Typing in Coq

Daria Walukiewicz-Chrząszcz

26 march 2019

Proving in Coq

Curry-Howard isomorphism

proof	of	a	given	formula
↕				↕
term	of the corresponding			type

$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz)\ :\ (A \to B \to C) \to (A \to B) \to (A \to C)$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof	of	a	given	formula
$\updownarrow$				$\updownarrow$
term	of the corresponding			type

$$\lambda x^{A\to B\to C} \lambda y^{A\to B} \lambda z^{A}\ xz(yz)\ :\ (A\to B\to C)\to(A\to B)\to(A\to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

proof	of	a	given	formula
$\updownarrow$				$\updownarrow$
term	of the corresponding			type

$$\lambda x^{A\to B\to C}\lambda y^{A\to B}\lambda z^{A}\ xz(yz)\ :\ (A\to B\to C)\to(A\to B)\to(A\to C)$$

Correctness of Coq relies on correctness of type-checking

Proving in Coq

Curry-Howard isomorphism

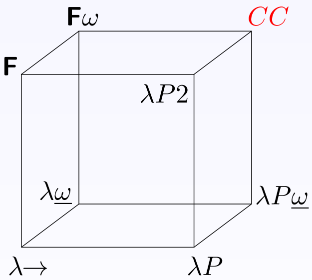
proof	of a given	formula
$\updownarrow$		$\updownarrow$
term	of the corresponding	type

$$\lambda x^{A \to B \to C} \lambda y^{A \to B} \lambda z^{A}\ xz(yz) \ : \ (A \to B \to C) \to (A \to B) \to (A \to C)$$

Correctness of Coq relies on correctness of type-checking

Coq — formalism

Coq — calculus of constructions (CC) + inductive definitions



- $\uparrow$ polimorphism
- $\nearrow$ type constructors
- $\rightarrow$ dependent types

Big picture of Coq architecture

The De Bruijn principle (“small” core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle (“small” core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Big picture of Coq architecture

The De Bruijn principle ("small" core, externally checkable terms)

- core / kernel (≈20KLOC), responsible for:
  - CIC typing
  - reduction
  - environment (definitions, axioms etc).
  - modules
- the rest (≈230KLOC), responsible for:
  - user interface
  - file management
  - sections
  - namespace management
  - proof mode (plus tactics, tactic language)
  - notations
  - implicit arguments (type reconstruction)
  - type classes
  - coercions and resolving mechanism
  - auto-generation of inductive principles
  - ...

Coq — a bit of history

- 1984 CoC - calculus of constructions - G. Huet, T. Coquand
- 1989 first public release (version 4.10)
- 1991 Coq - calculus of inductive constructions - C. Paulin (version 5.6)

⋮

- 2000 version 7.0 with new (safer) architecture
- 2003 version 7.4 with modules
- 2004 version 8.0 with new syntax
- 2009 version 8.2 with "type classes"
- 2012 version 8.4 with eta-reduction, structural proof syntax...
- 2018 version 8.7.2 — fixes a critical bug in the universes (present since 8.5)

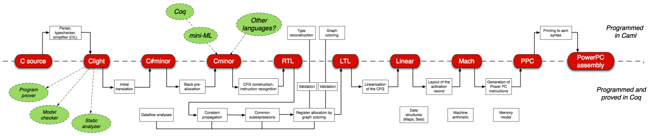
Coq — famous formalizations

- Fundamental theorem of algebra, Nijmegen 2000
- JavaCard Platform formalization, Trusted Logic 2003

  > *September 2007: a big step in program certification in the real world: The Technology and Innovation group at Gemalto has successfully completed a Common Criteria (CC) evaluation on a JavaCard based commercial product. This evaluation is the world's first CC certificate of a Java product involving EAL7 components.* (the official press release)

- Four color theorem, Cambridge 2004
- CompCert certified Clight compiler, 2008-now

  > *The main result of the project is the CompCert C verified compiler, a high-assurance compiler for almost all of the ISO C90 / ANSI C language, generating efficient code for the PowerPC, ARM and x86 processors.*



Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — programming language

- predicative sorts `Set` and `Type`
- abstraction and application
- inductive types,
- (structural) recursion
- polimorphism
- dependant types and dependent pattern-matching
- modules i functors
- type classes
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq — logic

- intuitionistic higher-order logic
- impredicative sort `Prop`
- forall and implication built-in
- boolean connectives, false, exists (defined)
- inductive predicates (including equality)
- ...

