# Coq without Casts : A complete proof of Coq Modulo Theory

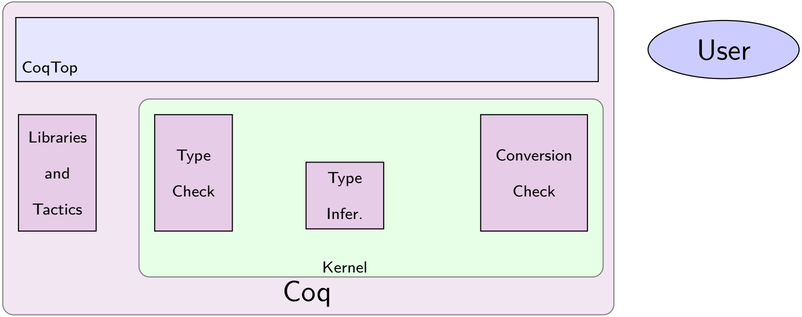
Jean-Pierre Jouannaud and Pierre-Yves Strub

LIX, Ecole Polytechnique, Université Paris-Saclay

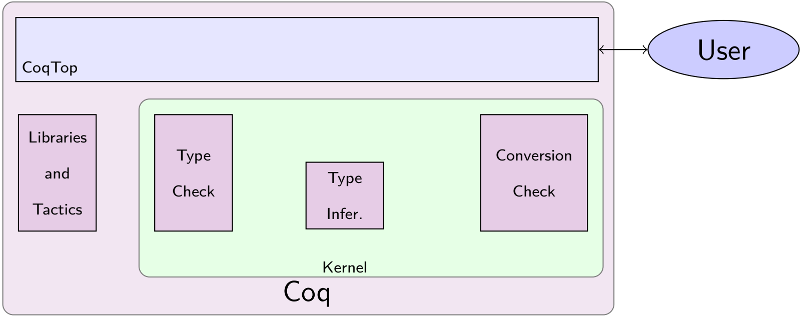
LPAR, May 12th, 2017

# Content

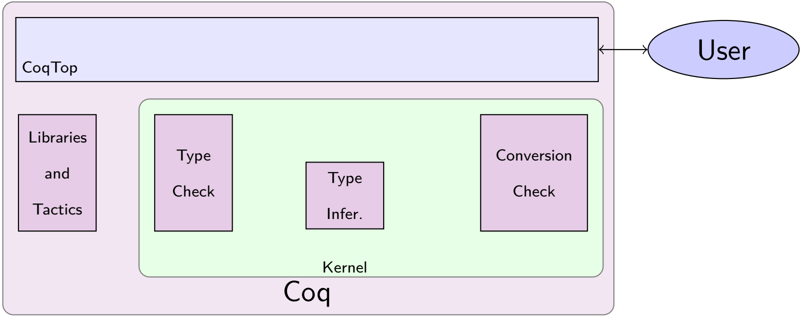
# Workflow of Coq



# Workflow of Coq

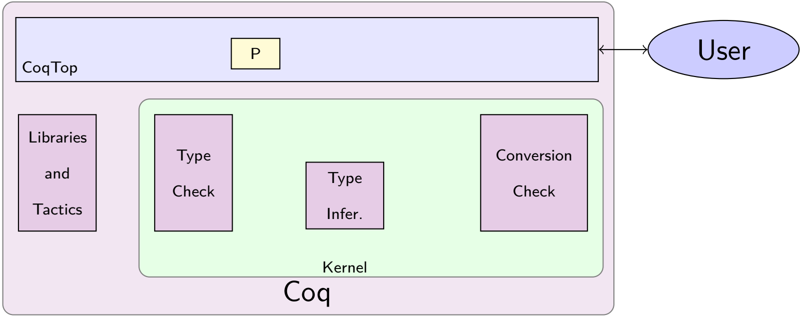


# Workflow of Coq



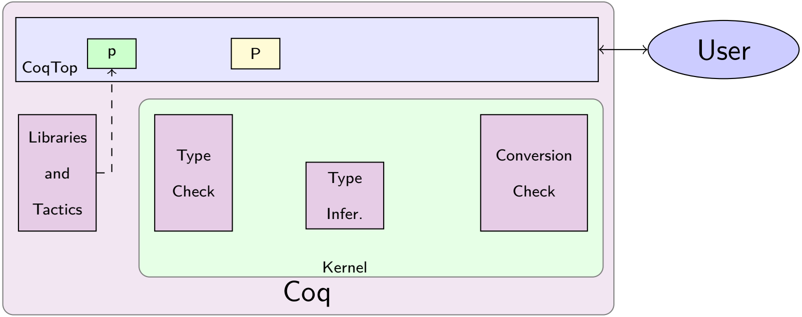

