Coq without Casts: A complete proof of Coq Modulo Theory

Jean-Pierre Jouannaud and Pierre-Yves Strub
LIX, Ecole Polytechnique, Université Paris-Saclay

LPAR, May 12th, 2017
Motivation and Goal
Workflow of Coq

CoqTop

User

Libraries and Tactics

Type Check

Type Infer.

Conversion Check

Kernel

Rich libraries and tactics provide strong functionality, small kernel ensures reliability. Conversion is purely intensional to ensure decidability.
Workflow of Coq

CoqTop

User

Libraries and Tactics

Type Check

Type Infer.

Conversion Check

Kernel

Γ ⊢ p i : P i

Γ ⊢ P i ≃ P i′ : u

Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Conversion is purely intensional to ensure decidability.
Rich libraries and tactics provide strong functionality,
Rich libraries and tactics provide strong functionality,
Rich libraries and tactics provide strong functionality,
Workflow of Coq

- Rich libraries and tactics provide strong functionality,
- Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Workflow of Coq

- Rich libraries and tactics provide strong functionality,
- Small kernel ensures reliability.

\[ \Gamma \vdash p_i : P_i' \]
\[ \Gamma \vdash P_i \simeq P_i' : u \]
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Rich libraries and tactics provide strong functionality,
Small kernel ensures reliability.
Conversion is purely intensional to ensure decidability.
Problems with Dependent Types

```
Inductive dword : nat -> Type :=
  | dword0 : dword 0
  | dword1 : T -> dword 1
  | dwordA : forall n p, dword n -> dword p -> dword (n + p).
```
Problems with Dependent Types

**Inductive** `dword : nat -> Type :=
| dword0 : dword 0
| dword1 : T -> dword 1
| dwordA : forall n p, dword n -> dword p -> dword (n + p).

**Fixpoint** `rev` `n (ds : dword n) : (dword n) :=
match ds with
| dword0 => dword0
| dword1 x => dword1 x
| dwordA n1 n2 dw1 dw2 =>
  dwordA (rev dw2) (rev dw1)
end.
Problems with Dependent Types

**Inductive** dword : nat -> Type :=
  | dword0 : dword 0
  | dword1 : T -> dword 1
  | dwordA : forall n p, dword n -> dword p -> dword (n + p).

**Fixpoint** rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 (*dword 0*) => dword0 (*dword 0*)
  | dword1 x => dword1 x
  | dwordA n1 n2 dw1 dw2 => dwordA (rev dw2) (rev dw1)
  end.
Problems with Dependent Types

\texttt{Inductive} dword : nat \to Type :=
\hspace{1em} | dword0 : dword 0
\hspace{1em} | dword1 : T \to dword 1
\hspace{1em} | dwordA : \texttt{forall} n p, dword n \to dword p \to dword (n + p).

\texttt{Fixpoint} rev n (ds : dword n) : (dword n) :=
\hspace{1em} \texttt{match} ds \texttt{with}
\hspace{2em} | dword0 \hspace{2em} \Rightarrow \hspace{2em} dword0
\hspace{2em} | dword1 x (*dword 1*) \hspace{2em} \Rightarrow \hspace{2em} dword1 x (*dword 1*)
\hspace{2em} | dwordA n1 n2 dw1 dw2 \hspace{2em} \Rightarrow \hspace{2em} dwordA (rev dw2) (rev dw1)
\hspace{1em} \texttt{end}.
Problems with Dependent Types

\[
\text{Inductive} \quad \text{dword} : \text{nat} \to \text{Type} := \\
\mid \text{dword0} : \text{dword 0} \\
\mid \text{dword1} : T \to \text{dword 1} \\
\mid \text{dwordA} : \forall n p, \text{dword } n \to \text{dword } p \rightarrow \text{dword } (n + p).
\]

\[
\text{Fixpoint} \quad \text{rev } n (ds : \text{dword } n) : (\text{dword } n) := \\
\text{match } ds \text{ with } \\
\mid \text{dword0} \Rightarrow \text{dword0} \\
\mid \text{dword1 } x \Rightarrow \text{dword1 } x \\
\mid \text{dwordA } n1 n2 dw1 dw2 (*\text{dword } n1+n2*) \Rightarrow \\
\quad \text{dwordA } (\text{rev } dw2) (\text{rev } dw1) (*\text{dword } n2+n1*) \\
\text{end}.
\]
Problems with Dependent Types

**Inductive** dword : nat -> Type :=
  | dword0 : dword 0
  | dword1 : T -> dword 1
  | dwordA : forall n p, dword n -> dword p -> dword (n + p).