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Coq proof machinery

- interactive proof mode (goal management)
- built-in tactics (constructing a bit of proof-term): intro, apply, etc.
- automatic ad-hoc tactics: auto, intuition, etc.
- decision procedures: omega, ring, field, tauto, etc.
- tactic language (`Ltac mytactic:=...`)

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

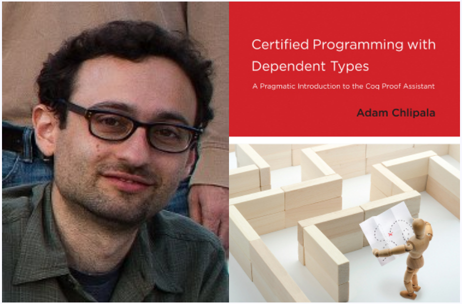
- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Program extraction

- program is extracted from the proof
- extracted program satisfies its specification *by definition*
- extraction — "elimination" of logical parts from the proof-term
- extraction possible because proofs are done in constructive logic (excluded-middle and double negation laws do not hold)
- target languages: O'Caml, Haskell, Scheme

Resources

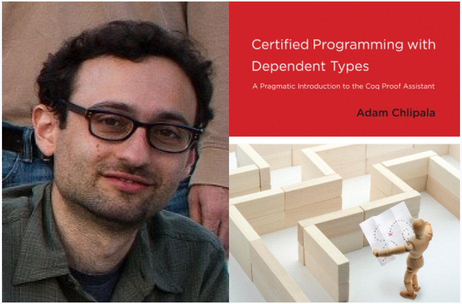
- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: http://adam.chlipala.net/cpdt/

Resources

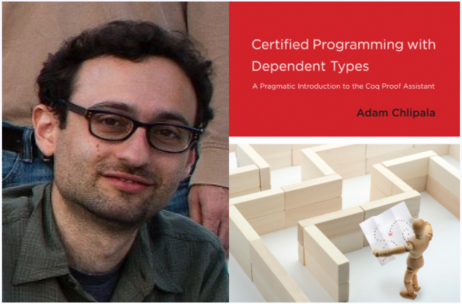
- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: http://adam.chlipala.net/cpdt/

Resources

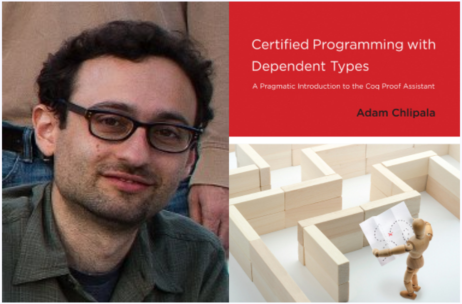
- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: http://adam.chlipala.net/cpdt/

Resources

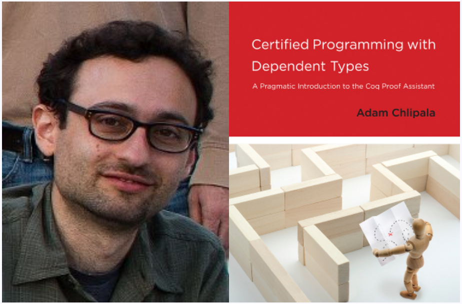
- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: http://adam.chlipala.net/cpdt/

Resources

- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)



editor: MIT Press 2013
accessible: http://adam.chlipala.net/cpdt/

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4 typing rules for inductive types

Coq — typing judgment

$$\texttt{environment} \vdash \texttt{term} : \texttt{type}$$

environment: global and local declarations and definitions

In Coq reference manual there are:

- 18 typing rules for CC,
- 4 typing rules for inductive types

Coq — sorts

- Sorts in Coq:

$$\begin{matrix} \textit{Prop} \\ \textit{Set} \end{matrix} \; : \textit{Type(1)}\!: \textit{Type(2)}\!: \ldots$$