- Rich libraries and tactics provide strong functionality,

# Workflow of Coq



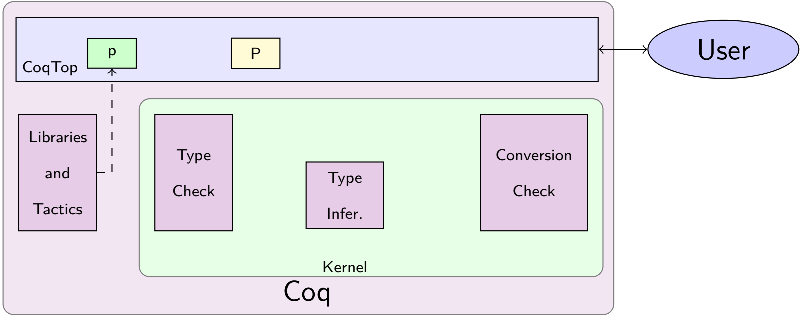

- Rich libraries and tactics provide strong functionality,

# Workflow of Coq



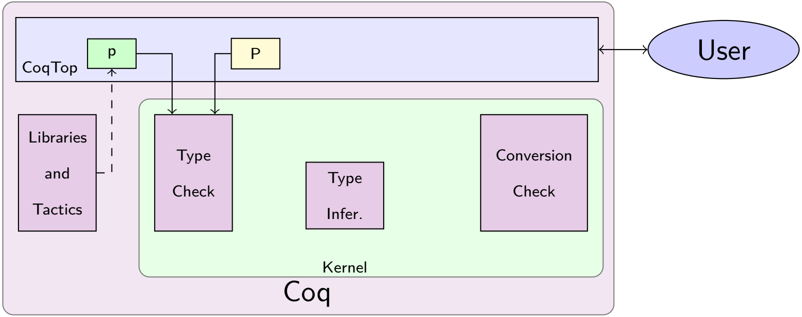

- Rich libraries and tactics provide strong functionality,

# Workflow of Coq



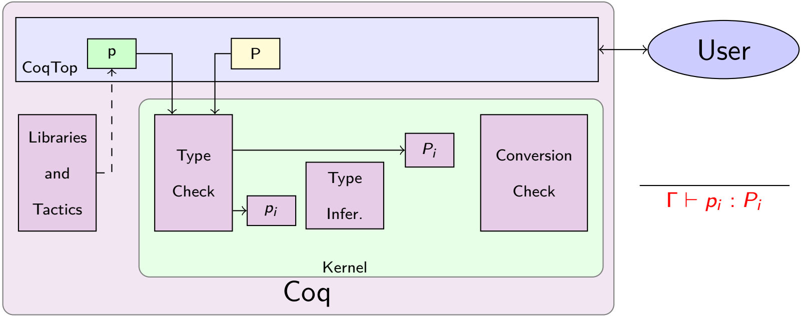

- Rich libraries and tactics provide strong functionality,
- Small kernel ensures reliability.