**Fixpoint** rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0               => dword0
  | dword1 x            => dword1 x
  | dwordA n1 n2 dw1 dw2 =>
    dwordA (rev dw2) (rev dw1)
  end.
Solution in Coq

```coq
Fixpoint rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 ds1 ds2 =>
    dwordA (rev ds2) (rev ds1)
  end.
```

J.-P. Jouannaud, Pierr-Yves Strub
LIX, Ecole Polytechnique, Université Paris-Saclay
Motivation and Goal

Problems using dependent types

Solution in Coq

```
Fixpoint rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 ds1 ds2 =>
    dwordA (rev ds2) (rev ds1)
  end.
```

- Force an extra user’s proof of equality.
Solution in Coq

```
Fixpoint rev n (ds : dword n) : (dword n) :=
    match ds with
    | dword0 => dword0
    | dword1 x => dword1 x
    | dwordA n1 n2 ds1 ds2 =>
        dwordA (rev ds2) (rev ds1)
    end.
```

```
Definition cast: forall m n, m=n->dword m->dword n.
```

- Force an extra user’s proof of equality.
Solution in Coq

```
Fixpoint rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 ds1 ds2 =>
    cast (addC n2 n1) (dwordA (rev ds2) (rev ds1))
  end.

Definition cast: forall m n, m=n->dword m->dword n.
```

- Force an extra user's proof of equality.
Motivation and Goal

Problems using dependent types

Coq Modulo Theory

Meta-Theory of Coq Modulo Theory

Solution in Coq

Fixpoint rev n (ds : dword n) : (dword n) :=
    match ds with
    | dword0 => dword0
    | dword1 x => dword1 x
    | dwordA n1 n2 ds1 ds2 =>
        cast (addC n2 n1) (dwordA (rev ds2) (rev ds1))
    end.

Definition cast: forall m n, m=n->dword m->dword n.

- Force an extra user’s proof of equality.
- The proof is carried out repeatedly at runtime.
Solution in Coq

\[
\text{Fixpoint \ rev \ n \ (ds : \ dword \ n) : (dword \ n) :=}
\]
\[
\text{match \ ds \ with}
\]
\[
| \ dword0 \ => \ dword0
\]
\[
| \ dword1 \ x \ => \ dword1 \ x
\]
\[
| \ dwordA \ n1 \ n2 \ ds1 \ ds2 \ =>
\]
\[
\text{cast (addC} \ n2 \ n1) \ (dwordA \ (rev \ ds2) \ (rev \ ds1))
\]
\[
\text{end.}
\]

\[
\text{Definition \ cast: \ forall \ m \ n, \ m=n->dword \ m->dword \ n.}
\]

- Force an \textit{extra user’s proof} of equality.
- The proof is carried out repeatedly \textit{at runtime}.
- Equality generates explicit computations in proofs.
The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[ \Gamma \vdash t : P' \quad \Gamma \vdash P \simeq P' : s \]

\[ \Gamma \vdash t : P \]
The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[
\frac{\Gamma \vdash t : P' \quad \Gamma \vdash P \simeq P' : s}{\Gamma \vdash t : P}
\]

\simeq\text{ is the closure of computations (\(\rightarrow\)), which is \textit{intensional}. Conversion is \textit{decided} thanks to the Church-Rosser property:
The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[
\Gamma \vdash t : P' \quad \Gamma \vdash P \simeq P' : s \\
\overline{\quad} \\
\Gamma \vdash t : P
\]

\(\simeq\) is the closure of computations (\(\rightarrow\)), which is intensional. Conversion is decided thanks to the Church-Rosser property:

\[
P \leftrightarrow P_1 \quad \cdots \quad P_n \leftrightarrow P'
\]
The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[ \Gamma \vdash t : P' \quad \Gamma \vdash P \rightsquigarrow P' : s \]
\[ \Gamma \vdash t : P \]

\rightsquigarrow is the closure of computations (\(\rightarrow\)), which is intensional. Conversion is decided thanks to the Church-Rosser property:

\[ P \leftrightarrow P_1 \quad \ldots \quad P_n \leftrightarrow P' \]

\[ P \Downarrow = P' \Downarrow \]
Where does the problem originate from?