- Cummulativity (or sub-sorting):

$$\textit{Prop} \leq \textit{Set} \leq \textit{Type(1)} \leq \textit{Type(2)} \leq \ldots$$

Coq — sorts

- Sorts in Coq:

$$\begin{matrix} \mathit{Prop} \\ \mathit{Set} \end{matrix} : \mathit{Type(1)} : \mathit{Type(2)} : \ldots$$

- Cummulativity (or sub-sorting):

$$\mathit{Prop} \leq \mathit{Set} \leq \mathit{Type(1)} \leq \mathit{Type(2)} \leq \ldots$$

Coq — sorts Prop and Set

$$d\colon A\colon Prop$$

A is a formula, d is a proof of A

$$n\colon T\colon Set$$

T is a type, n is a value of type T

Coq — sorts Prop and Set

$$d\colon A\colon Prop$$

`A` is a formula, `d` is a proof of `A`

$$n\colon T\colon Set$$

`T` is a type, `n` is a value of type `T`

Coq — sorts Prop and Set

$$d\colon A\colon Prop$$

A is a formula, d is a proof of A

$$n\colon T\colon Set$$

T is a type, n is a value of type T

Coq — sorts Prop and Set

$$d\colon A\colon Prop$$

`A` is a formula, `d` is a proof of `A`

$$n\colon T\colon Set$$

`T` is a type, `n` is a value of type `T`

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \rightarrow B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \;:\; \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — abstraction and application

dependent types abstraction rule:

$$\frac{\Gamma, x{:}A \vdash M : B(x)}{\Gamma \vdash \lambda x{:}A.M \ : \ \forall x{:}A.B(x)}$$

Shorthand: $A \to B$ to $\forall x{:}A.B$, where $x \notin FV(B)$

application rule:

$$\frac{\Gamma \vdash F : \forall x{:}A.B(x) \quad \Gamma \vdash G : A}{\Gamma \vdash F\,G : B[G/x]}$$

Coq — products

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : Prop}{\Gamma \vdash \forall x{:}A.B : Prop} \quad \textbf{Prod-Prop}$$

$$\frac{\Gamma \vdash A : s \qquad s \in Prop, Set \qquad \Gamma, x{:}A \vdash B : Set}{\Gamma \vdash \forall x{:}A.B : Set} \quad \textbf{Prod-Set}$$

$$\frac{\Gamma \vdash A : Type(i) \qquad \Gamma, x{:}A \vdash B : Type(i)}{\Gamma \vdash \forall x{:}A.B : Type(i)} \quad \textbf{Prod-Type}$$

`Prop` is impredicative, `Set`, `Type` are predicative

Coq — products

$$\frac{\Gamma \vdash A : s \qquad s \texttt{ is a sort} \qquad \Gamma, x{:}A \vdash B : \mathit{Prop}}{\Gamma \vdash \forall x{:}A.B : \mathit{Prop}} \quad \textbf{Prod-Prop}$$

$$\frac{\Gamma \vdash A : s \qquad s \in \mathit{Prop}, \mathit{Set} \qquad \Gamma, x{:}A \vdash B : \mathit{Set}}{\Gamma \vdash \forall x{:}A.B : \mathit{Set}} \quad \textbf{Prod-Set}$$

$$\frac{\Gamma \vdash A : \mathit{Type}(i) \qquad \Gamma, x{:}A \vdash B : \mathit{Type}(i)}{\Gamma \vdash \forall x{:}A.B : \mathit{Type}(i)} \quad \textbf{Prod-Type}$$

`Prop` is impredicative, `Set`, `Type` are predicative

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (reduction of `match` applied to constructor term)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (reduction of `match` applied to constructor term)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (reduction of `match` applied to constructor term)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (reduction of `match` applied to constructor term)

Coq — reductions

- beta

  $(\lambda x{:}A.M)N \longrightarrow_\beta M[N/x]$

- eta expansion (if M is of a functional type)

