# Dowodzenie twierdzeń i programowanie z typami zależnymi w Coqu

Daria Walukiewicz-Chrząszcz

22 marca 2016

# Motywacja: poprawność programów

- Dowodzenie własności programów

## Przykład sprzed tygodnia

*ciężko testować np funkcję insert*

```
...
insert :: a -> [a] -> [a]
...
```

*czy dla dowolnej listy xs i elementu z spełnia własność:*

```
...
sorted (xs) -> sorted (insert z xs)
...
```

# Motywacja: poprawność programów

- Dowodzenie własności programów

**Przykład sprzed tygodnia**

*ciężko testować np funkcję insert*

```
...
insert :: a -> [a] -> [a]
...
```

*czy dla dowolnej listy xs i elementu z spełnia własność:*

```
...
sorted (xs) -> sorted (insert z xs)
...
```

# Motywacja: poprawność programów

- Dowodzenie własności programów

## Przykład sprzed tygodnia

*ciężko testować np funkcję insert*

```
...
insert :: a -> [a] -> [a]
...
```

*czy dla dowolnej listy xs i elementu z spełnia własność:*

```
...
sorted (xs) -> sorted (insert z xs)
...
```

# Motywacja: typy jako specyfikacje programów

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

# Motywacja: typy jako specyfikacje programów

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

# Motywacja: typy jako specyfikacje programów

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

# Motywacja: typy jako specyfikacje programów

- Lepsze specyfikowanie funkcji

```
Inductive ftree : nat → Set :=
| Leaf : ftree 0
| Node : ∀ n: nat, Z → ftree n → ftree n → ftree (S n).
```

```
Fixpoint root (n : nat)(t : ftree(S n)) : Z :=
  match t with
  | Node n k l r ⇒ k
  end.
```

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ

$$\forall n : nat. M_{n,n}$$

- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
  typ `ftree n` zależy od wartości `n` (typy zależne)
  typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \quad \equiv \quad \texttt{nat} \to \texttt{bool}$

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ

$$\forall n : nat. M_{n,n}$$

- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
  typ `ftree n` zależy od wartości `n` (typy zależne)
  typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat → bool`

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. ftree\ n$$

$$\Pi n : nat. ftree\ n$$

`forall n : nat, ftree n`

Uwaga: `forall n : nat, bool` $\equiv$ `nat` $\to$ `bool`

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. \mathit{ftree}\, n$$

$$\Pi n : nat. \mathit{ftree}\, n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \equiv \texttt{nat} \to \texttt{bool}$

# Typy zależne - wstęp

Typ zależny - typ funkcji, której przeciwdziedzina zależy od argumentu

- przyporządkowanie $n \mapsto I_{n,n}$ macierzy jednostkowej o wymiarach $n$ na $n$ ma typ
$$\forall n : nat. M_{n,n}$$
- w monoidzie $(A, e, *)$ operacja $*$ jest funkcją z $A \to A \to A$, taka że $e$ jest jej elementem neutralnym
- typ `List A` zależy od typu `A` (polimorfizm)
typ `ftree n` zależy od wartości `n` (typy zależne)
typ `vector A n` zależy od typu `A` i wartości `n`

Notacja:

$$\forall n : nat. \mathit{ftree}\, n$$

$$\Pi n : nat. \mathit{ftree}\, n$$

$$\texttt{forall n : nat, ftree n}$$

Uwaga: $\texttt{forall n : nat, bool} \quad \equiv \quad \texttt{nat} \to \texttt{bool}$

# Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

# Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

# Typy zależne - obliczenia w typach

`ftree (2+2)` $\equiv$ `ftree (4)`

reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

# Typy zależne - obliczenia w typach

$$\texttt{ftree (2+2)} \equiv \texttt{ftree (4)}$$

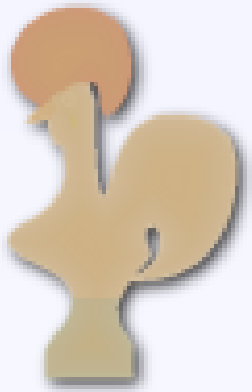
reguła *konwersji* utożsamia typy jak powyżej

```
data Parity : nat -> Type where
| Even : forall n:nat, Parity (n + n)
| Odd : forall n:nat, Parity (S (n + n))
```

```
parity : (n:nat) -> Parity n
parity O = Even O
parity (S O) = Odd O
parity (S (S k)) = match (parity k) with
| Even j => Even (S j)
| Odd j => Odd (S j)
```

Wniosek: typy zależne zazwyczaj potrzebują lematów, dowodów....