- The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[
\begin{align*}
&\Gamma \vdash t : P' \\
&\Gamma \vdash P \simeq P' : s \\
\hline
&\Gamma \vdash t : P
\end{align*}
\]

- \(\simeq\) is the closure of computations (\(\rightarrow\)), which is intensional. Conversion is decided thanks to the Church-Rosser property:

\[
\begin{align*}
P &\leftrightarrow P_1 & \cdots & P_n &\leftrightarrow P' \\
&\downarrow \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \downarrow \\
P \downarrow = P' \downarrow \\
n1 + n2 \downarrow \\
n1 + n2
\end{align*}
\]
Where does the problem originate from?

- The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[
\Gamma \vdash t : P' \quad \Gamma \vdash P \simeq P' : s
\]

\[
\Gamma \vdash t : P
\]

- \(\simeq\) is the closure of computations (\(\rightarrow\)), which is intensional. Conversion is decided thanks to the Church-Rosser property:

\[
P \leftrightarrow P_1 \quad \cdots \quad P_n \leftrightarrow P'
\]

\[
P \downarrow = P' \downarrow
\]

\[
n1 + n2 \quad n2 + n1
\]

\[
n1 + n2 \quad n2 + n1
\]
Where does the problem originate from?

- The kernel implements the Calculus of Inductive Constructions (CIC): the conversion rule carries out the conversion check:

\[
\frac{\Gamma \vdash t : P' \quad \Gamma \vdash P \simeq P' : s}{\Gamma \vdash t : P}
\]

- \(\simeq\) is the closure of computations (\(\to\)), which is intensional. Conversion is decided thanks to the Church-Rosser property:

\[
P \leftrightarrow P_1 \quad \cdots \quad P_n \leftrightarrow P' \quad n1 + n2 \quad n2 + n1
\]

\[
P \downarrow = P' \downarrow \quad n1 + n2 \quad \neq n2 + n1
\]
Content

2 Coq Modulo Theory
Solution: CoqMT

- Build in \textit{decidable equational} theories!
- A decision procedure \textit{automatically} checks equality in the theory.
Solution: CoqMT

- Build in **decidable equational theories**!
- A decision procedure **automatically** checks equality in the theory.
Solution: CoqMT

- Build in **decidable equational theories**!
- A decision procedure **automatically** checks equality in the theory.
Solution: CoqMT

- Build in **decidable equational** theories!
- A decision procedure **automatically** checks equality in the theory.
Solution: CoqMT

- Build in **decidable equational theories**!
- A decision procedure automatically checks equality in the theory.
Solution: CoqMT

- Build in *decidable equational theories*!
- A decision procedure *automatically* checks equality in the theory.
Inductive dword : nat -> Type :=
| dword0 : dword 0
| dword1 : T -> dword 1
| dwordA : forall n p, dword n -> dword p -> dword (n + p).
Type-checking dependent definitions in CoqMT

\textbf{Inductive} \texttt{dword : nat \rightarrow Type :=}
\begin{itemize}
  \item \texttt{dword0 : dword 0}
  \item \texttt{dword1 : T \rightarrow dword 1}
  \item \texttt{dwordA : forall n p, dword n \rightarrow dword p \rightarrow dword (n + p)}.
\end{itemize}

\textbf{Fixpoint} \texttt{rev n (ds : dword n) : (dword n) :=}
\begin{verbatim}
match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 dw1 dw2 =>
    dwordA (rev dw2) (rev dw1)
end.
\end{verbatim}
Type-checking dependent definitions in CoqMT

Inductive dword : nat -> Type :=
  | dword0 : dword 0
  | dword1 : T -> dword 1
  | dwordA : forall n p, dword n -> dword p -> dword (n + p).