# Workflow of Coq



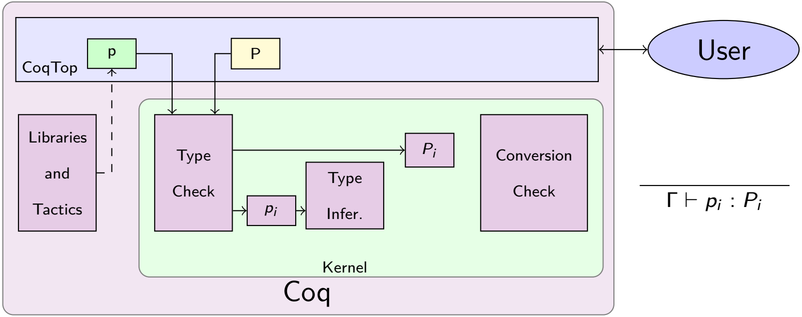

- Rich libraries and tactics provide strong functionality,
- Small kernel ensures reliability.

# Workflow of Coq




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

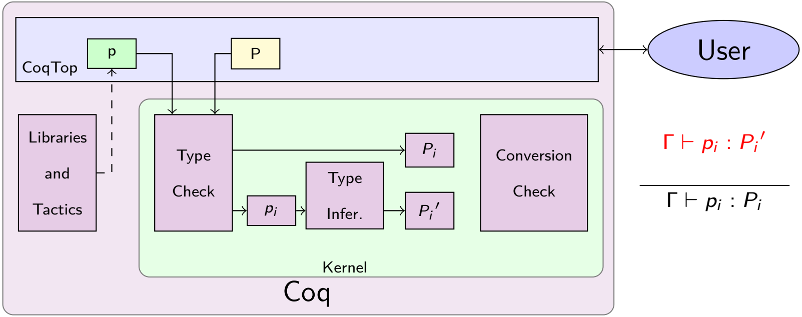
# Workflow of Coq




$$\frac{}{\Gamma \vdash p_i : P_i}$$

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

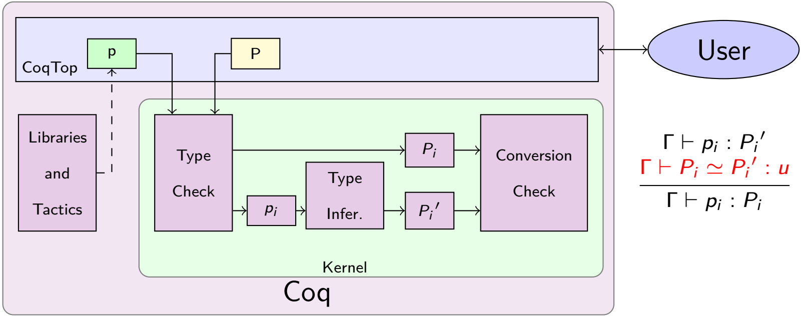
# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i{}'}{\Gamma \vdash p_i : P_i}$$

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

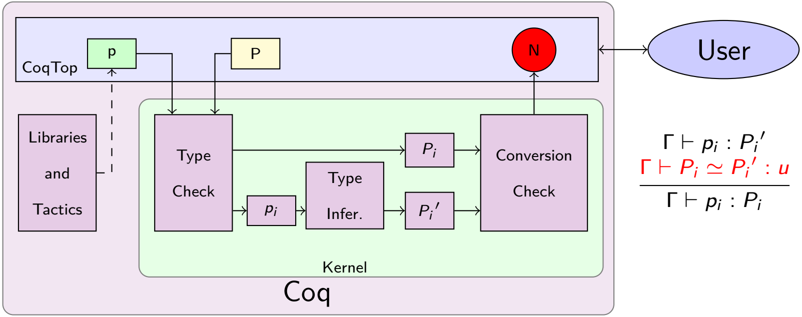
# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i{}' \qquad \color{red}{\Gamma \vdash P_i \simeq P_i{}' : u}}{\Gamma \vdash p_i : P_i}$$

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

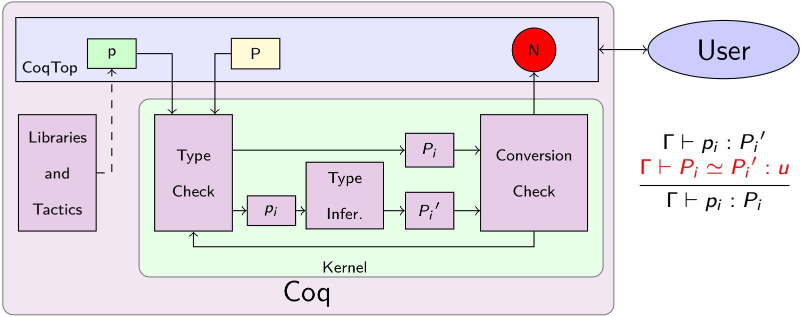
# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i{}' \qquad {\color{red}\Gamma \vdash P_i \simeq P_i{}' : u}}{\Gamma \vdash p_i : P_i}$$