  $M \longrightarrow_\eta \lambda x{:}A.Mx$

- delta

  (definition unfolding)

- zeta

  `(let x:=N in M)` $\longrightarrow_\zeta$ `M[N/x]`

- iota

  (reduction of `match` applied to constructor term)

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

`vector nat 4` $=_{iota}$ `vector nat (2+2)`

includes subtyping on sorts:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ if } s_1 \leq s_2$$

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

`vector nat 4` $=_{iota}$ `vector nat (2+2)`

includes subtyping on sorts:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ if } s_1 \leq s_2$$

Coq — conversion

conversion rule

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A =_{\beta\eta\delta\zeta\iota} A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

`vector nat 4` $=_{iota}$ `vector nat (2+2)`

includes subtyping on sorts:

$$\frac{\Gamma \vdash M : s_1}{\Gamma \vdash M : s_2} \text{ if } s_1 \leq s_2$$

Coq — examples of product types

- functional type nat → nat

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \ : \ \texttt{Set}}\textbf{Prod-Set}$$

- type constructor (ex: List)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x : \texttt{Set} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

- type of a predicate (ex: Even)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

Coq — examples of product types

- functional type $\texttt{nat} \to \texttt{nat}$

$$\frac{\Gamma \vdash \texttt{nat} \;:\; \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{nat} \;:\; \texttt{Set}}{\Gamma \vdash \texttt{nat} \to \texttt{nat} \;:\; \texttt{Set}}\textbf{Prod-Set}$$

- type constructor (ex: $\texttt{List}$)

$$\frac{\Gamma \vdash \texttt{Set} \;:\; \texttt{Type} \qquad \Gamma, x : \texttt{Set} \vdash \texttt{Set} \;:\; \texttt{Type}}{\Gamma \vdash \texttt{Set} \to \texttt{Set} \;:\; \texttt{Type}}\textbf{Prod-Type}$$

- type of a predicate (ex: $\texttt{Even}$)

$$\frac{\Gamma \vdash \texttt{nat} \;:\; \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Prop} \;:\; \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Prop} \;:\; \texttt{Type}}\textbf{Prod-Type}$$

Coq — examples of product types

- functional type nat $\rightarrow$ nat

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x:\texttt{nat} \vdash \texttt{nat} \ : \ \texttt{Set}}{\Gamma \vdash \texttt{nat} \rightarrow \texttt{nat} \ : \ \texttt{Set}}\textbf{Prod-Set}$$

- type constructor (ex: List)

$$\frac{\Gamma \vdash \texttt{Set} \ : \ \texttt{Type} \qquad \Gamma, x:\texttt{Set} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{Set} \rightarrow \texttt{Set} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

- type of a predicate (ex: Even)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x:\texttt{nat} \vdash \texttt{Prop} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \rightarrow \texttt{Prop} \ : \ \texttt{Type}}\textbf{Prod-Type}$$

Coq — examples of product types cont.

- dependent type (ex: ftree)

$$\frac{\Gamma \vdash \texttt{nat} \ : \ \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \ : \ \texttt{Type}}{\Gamma \vdash \texttt{nat} \rightarrow \texttt{Set} \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- polimorphic type $\forall \alpha : \texttt{Set}.\alpha \rightarrow \alpha \ : \ \texttt{Type}$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \rightarrow \alpha : \texttt{Type}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \rightarrow \alpha \ : \ \texttt{Type}} \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \ : \ \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \ : \ \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}.\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \ : \ \texttt{Prop}}$$

Coq — examples of product types cont.