# Coq — system wspomagania dowodzenia

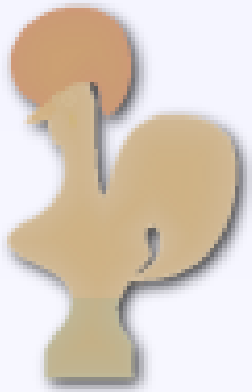


http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych" języków: Ocaml, Haskell, Scheme...

# Coq — system wspomagania dowodzenia

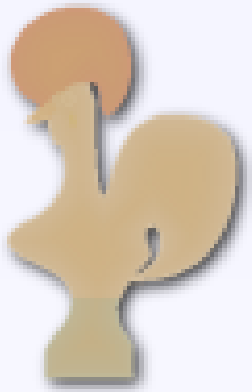


http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych” języków: Ocaml, Haskell, Scheme...

# Coq — system wspomagania dowodzenia

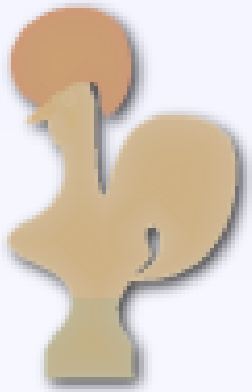


http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych” języków: Ocaml, Haskell, Scheme...

# Coq — system wspomagania dowodzenia

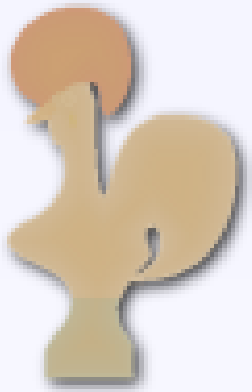


http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych” języków: Ocaml, Haskell, Scheme...

# Coq — system wspomagania dowodzenia

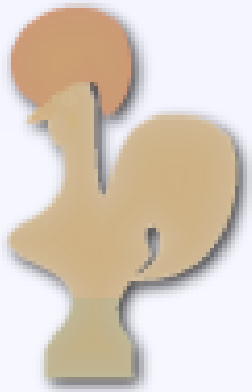


http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych” języków: Ocaml, Haskell, Scheme...

# Coq — system wspomagania dowodzenia



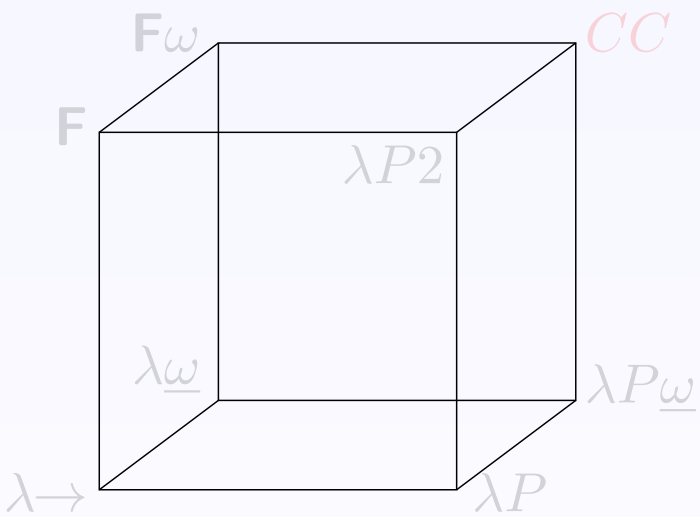
http://coq.inria.fr/

```
(***********)
(*  v      *)
(* <O___,, *)
(*   \VV/  *)
(*    //   *)
(*         *)
(***********)
```

- programowanie funkcyjne z typami zależnymi
- specyfikowanie własności programów
- półautomatyczne dowodzenie własności programów (i nie tylko)
- ekstrakcja poprawnych programów do „prawdziwych” języków: Ocaml, Haskell, Scheme...