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

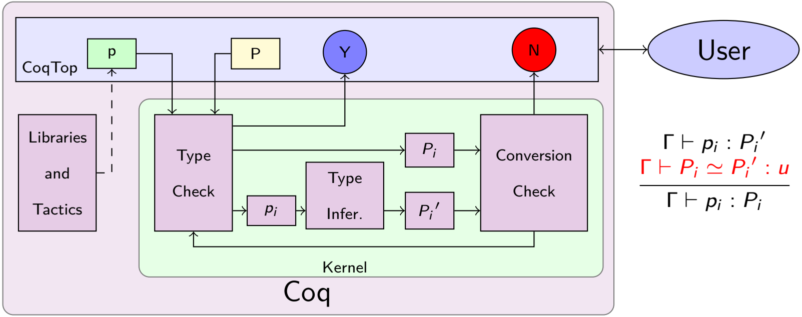
# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i' \qquad \textcolor{red}{\Gamma \vdash P_i \simeq P_i' : u}}{\Gamma \vdash p_i : P_i}$$

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

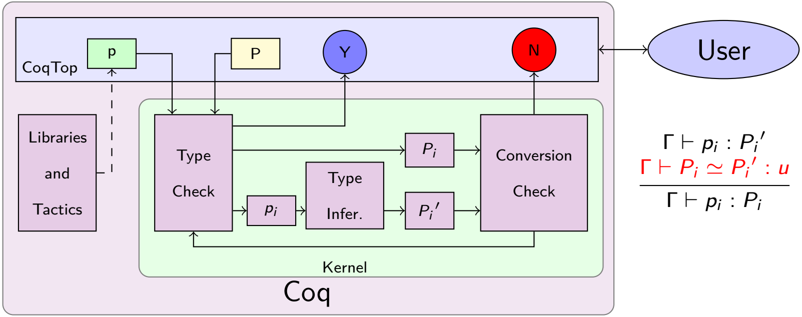
# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i' \qquad \color{red}{\Gamma \vdash P_i \simeq P_i' : u}}{\Gamma \vdash p_i : P_i}$$

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

# Workflow of Coq




$$\frac{\Gamma \vdash p_i : P_i{}' \qquad \Gamma \vdash P_i \simeq P_i{}' : u}{\Gamma \vdash p_i : P_i}$$

- 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

```
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.
```

# 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 (*dword n1+n2*) =>
       dwordA (rev dw2) (rev dw1) (*dword n2+n1*)
  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

```
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.
```