Fixpoint rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 dw1 dw2 =>
    dwordA (rev dw2) (rev dw1)
  end.
**Type-checking dependent definitions in CoqMT**

```coq
Inductive dword : nat -> Type :=
| dword0 : dword 0
| dword1 : T -> dword 1
| dwordA : forall n p, dword n -> dword p -> dword (n + p).

Fixpoint rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x (*dword 1*) => dword1 x (*dword 1*)
  | dwordA n1 n2 dw1 dw2 =>
    dwordA (rev dw2) (rev dw1)
  end.
```
Type-checking dependent definitions in CoqMT

**Inductive** `dword : nat -> Type :=
  | dword0 : dword 0
  | dword1 : T -> dword 1
  | dwordA : forall n p, dword n -> dword p -> dword (n + p).

**Fixpoint** `rev n (ds : dword n) : (dword n) :=
  match ds with
  | dword0 => dword0
  | dword1 x => dword1 x
  | dwordA n1 n2 dw1 dw2 (*dword n1+n2*) =>
    dwordA (rev dw2) (rev dw1) (*dword n2+n1*)
  end.
Type-checking dependent definitions in CoqMT

\textbf{Inductive} dword : nat \rightarrow \textbf{Type} :=
\begin{itemize}
  \item dword0 : dword 0
  \item dword1 : T \rightarrow dword 1
  \item dwordA : \textbf{forall} n p, dword n \rightarrow dword p \rightarrow dword (n + p).
\end{itemize}

\textbf{Fixpoint} rev n (ds : dword n) : (dword n) :=
\begin{itemize}
  \textbf{match} ds \textbf{with}
  \item dword0 \rightarrow \text{dword0}
  \item dword1 x \rightarrow \text{dword1} x
  \item dwordA n1 n2 dw1 dw2 \rightarrow \\
    \text{dwordA} (\text{rev} dw2) (\text{rev} dw1)
\end{itemize}
\textbf{end}. 
In CoqMT the user can use a predefined theory $T$ or declare her own theory $T$ which is then dynamically downloaded.
Soundness of CoqMT

- In CoqMT the user can use a predefined theory $T$ or declare her/his own theory $T$ which is then dynamically downloaded.
- The type checker must return a result: decidability.
Soundness of CoqMT

- In CoqMT the user can use a predefined theory \( T \) or declare her own theory \( T \) which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
Soundness of CoqMT

- In CoqMT the user can use a predefined theory $T$ or declare her own theory $T$ which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
- Requirements needed to ensure the above two:
  - All computations terminate: strong normalization (SN).
Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare her own theory T which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
- Requirements needed to ensure the above two:
  - All computations terminate: strong normalization (SN).
  - Order of evaluation is arbitrary: confluence.
In CoqMT the user can use a predefined theory $T$ or declare her own theory $T$ which is then dynamically downloaded.

The type checker must return a result: decidability.

The result should be correct: consistency.

**Requirements** needed to ensure the above two:

- All computations terminate: strong normalization (SN).
- Order of evaluation is arbitrary: confluence.
- Computations preserve types: subject reduction (SR).
Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare her own theory T which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
- Requirements needed to ensure the above two:
  - All computations terminate: strong normalization (SN).
  - Order of evaluation is arbitrary: confluence.
  - Computations preserve types: subject reduction (SR).
  - Conversion is decidable: Church-Rosser modulo T (CRT).
Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare her own theory T which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
- **Requirements** needed to ensure the above two:
  - All computations terminate: strong normalization (SN).
  - Order of evaluation is arbitrary: confluence.
  - Computations preserve types: subject reduction (SR).
  - Conversion is decidable: Church-Rosser modulo T (CRT).

\[ m + (1 + 1) \times n \simeq 2 \times n + (2 - 1) \times m \]

- \( =_{T} \) should be correct: certification of \( =_{T} \).
Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare her own theory T which is then dynamically downloaded.
- The type checker must return a result: decidability.
- The result should be correct: consistency.
- Requirements needed to ensure the above two:
  - All computations terminate: strong normalization (SN).
  - Order of evaluation is arbitrary: confluence.
  - Computations preserve types: subject reduction (SR).
  - Conversion is decidable: Church-Rosser modulo T (CRT).

\[
m + (1 + 1) \times n \leadsto 2 \times n + (2 - 1) \times m
\]

- should be correct: certification of F. = T.
In CoqMT the user can use a predefined theory $T$ or declare her own theory $T$ which is then dynamically downloaded.