- dependent type (ex: `ftree`)

$$\frac{\Gamma \vdash \mathtt{nat} \ : \ \mathtt{Set} \qquad \Gamma, x : \mathtt{nat} \vdash \mathtt{Set} \ : \ \mathtt{Type}}{\Gamma \vdash \mathtt{nat} \to \mathtt{Set} \ : \ \mathtt{Type}} \textbf{Prod-Type}$$

- polimorphic type $\forall \alpha : \mathtt{Set}.\alpha \to \alpha \ : \ \mathtt{Type}$

$$\frac{\Gamma \vdash \mathtt{Set} : \mathtt{Type} \qquad \Gamma, \alpha : \mathtt{Set} \vdash \alpha \to \alpha : \mathtt{Type}}{\Gamma \vdash \forall \alpha : \mathtt{Set}.\alpha \to \alpha \ : \ \mathtt{Type}} \quad \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \mathtt{Prop} \ : \ \mathtt{Type} \qquad \Gamma, \alpha : \mathtt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \ : \ \mathtt{Prop}}{\Gamma \vdash \forall \alpha : \mathtt{Prop}.\alpha \to (\alpha \to \alpha) \to \alpha \ : \ \mathtt{Prop}}$$

Coq — examples of product types cont.

- dependent type (ex: ftree)

$$\frac{\Gamma \vdash \texttt{nat} \; : \; \texttt{Set} \qquad \Gamma, x : \texttt{nat} \vdash \texttt{Set} \; : \; \texttt{Type}}{\Gamma \vdash \texttt{nat} \to \texttt{Set} \; : \; \texttt{Type}} \textbf{Prod-Type}$$

- polimorphic type $\forall \alpha : \texttt{Set}.\alpha \to \alpha \; : \; \texttt{Type}$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Type}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \; : \; \texttt{Type}} \quad \textbf{Prod-Type}$$

- impredicativity (type of Church numerals)

$$\frac{\Gamma \vdash \texttt{Prop} \; : \; \texttt{Type} \qquad \Gamma, \alpha : \texttt{Prop} \vdash \alpha \to (\alpha \to \alpha) \to \alpha \; : \; \texttt{Prop}}{\Gamma \vdash \forall \alpha : \texttt{Prop}.\alpha \to (\alpha \to \alpha) \to \alpha \; : \; \texttt{Prop}}$$

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Fixpoint plus (n m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Theorem O_plus_n : ∀ n : nat, plus O n = n.
  intro; simpl; reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

$plus\ O\ O = O$

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

`Theorem` n_plus_O : ∀ n : **nat**, plus n O = n.
  `induction` n.

The first subgoal:

plus O O = O

```
reflexivity.
```

holds because of conversion (iota reduction). The second is:

n : **nat**
IHn : plus n O = n
==============================
plus (S n) O = S n

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction

```
Theorem n_plus_O : ∀ n : nat, plus n O = n.
  induction n.
```

The first subgoal:

plus O O = O

```
  reflexivity.
```

holds because of conversion (iota reduction). The second is:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

```
simpl.
rewrite IHn.
reflexivity.
Qed.
```

Natural numbers — induction principle

```
Check nat_ind.
```

$$\text{nat\_ind} : \forall\ P : \textbf{nat} \rightarrow \texttt{Prop},\quad P\ \mathsf{O} \rightarrow (\forall\ n : \textbf{nat},\ P\ n \rightarrow P\ (\mathsf{S}\ n)) \rightarrow \forall\ n : \textbf{nat},\ P\ n$$

```
Theorem n_plus_O' : ∀ n : nat, plus n O = n.
  apply (nat_ind (fun n ⇒ plus n O = n));
[reflexivity | intros n IHn; simpl; rewrite IHn; reflexivity].
Qed.
```

Natural numbers — induction principle

```
Check nat_ind.
```

nat_ind : $\forall\ P : \textbf{nat} \to \texttt{Prop},$
$P\ \mathrm{O} \to (\forall\ n : \textbf{nat},\ P\ n \to P\ (\mathrm{S}\ n)) \to \forall\ n : \textbf{nat},\ P\ n$

Theorem n_plus_O' : $\forall\ n : \textbf{nat}$, plus n O = n.
  apply (nat_ind (fun $n \Rightarrow$ plus n O = n));
[reflexivity | intros n IHn; simpl; rewrite IHn; reflexivity].
Qed.