# 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.

# 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

```
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.
- Equality generates explicit computations in proofs.

# Where does the problem originate from ?

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

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

# 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 ($\rightarrow$ ), which is intensional. Conversion is decided thanks to the Church-Rosser property:

# 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 ($\rightarrow$ ), which is intensional. Conversion is decided thanks to the Church-Rosser property:

$$P \longleftrightarrow P_1 \quad \cdots \quad P_n \longleftrightarrow P'$$

# Where does the problem originate from ?

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

$$\frac{\Gamma \vdash t : P' \qquad \Gamma \vdash P \simeq P' : s}{\Gamma \vdash t : P}$$

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

$$\begin{tikzcd}
P \arrow[r, leftrightarrow] \arrow[drr] & P_1 & \cdots & P_n \arrow[r, leftrightarrow] & P' \arrow[dll] \\
 & & P \downarrow = P' \downarrow & &
\end{tikzcd}$$

# 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 ($\rightarrow$), which is intensional. Conversion is decided thanks to the Church-Rosser property:

$$\begin{tikzcd}
P \arrow[r, leftrightarrow] \arrow[drr] & P_1 & \cdots & P_n \arrow[r, leftrightarrow] & P' \arrow[dll] \\
 & & P \downarrow = P' \downarrow & &
\end{tikzcd}
\qquad
\begin{tikzcd}
n1 + n2 \arrow[d] \\
n1 + n2
\end{tikzcd}$$

# 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 ($\rightarrow$ ), which is intensional. Conversion is decided thanks to the Church-Rosser property:

$$\begin{tikzcd}
P \arrow[r, leftrightarrow] \arrow[drr] & P_1 & \cdots & P_n \arrow[r, leftrightarrow] & P' \arrow[dll] \\
 & & P\downarrow = P'\downarrow & &
\end{tikzcd}
\qquad
\begin{tikzcd}
n1 + n2 \arrow[d] & n2 + n1 \arrow[d] \\
n1 + n2 & n2 + n1
\end{tikzcd}$$

# 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 ($\rightarrow$ ), which is intensional. Conversion is decided thanks to the Church-Rosser property:

$$\begin{tikzcd}
P \arrow[r, leftrightarrow] \arrow[drr] & P_1 & \cdots & P_n \arrow[r, leftrightarrow] & P' \arrow[dll] \\
 & & P\downarrow = P'\downarrow & &
\end{tikzcd}
\qquad
\begin{tikzcd}
n1 + n2 \arrow[d] & n2 + n1 \arrow[d] \\
n1 + n2 \arrow[r, phantom, "\neq"] & n2 + n1
\end{tikzcd}$$

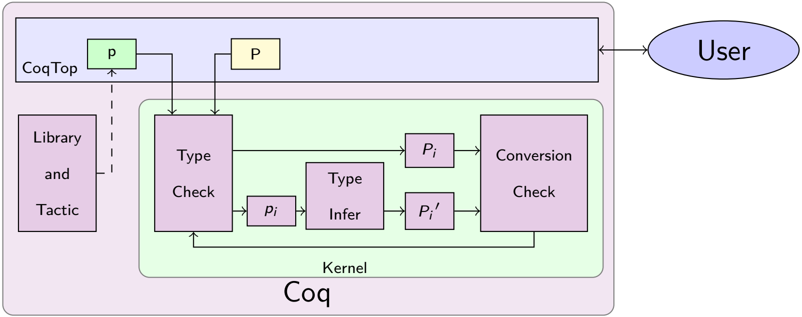
# Content

# Solution : CoqMT

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

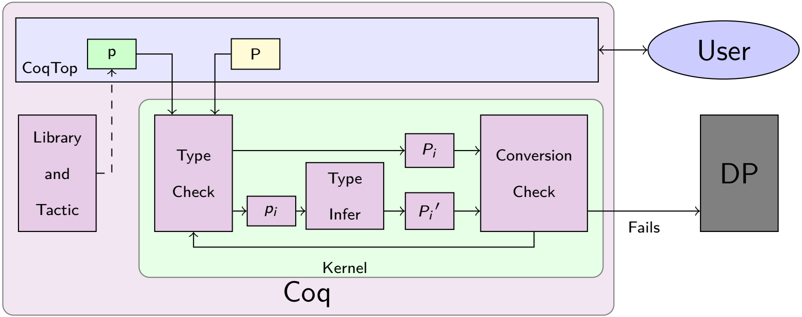
# Solution : CoqMT

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



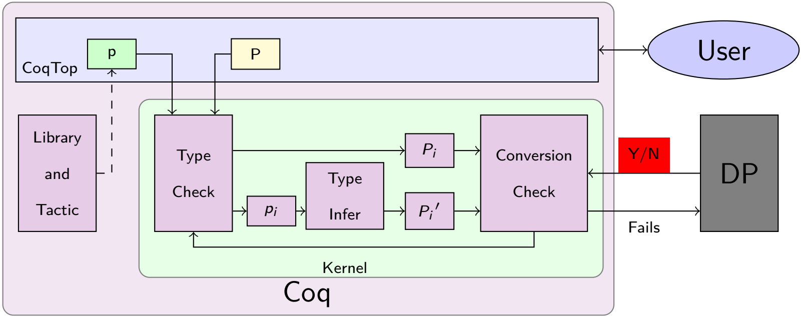
# Solution : CoqMT

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



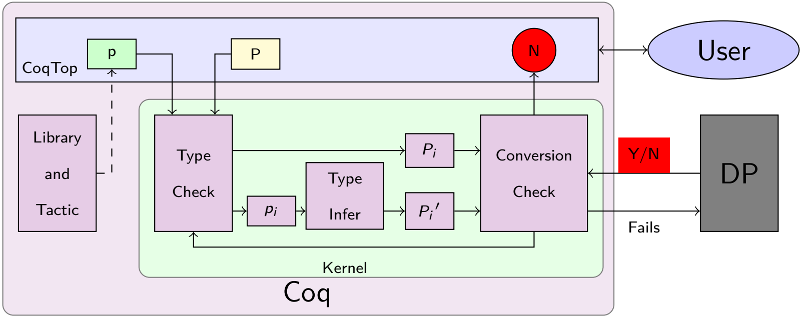
# Solution : CoqMT

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



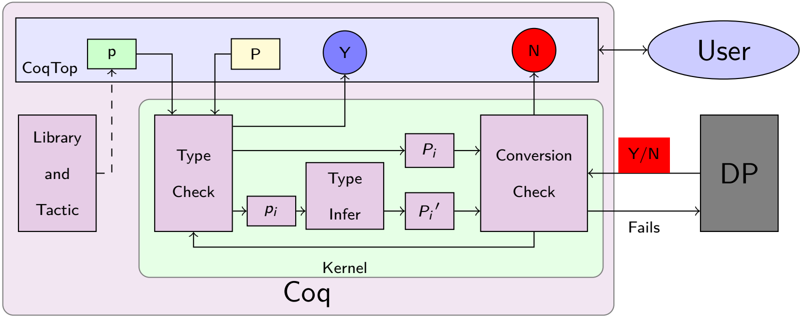
# Solution : CoqMT

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



# Solution : CoqMT

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



# 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).
```