The type checker must return a result: decidability.

The result should be correct: consistency.

Requirements needed to ensure the above two:

- All computations terminate: strong normalization (SN).
- Order of evaluation is arbitrary: confluence.
- Computations preserve types: subject reduction (SR).
- Conversion is decidable: Church-Rosser modulo $T$ (CRT).

\[
m + (1 + 1) \ast n \simeq 2 \ast n + (2 - 1) \ast m
\]

\[
m + 2n \equiv_{\tau} 2n + m
\]

- $\equiv_{\tau}$ should be correct: certification of $\equiv_{\tau}$. 

Definition of $CIC^\omega(T)$

$CIC^\omega(T)$ contains $CIC^\omega$ and a $T$-inductive type $o$ of objects s.t. 

- $o$ is equipped with first-order constructors $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\text{Elim}_o$ of the usual type
**Definition of \( CIC^\omega(T) \)**

\( CIC^\omega(T) \) contains \( CIC^\omega \) and a \( T \)-inductive type \( o \) of objects s.t.

- \( o \) is equipped with first-order constructors \( C \), defined symbols \( D \), an equality \( =_T \), and an eliminator \( \text{Elim}_o \) of the usual type

- **Freeness**: ground constructor terms are all different (in \( =_T \))

\[\]
**Definition of \( CIC^\omega(T) \)**

\( CIC^\omega(T) \) contains \( CIC^\omega \) and a \( T \)-inductive type \( o \) of objects s.t.

- \( o \) is equipped with first-order constructors \( C \), defined symbols \( D \), an equality \( =_T \), and an eliminator \( \text{Elim}_o \) of the usual type

- **Freeness** : ground constructor terms are all different (in \( =_T \))

- **Non-triviality** : The ground constructor term algebra contains at least two elements
Definition of $CIC^\omega(T)$

$CIC^\omega(T)$ contains $CIC^\omega$ and a $T$-inductive type $o$ of objects s.t.

- $o$ is equipped with first-order constructors $C$, defined symbols $D$, an equality $=_T$, and an eliminator $\text{Elim}_o$ of the usual type

- **Freeness**: ground constructor terms are all different (in $=_T$)

- **Non-triviality**: The ground constructor term algebra contains at least two elements

- **Completeness**: the ground term algebra is isomorphic to the ground constructor term algebra
Definition of $CIC^\omega(T)$

$CIC^\omega(T)$ contains $CIC^\omega$ and a $T$-inductive type $o$ of objects s.t.

- $o$ is equipped with first-order constructors $C$, defined symbols $D$, an equality $\equiv_T$, and an eliminator $\text{Elim}_o$ of the usual type $F$.

- **Freeness**: ground constructor terms are all different (in $\equiv_T$).

- **Non-triviality**: The ground constructor term algebra contains at least two elements.

- **Completeness**: the ground term algebra is isomorphic to the ground constructor term algebra.

- $\equiv_T$ is decidable.
Definition of $CIC^\omega(T)$

$CIC^\omega(T)$ contains $CIC^\omega$ and a $T$-inductive type $o$ of objects s.t.

- $o$ is equipped with first-order constructors $C$, defined symbols $D$, an equality $=_T$, and an eliminator $\text{Elim}_o$ of the usual type
- **Freeness**: ground constructor terms are all different (in $=_T$)
- **Non-triviality**: The ground constructor term algebra contains at least two elements
- **Completeness**: the ground term algebra is isomorphic to the ground constructor term algebra
- $=_T$ is decidable
- Elimination has the usual typing rule

\[
\Gamma \vdash t : \text{nat} \quad \Gamma \vdash P : \forall [x : \text{nat}]. \{\text{Prop}, \text{Type}\} \\
\Gamma \vdash f_0 : P \ 0 \quad \Gamma \vdash f_S : \forall [x : \text{nat}]. (P x \to P (S x)) \\
\hline
\Gamma \vdash \text{ELIM}_{\text{nat}}(P, f_0, f_S, t) : P \ t
\]
Definition of $CIC_{\omega}(T)$

$CIC_{\omega}(T)$ contains $CIC_{\omega}$ and a $T$-inductive type $o$ of objects s.t.