Natural numbers — induction principle and recursors

`Print` *nat_ind*.

nat_ind =
$\texttt{fun}\ P : \mathbf{nat} \to \texttt{Prop} \Rightarrow \text{nat\_rect}\ P$
$\quad : \forall\ P : \mathbf{nat} \to \texttt{Prop},$
$\qquad P\ \mathrm{O} \to (\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathrm{S}\ n)) \to \forall\ n : \mathbf{nat},\ P\ n$

`Print` *nat_rec*.

nat_rec =
$\texttt{fun}\ P : \mathbf{nat} \to \texttt{Set} \Rightarrow \text{nat\_rect}\ P$
$\quad : \forall\ P : \mathbf{nat} \to \texttt{Set},$
$\qquad P\ \mathrm{O} \to (\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathrm{S}\ n)) \to \forall\ n : \mathbf{nat},\ P\ n$

`Check nat_rect.`

nat_rect
$\quad : \forall\ P : \mathbf{nat} \to \texttt{Type},$
$\qquad P\ \mathrm{O} \to (\forall\ n : \mathbf{nat},\ P\ n \to P\ (\mathrm{S}\ n)) \to \forall\ n : \mathbf{nat},\ P\ n$

Natural numbers — induction principle and recursors

Print *nat_ind*.

nat_ind =
fun P : **nat** → Prop ⇒ nat_rect P
  : ∀ P : **nat** → Prop,
    P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Print *nat_rec*.

nat_rec =
fun P : **nat** → Set ⇒ nat_rect P
  : ∀ P : **nat** → Set,
    P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Check nat_rect.

nat_rect
  : ∀ P : **nat** → Type,
    P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Natural numbers — induction principle and recursors

Print *nat_ind*.

nat_ind =
fun P : **nat** → Prop ⇒ nat_rect P
 : ∀ P : **nat** → Prop,
 P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Print *nat_rec*.

nat_rec =
fun P : **nat** → Set ⇒ nat_rect P
 : ∀ P : **nat** → Set,
 P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Check nat_rect.

nat_rect
 : ∀ P : **nat** → Type,
 P O → (∀ n : **nat**, P n → P (S n)) → ∀ n : **nat**, P n

Primitives fix and match

```
Print nat_rect.

nat_rect =
fun (P : nat → Type) (f : P O) (f0 : ∀ n : nat, P n → P (S n)) ⇒
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O ⇒ f
  | S n' ⇒ f0 n' (F n')
  end
     : ∀ P : nat → Type,
       P O → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

Parametric lists

Inductive **list** $(T : \texttt{Set}) : \texttt{Set} :=$
$|$ Nil : **list** T
$|$ Cons : $T \to$ **list** $T \to$ **list** T.

Check list_ind.

list_ind
: $\forall$ $(T : \texttt{Set})$ $(P :$ **list** $T \to \texttt{Prop})$,
P (Nil T) $\to$
($\forall$ $(t : T)$ $(l :$ **list** $T)$, $P\ l \to P$ (Cons $T\ t\ l$)) $\to$
$\forall\ l :$ **list** T, $P\ l$

Parametric lists

```
Inductive list (T : Set) : Set :=
| Nil : list T
| Cons : T → list T → list T.
```

```
Check list_ind.
```

```
list_ind
    : ∀ (T : Set) (P : list T → Prop),
      P (Nil T) →
      (∀ (t : T) (l : list T), P l → P (Cons T t l)) →
      ∀ l : list T, P l
```

Parametric lists cont.

```
Arguments Nil [T].
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil => O
    | Cons _ ls' => S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil => ls2
    | Cons x ls1' => Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil [T].
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil => O
    | Cons _ ls' => S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil => ls2
    | Cons x ls1' => Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil [T].
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil ⇒ O
    | Cons _ ls' ⇒ S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil ⇒ ls2
    | Cons x ls1' ⇒ Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Parametric lists cont.

```
Arguments Nil [T].
Arguments Cons [T].

Fixpoint length {T} (ls : list T) : nat :=
  match ls with
    | Nil => O
    | Cons _ ls' => S (length ls')
  end.

Fixpoint app {T} (ls1 ls2 : list T) : list T :=
  match ls1 with
    | Nil => ls2
    | Cons x ls1' => Cons x (app ls1' ls2)
  end.

Theorem length_app : ∀ T (ls1 ls2 : list T), length (app ls1 ls2)
  = plus (length ls1) (length ls2).
  induction ls1....
Qed.
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.
```

```
Check lista_ind.
```

```
lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.
```

```
Check lista_ind.
```

```
lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Nonparametric lists