# 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

```
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.
```

# 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 (*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

```
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.
```

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns a result: decidability.

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns a result: decidability.
- The result should be correct : consistency.

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns 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 heris own theory T which is then dynamically downloaded
- The type checker must returns 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.

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns 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 heris own theory T which is then dynamically downloaded
- The type checker must returns 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 heris own theory T which is then dynamically downloaded
- The type checker must returns 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).

$$m + (1+1) * n \quad \simeq \quad 2 * n + (2-1) * m$$

  - $=_T$ should be correct : certification of $=_T$.

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns 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).

$$\begin{tikzcd} m + (1+1) * n \arrow[d] & \simeq & 2 * n + (2-1) * m \arrow[d] \\ m + 2n & & 2n + m \end{tikzcd}$$

  - $=_T$ should be correct : certification of $=_T$.

# Soundness of CoqMT

- In CoqMT the user can use a predefined theory T or declare heris own theory T which is then dynamically downloaded
- The type checker must returns 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).

$$\begin{tikzcd} m+(1+1)*n \arrow[d] \arrow[r, phantom, "\simeq"] & 2*n+(2-1)*m \arrow[d] \\ m+2n \arrow[r, phantom, "=_T"] & 2n+m \end{tikzcd}$$

  - $=_T$ should be correct : certification of $=_T$.

# 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\textsc{Elim}_o$ 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\textsc{Elim}_o$ 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\textsc{Elim}_o$ 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\textsc{Elim}_o$ 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, an equality $=_T$, and an eliminator $\mathrm{E{\scriptsize LIM}}_o$ 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

# 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 $\textsc{Elim}_o$ 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

$$\frac{\Gamma \vdash t : \mathbf{nat} \quad \Gamma \vdash P : \forall[x : \mathbf{nat}].\,\{\mathbf{Prop}, \mathbf{Type}\} \qquad \Gamma \vdash f_{\mathbf{0}} : P\ \mathbf{0} \quad \Gamma \vdash f_{\mathbf{S}} : \forall[x : \mathbf{nat}].\,(P\ x \to P\ (\mathbf{S}\ x))}{\Gamma \vdash \textsc{Elim}_{nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, t) : P\ t}$$