- $o$ is equipped with first-order constructors $C$, defined symbols $D$, an equality $\equiv_T$, and an eliminator $\text{Elim}_o$ of the usual type
- **Freeness**: ground constructor terms are all different (in $\equiv_T$)
- **Non-triviality**: The ground constructor term algebra contains at least two elements
- **Completeness**: the ground term algebra is isomorphic to the ground constructor term algebra
- $\equiv_T$ is decidable
- Elimination has the usual typing rule

\[
\Gamma \vdash t : \text{nat} \quad \Gamma \vdash P : \forall[x : \text{nat}]. \{\text{Prop, Type}\} \\
\Gamma \vdash f_0 : P \ 0 \quad \Gamma \vdash f_S : \forall[x : \text{nat}]. (P \ x \to P \ (S \ x)) \\
\hline
\Gamma \vdash \text{Elim}_{\text{nat}}(P, f_0, f_S, t) : P \ t
\]

- Elimination rules match their argument of type $o$ modulo $\equiv_T$
Definition of $CIC^\omega(T)$: Pseudo-Terms

$u, v, U, V ::= \text{Prop} | \text{Type}_j$ (Universes)

| $\forall u v | \lambda [x : U]. v | \forall [x : U]. V$ (CC)
| $o | C | D | \text{ELIM}_o(U, \overrightarrow{u}, v)$ ($T$-Inductives)
Reductions and Conversion

- $\beta$-reduction is defined as usual:
  \[(\lambda[x : U]. v)u \rightarrow_\beta v[x \mapsto u]\]
Reductions and Conversion

- $\beta$-reduction is defined as usual:

$$ (\lambda[x : U]. v)u \rightarrow \beta v\{x \mapsto u\} $$

- $\iota$-reduction is the same as in CIC for normal inductive types.
Reductions and Conversion

- $\beta$-reduction is defined as usual:
  $$\lambda[x : U]. \nu)u \to \beta \nu\{x \mapsto u\}$$

- $\iota$-reduction is the same as in CIC for normal inductive types.

- $\iota_T$-reduction generalizes pure $\iota$-reduction. For “Presburger”
  $$\text{ELIM}_{\text{Nat}}(P, f_0, f_S, \nu) \to \iota_T \begin{cases} f_0 \quad (1) \\ f_S u \text{ ELIM}_{\text{Nat}}(P, f_0, f_S, u) \quad (2) \end{cases}$$
  provided
  - $\nu = T 0$ for case (1), and
  - exists $u$, $\nu = T S u$ and $\mathcal{V}(u) \subseteq \mathcal{V}(\nu)$ for case (2)
Reducions and Conversion

- β-reduction is defined as usual:

\[(\lambda[x : U]. v)u \rightarrow \beta v\{x \mapsto u\}\]

- ι-reduction is the same as in CIC for normal inductive types

- ι\(_T\)-reduction generalizes pure ι-reduction. For “Presburger”

\[\text{\textsc{Elim}}\textsubscript{Nat}(P, f_0, f_s, \nu) \rightarrow \iota\_T \begin{cases} 
  f_0 \\
  f_s \ u \ \text{\textsc{Elim}}\textsubscript{Nat}(P, f_0, f_s, u)
\end{cases}\]  

provided

- \(\nu = \_T 0\) for case (1), and
- exists \(u\), \(\nu = \_T S \ u\) and \(\mathcal{V}(u) \subseteq \mathcal{V}(\nu)\) for case (2)

- The conversion relation \(\simeq\) is the reflexive, symmetric and transitive closure of \(\rightarrow \beta \cup \rightarrow \iota\_T \cup \equiv \_T = \rightarrow \beta \cup \rightarrow \iota \cup = \_T\).
Reductions and Conversion

- $\beta$-reduction is defined as usual:

\[(\lambda[x : U]. \nu)u \rightarrow \beta \nu\{x \mapsto u\}\]