```
Inductive lista : Set -> Type :=
| Nila : forall (A:Set), lista A
| Consa : forall (A:Set), A -> lista A -> lista A.
```

```
Check lista_ind.
```

```
lista_ind:
    forall P : (forall A : Set, lista A -> Prop),
    (forall A : Set, P A (Nila A)) ->
    (forall (A : Set) (a : A) (l : lista A),
          P A l -> P A (Consa A a l)) ->
    forall (P0 : Set) (l : lista P0), P P0 l
```

Trees

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.
```

```
Check nat_btree_ind.

  nat_btree_ind
      : ∀ P : nat_btree → Prop,
        P NLeaf →
        (∀ n : nat_btree, P n → ∀ (n0 : nat) (n1 : nat_btree),
         P n1 → P (NNode n n0 n1)) →
        ∀ n : nat_btree, P n
```

Trees

```
Inductive nat_btree : Set :=
| NLeaf : nat_btree
| NNode : nat_btree → nat → nat_btree → nat_btree.
```

```
Check nat_btree_ind.
```

```
nat_btree_ind
    : ∀ P : nat_btree → Prop,
      P NLeaf →
      (∀ n : nat_btree, P n → ∀ (n0 : nat) (n1 : nat_btree),
       P n1 → P (NNode n n0 n1)) →
      ∀ n : nat_btree, P n
```

Mutually recursive types: odd_list and even_list

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
```

```
Check even_list_ind.

  even_list_ind
      : ∀ P : even_list → Prop,
        P ENil →
        (∀ (n : nat) (o : odd_list), P (ECons n o)) →
        ∀ e : even_list, P e
```

Mutually recursive types: odd_list and even_list

```
Inductive even_list : Set :=
| ENil : even_list
| ECons : nat → odd_list → even_list

with odd_list : Set :=
| OCons : nat → even_list → odd_list.
```

```
Check even_list_ind.
```

```
even_list_ind
    : ∀ P : even_list → Prop,
      P ENil →
      (∀ (n : nat) (o : odd_list), P (ECons n o)) →
      ∀ e : even_list, P e
```

Scheme — generation of induction principles

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.
```

```
Check even_list_mut.
```

even_list_mut
: $\forall$ (P : **even_list** $\rightarrow$ Prop) ($P0$: **odd_list** $\rightarrow$ Prop),
P ENil $\rightarrow$
($\forall$ (n : **nat**) (o : **odd_list**), $P0$ o $\rightarrow$ P (ECons n o)) $\rightarrow$
($\forall$ (n : **nat**) (e : **even_list**), P e $\rightarrow$ $P0$ (OCons n e)) $\rightarrow$
$\forall$ e : **even_list**, P e

Scheme — generation of induction principles

```
Scheme even_list_mut := Induction for even_list Sort Prop
with odd_list_mut := Induction for odd_list Sort Prop.
```

```
Check even_list_mut.
```

even_list_mut

$: \forall\ (P : \textbf{even\_list} \to \texttt{Prop})\ (P0 : \textbf{odd\_list} \to \texttt{Prop}),$

$P\ \text{ENil} \to$

$(\forall\ (n : \textbf{nat})\ (o : \textbf{odd\_list}),\ P0\ o \to P\ (\text{ECons}\ n\ o)) \to$

$(\forall\ (n : \textbf{nat})\ (e : \textbf{even\_list}),\ P\ e \to P0\ (\text{OCons}\ n\ e)) \to$

$\forall\ e : \textbf{even\_list},\ P\ e$

Reflexive type: formula

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

```
Check formula_ind.

  formula_ind
      : ∀ P : formula → Prop,
        (∀ n n0 : nat, P (Eq n n0)) →
        (∀ f0 : formula,
         P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
        (∀ f1 : nat → formula,
         (∀ n : nat, P (f1 n)) → P (Forall f1)) →
        ∀ f2 : formula, P f2
```