# Coq — podstawy teoretyczne

Coq — rachunek konstrukcji indukcyjnych (CIC)



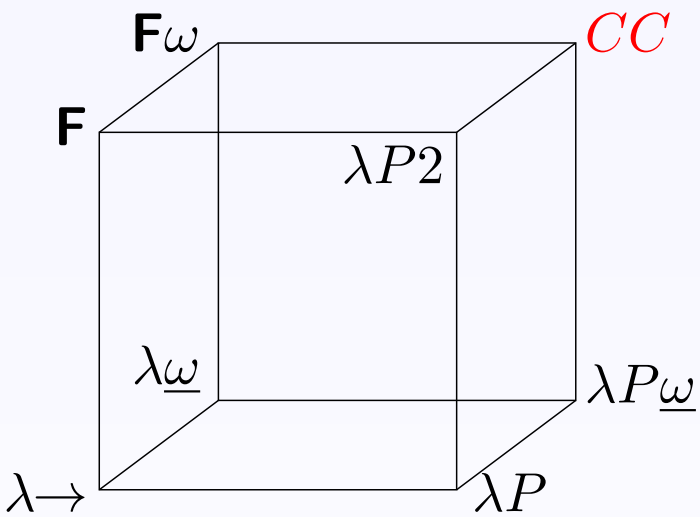

↑ polimorfizm

↗ konstruktory typów

→ typy zależne

# Coq — podstawy teoretyczne

Coq — rachunek konstrukcji indukcyjnych (CIC)



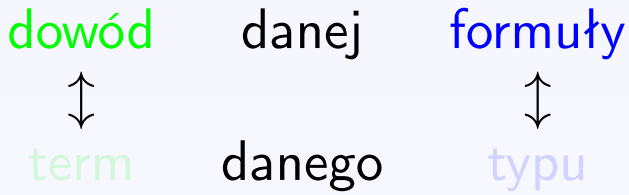

↑ polimorfizm

↗ konstruktory typów

→ typy zależne

# Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

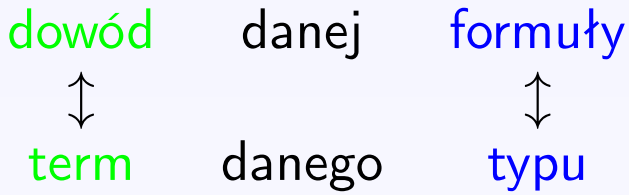



$$\lambda x^{A \to B \to C}\, \lambda y^{A \to B}\, \lambda z^{A}\, xz(yz) : (A \to B \to C) \to (A \to B) \to (A \to C)$$

Poprawność Coqa: poprawność algorytmu sprawdzania typów

# Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

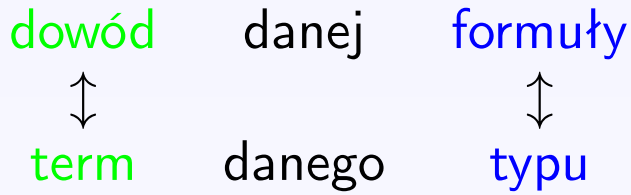



$$\lambda x^{A \to B \to C}\, \lambda y^{A \to B}\, \lambda z^{A}\, xz(yz) : (A \to B \to C) \to (A \to B) \to (A \to C)$$

Poprawność Coqa: poprawność algorytmu sprawdzania typów

# Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

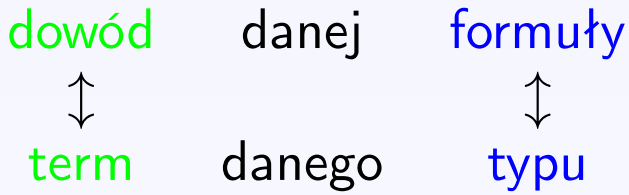



$$\lambda x^{A \to B \to C}\, \lambda y^{A \to B}\, \lambda z^{A}\; xz(yz) \;:\; (A \to B \to C) \to (A \to B) \to (A \to C)$$

Poprawność Coqa: poprawność algorytmu sprawdzania typów

# Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda




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

Poprawność Coqa: poprawność algorytmu sprawdzania typów

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Architektura Coq

Zasada De Bruijna ("małe" jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły
- reszta (≈230KLOC), odpowiedzialna za:
  - interfejs użytkownika
  - zarządzanie plikami
  - sekcje
  - zarządzanie nazwami
  - tryb dowodowy (plus taktyki i język taktyk)
  - notacje
  - argumenty domyślne (rekonstrukcja typów)
  - type classes
  - koercje
  - automatyczne generowanie zasad indukcji
  - ...