# 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 $\mathcal{C}$, defined symbols $\mathcal{D}$, 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

$$\frac{\Gamma \vdash t : \mathbf{nat} \quad \Gamma \vdash P : \forall[x : \mathbf{nat}].\,\{\mathbf{Prop}, \mathbf{Type}\} \qquad \Gamma \vdash f_{\mathbf{0}} : P\ \mathbf{0} \quad \Gamma \vdash f_{\mathbf{S}} : \forall[x : \mathbf{nat}].\,(P\ x \to P\ (\mathbf{S}\ x))}{\Gamma \vdash \text{ELIM}_{nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, t) : P\ t}$$

- Elimination rules match their argument of type $o$ **modulo** $=_T$

# Definition of $CIC^{\omega}(T)$: Pseudo-Terms

$$
\begin{array}{rlr}
u, v, U, V ::= & \mathbf{Prop} \mid \mathbf{Type}_j & \text{(Universes)} \\
& \mid \mathcal{V} \mid u\ v \mid \lambda[x : U].\, v \mid \forall[x : U].\, V & \text{(CC)} \\
& \mid o \mid \mathcal{C} \mid \mathcal{D} \mid \textsc{Elim}_o(U, \overrightarrow{u}, v) & (T\text{-Inductives})
\end{array}
$$

## 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:

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

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

- $\iota_T$-reduction generalizes pure $\iota$-reduction. For "Presburger"

  $$\mathrm{E{\scriptstyle LIM}}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, v) \rightarrow \iota_T \begin{cases} f_{\mathbf{0}} & (1) \\ f_{\mathbf{S}}\ u\ \mathrm{E{\scriptstyle LIM}}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, u) & (2) \end{cases}$$

  provided

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

# 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

- $\iota_T$-reduction generalizes pure $\iota$-reduction. For "Presburger"

$$\mathrm{ELIM}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, v) \rightarrow \iota_T \begin{cases} f_{\mathbf{0}} & (1) \\ f_{\mathbf{S}}\ u\ \mathrm{ELIM}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, u) & (2) \end{cases}$$

  provided

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

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

# 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

- $\iota_T$-reduction generalizes pure $\iota$-reduction. For "Presburger"

$$\mathrm{E{\scriptsize LIM}}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, v) \rightarrow_{\iota_T} \begin{cases} f_{\mathbf{0}} & (1) \\ f_{\mathbf{S}}\, u\, \mathrm{E{\scriptsize LIM}}_{Nat}(P, f_{\mathbf{0}}, f_{\mathbf{S}}, u) & (2) \end{cases}$$

  provided

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

# Content

# 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)
  Barras, Jouannaud, Strub, Wang, "CoqMTU" in LICS : 2011
- $CIC^{\omega}(T)$: 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

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

**Lemma**

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

**Lemma**

$\longrightarrow_{\beta\iota_T} \subseteq \longrightarrow^+_{\beta\iota\mathcal{T}}$ *where* $\longrightarrow_{\beta\iota\mathcal{T}} \stackrel{\text{def}}{=} \longrightarrow_\beta \cup \longrightarrow_\iota \cup \longrightarrow_\mathcal{T}$.

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

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 $\mathcal{T}$.

# Is strong elimination modulo needed in practice ? (1/2)

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:

```
Fixpoint mpoly (K : ring) (n : nat) : Type :=
match n with 0 => K | S p => poly (mpoly K p).
```

# Is elimination modulo needed in practice ? (2/2)

In the future version of CoqMT justified here, not only are `mpoly K (n+1+p)` and `mpoly K (p+n+1)` identified, which is not the case in Coq nor in the previous version of CoqMT, but because `(S (n+p))` simplifies `n+1+p` and `p+n+1`, they both compute to `poly (mpoly K (n+p))`, providing some canonical form of our initial type which highlights that `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.

# 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.

# 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.

# Last slide

Thank you for your attention