Reflexive type: formula

```
Inductive formula : Set :=
| Eq : nat → nat → formula
| And : formula → formula → formula
| Forall : (nat → formula) → formula.
```

```
Check formula_ind.

  formula_ind
      : ∀ P : formula → Prop,
        (∀ n n0 : nat, P (Eq n n0)) →
        (∀ f0 : formula,
         P f0 → ∀ f1 : formula, P f1 → P (And f0 f1)) →
        (∀ f1 : nat → formula,
         (∀ n : nat, P (f1 n)) → P (Forall f1)) →
        ∀ f2 : formula, P f2
```

Restrictions: positivity condition

Inductive **term** : Set :=
| App : **term** → **term** → **term**
| Abs : (**term** → **term**) → **term**.

```
Error: Non strictly positive occurrence of "term" in "(term
-> term) -> term"
```

Restrictions: positivity condition

Inductive **term** : Set :=
| App : **term** → **term** → **term**
| Abs : (**term** → **term**) → **term**.

```
Error: Non strictly positive occurrence of "term" in "(term
-> term) -> term"
```

Restrictions: only small inductive types in Set

Correct (definition of $\exists\, \phi \;\; P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.
```

```
Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene:

$$\texttt{exType} : (P : \mathrm{Type}_i \to \mathrm{Prop}) \to \mathrm{Type}_j$$

$$\texttt{exT\_intro} : \forall\, X : \mathrm{Type}_k,\; P\, X -> \texttt{exType}\, P$$

where $k < j$ and $k < i$ (universe constraints)

Restrictions: only small inductive types in Set

Correct (definition of $\exists \phi \ \ P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.
```

```
Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene:

$$\text{exType} : (P : \text{Type}_i \rightarrow \text{Prop}) \rightarrow \text{Type}_j$$

$$\text{exT\_intro} : \forall\, X : \text{Type}_k,\ P\, X -> \text{exType}\, P$$

where $k < j$ and $k < i$ (universe constraints)

Restrictions: only small inductive types in Set

Correct (definition of $\exists\, \phi \;\; P(\phi)$):

```
Inductive exProp (P:Prop->Prop) : Prop
:= exP_intro : forall X:Prop, P X -> exProp P.
```

Incorrect:

```
Inductive exSet (P:Set->Prop) : Set
:= exS_intro : forall X:Set, P X -> exSet P.


Error: Large non-propositional inductive types must be in Type.
```

Correct:

```
Inductive exType (P:Type->Prop) : Type
:= exT_intro : forall X:Type, P X -> exType P.
```

behind the scene:

$$\texttt{exType} : (P : \texttt{Type}_i \to \texttt{Prop}) \to \texttt{Type}_j$$

$$\texttt{exT\_intro} : \forall\, X : \texttt{Type}_k,\; P\, X - > \texttt{exType}\, P$$

where $k < j$ and $k < i$ (universe constraints)

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \text{Type})\ (x : A) : A \rightarrow \text{Prop} := \text{eq\_refl} : x = x$

```
Check @eq_refl.
@eq_refl
    : forall (A : Type) (x : A), x = x
```

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

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \mathsf{eq\_refl} : x = x$

```
Check @eq_refl.
@eq_refl
    : forall (A : Type) (x : A), x = x
```

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

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \texttt{Type})\ (x : A) : A \to \texttt{Prop} := \mathsf{eq\_refl} : x = x$

```
Check @eq_refl.
@eq_refl
    : forall (A : Type) (x : A), x = x
```

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

Definitional equality

```
Print "=".
```

Inductive **eq** $(A : \text{Type})\ (x : A) : A \to$ Prop := eq_refl : $x = x$

```
Check @eq_refl.
@eq_refl
    : forall (A : Type) (x : A), x = x
```

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

reflexivity and rewrite

`reflexivity` ≡ `apply eq_refl`

`rewrite H` ≡ `apply eq_ind`

(where `H: a=b`)

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

reflexivity and rewrite

reflexivity $\equiv$ apply eq_refl

rewrite H $\equiv$ apply eq_ind

(where H: a=b)

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