22 – Observational Equality: Now for good
Decidability proof

Using the second approach, we can add it on top of our logical relation model for $\text{TT}^{obs}/\text{CC}^{obs}$.

→ Formal Agda proof of the decidability of conversion for our new rule

---

3P, Tabareau '22 – Observational Equality: Now for good
Coming soon-ish

In your favourite rooster-themed proof assistant!

- Set Observational Inductives.

(* Declaring an inductive automatically adds equalities and rewrite rules for cast *)
Inductive list (A : Type) : Type :=
| nil : list A
| cons : forall (a : A) (l : list A), list A.

Parameter A B : Type.
Parameter e : list A ~ list B.
Parameter a : A.

Eval cbn in (cast (list A) (list B) e [ a ]).
(* [ cast A B (obseq_cons_∅ A B e) a ] *)