- $\iota$-reduction is the same as in CIC for normal inductive types

- $\iota_T$-reduction generalizes pure $\iota$-reduction. For “Presburger”

\[\text{ELIM}_{Nat}(P, f_0, f_S, \nu) \rightarrow \iota_T \left\{ f_0 \right\} f_S u \text{ELIM}_{Nat}(P, f_0, f_S, u) \]

provided

- $\nu =_T 0$ for case (1), and
- exists $u$, $\nu =_T S u$ and $\mathcal{V}(u) \subseteq \mathcal{V}(\nu)$ for case (2)

- The conversion relation $\simeq$ is the reflexive, symmetric and transitive closure of $\rightarrow \beta \cup \rightarrow \iota_T \cup =_T = \rightarrow \beta \cup \rightarrow \iota \cup =_T$.

- The typing rules are as usual.
3 Meta-Theory of Coq Modulo Theory
Consistency and DTC proofs of fragments of $CIC^\omega(T)$

- $CIC^\omega$: Paper proof of consistency and DTC
  B. Werner, "Sets in types, types in sets", in TACS : 1997

- $CIC^1(T)$: Implementation, paper proof of consistency and DTC,
  P.-Y. Strub, in CSL : 2010

- $CIC^\omega(T)$: Implementation, paper proof of consistency and DTC
  (restricted to weak-$T$-elimination)

- $CIC^\omega(T)$: Formal proof of Consistency,
  Barras, Wang, in CSL : 2012

- $CIC^\omega(T)$: Paper proof of DTC,
  Jouannaud, Strub, in LPAR : 2017

- $CIC^\omega(T)$: Implementation, on-going.
Strong normalization proof

Let $\mathcal{T} = \{ t \to C(u) : C(u) \text{ simplifies } t \}$

**Lemma**

$\mathcal{T}$ is a confluent and terminating rewriting system for $\leftrightarrow^*_{\mathcal{T}}$.

**Lemma**

$\to_{\beta \iota \mathcal{T}} \subseteq \to^+_{\beta \iota \mathcal{T}}$ where $\to_{\beta \iota \mathcal{T}} \overset{\text{def}}{=} \to_{\beta} \cup \to_{\iota} \cup \to_{\mathcal{T}}$.

We prove that $\to_{\beta \iota \mathcal{T}}$ is SN by induction over $\to_{\beta \iota} \cup \to_{\iota}$.

This proof uses syntactic arguments only, in particular the left-linearity of the rules in $\{\beta, \iota\}$ which provide with key commutation properties between $\{\beta, \iota\}$ and $\mathcal{T}$. 
Assume we have a type constructor **poly** : Type → Type such that poly K stands for the type of polynomials in 1 indeterminate over K, we can construct the type mpoly K n, of multinomials over n indeterminates over K as:

\[
\text{Fixpoint } \text{mpoly } (K : \text{ring}) \ (n : \text{nat}) : \text{Type} := \\
\text{match } n \text{ with } 0 \Rightarrow K \mid \text{S } p \Rightarrow \text{poly } (\text{mpoly } K \ p) .
\]
In the future version of CoqMT justified here, not only are \texttt{mpoly K (n+1+p)} and \texttt{mpoly K (p+n+1)} identified, which is not the case in Coq nor in the previous version of CoqMT, but because \texttt{(S (n+p))} simplifies \texttt{n+1+p} and \texttt{p+n+1}, they both compute to \texttt{poly (mpoly K (n+p))}, providing some canonical form of our initial type which highlights that \texttt{poly} is iterated at least once.

This would allow, for instance, to easily use properties on multivariate polynomials without relying on unnecessary type casts. Such needs arise quite naturally in the proof of the symmetric polynomials fundamental lemma, where all type casts occurring in the proof can be removed in CoqMT.
No type casts are ever needed in $\text{CoqMT}$ provided the decidable theory $T$ contains the necessary syntax to express all equalities on dependent types whose proofs are needed to type the user's development.
Conclusion: when are casts needed?

No type casts are ever needed in CoQMT provided the decidable theory $T$ contains the necessary syntax to express all equalities on dependent types whose proofs are needed to type the user’s development.

Casts become needed when the theory $T$ is undecidable.
Thank you for your attention