# Coq — historia

1984 CoC - rachunek konstrukcji - G. Huet, T. Coquand

1989 pierwsze wydanie publiczne (wersja 4.10)

1991 Coq - rachunek konstrukcji indukcyjnych - C. Paulin (wersja 5.6)

⋮

2000 wersja 7.0 z nową (bezpieczną) architekturą

2003 wersja 7.4 z modułami

2004 wersja 8.0 z nową składnią

2009 wersja 8.2 z "type classes"

2012 wersja 8.4 z eta-redukcją, strukturalną składnią dowodów

2016 wersja 8.5

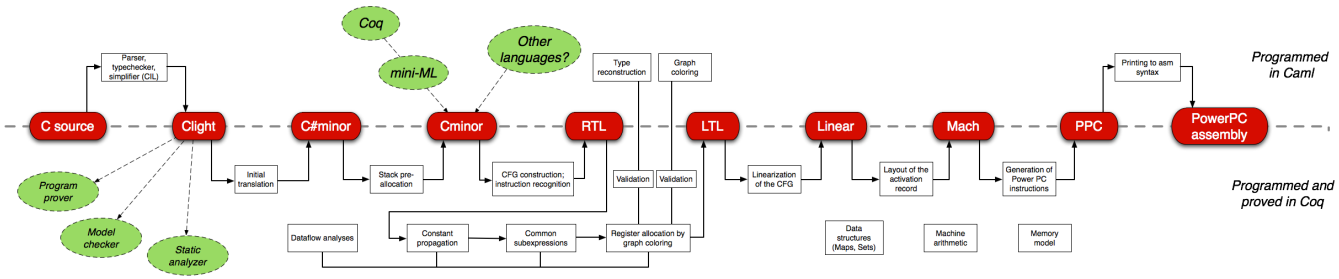
# Coq — znaczące formalizacje

- Zasadnicze Twierdzenie Algebry, Nijmegen 2000
- Formalizacja platformy JavaCard, 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)

- Twierdzenie o 4 barwach, Cambridge 2004
- CompCert zweryfikowany kompilator Clight, 2008-teraz

  *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 — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — język programowania

- predykatywne sorty `Set` i `Type`
- abstrakcja i aplikacja
- typy indukcyjne,
- dopasowywanie wzorca (dependent pattern-matching)
- schematy rekursji i indukcji
- polimorfizm
- typy zależne
- moduły i funktory
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja ("wbudowane")
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja (“wbudowane”)
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja (“wbudowane”)
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja (“wbudowane”)
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja (“wbudowane”)
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Coq — logika

- intuicjonistyczna logika wyższego rzędu
- impredykatywny sort `Prop`
- kwantyfikator ogólny i implikacja ("wbudowane")
- spójniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- predykaty indukcyjne (w tym równość)
- ...

# Maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc taktyki: auto, intuition, etc.
- procedury decyzyjne: omega, ring, field, tauto, etc.
- język taktyk (`Ltac mytactic:=...`)

# Maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc taktyki: auto, intuition, etc.
- procedury decyzyjne: omega, ring, field, tauto, etc.
- język taktyk (`Ltac mytactic:=...`)

# Maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc taktyki: auto, intuition, etc.
- procedury decyzyjne: omega, ring, field, tauto, etc.
- język taktyk (`Ltac mytactic:=...`)

# Maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc taktyki: auto, intuition, etc.
- procedury decyzyjne: omega, ring, field, tauto, etc.
- język taktyk (`Ltac mytactic:=...`)

# Maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc taktyki: auto, intuition, etc.
- procedury decyzyjne: omega, ring, field, tauto, etc.
- język taktyk (`Ltac mytactic:=...`)

# Ekstrakcja programów

- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, w której prawa takie jak "wyłączonego środka", "przez sprzeczność" nie są prawdziwe,
- języki docelowe: O'Caml, Haskell, Scheme

# Ekstrakcja programów

- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, w której prawa takie jak "wyłączonego środka", "przez sprzeczność" nie są prawdziwe,
- języki docelowe: O'Caml, Haskell, Scheme

# Ekstrakcja programów

- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, w której prawa takie jak "wyłączonego środka", "przez sprzeczność" nie są prawdziwe,
- języki docelowe: O'Caml, Haskell, Scheme

# Ekstrakcja programów

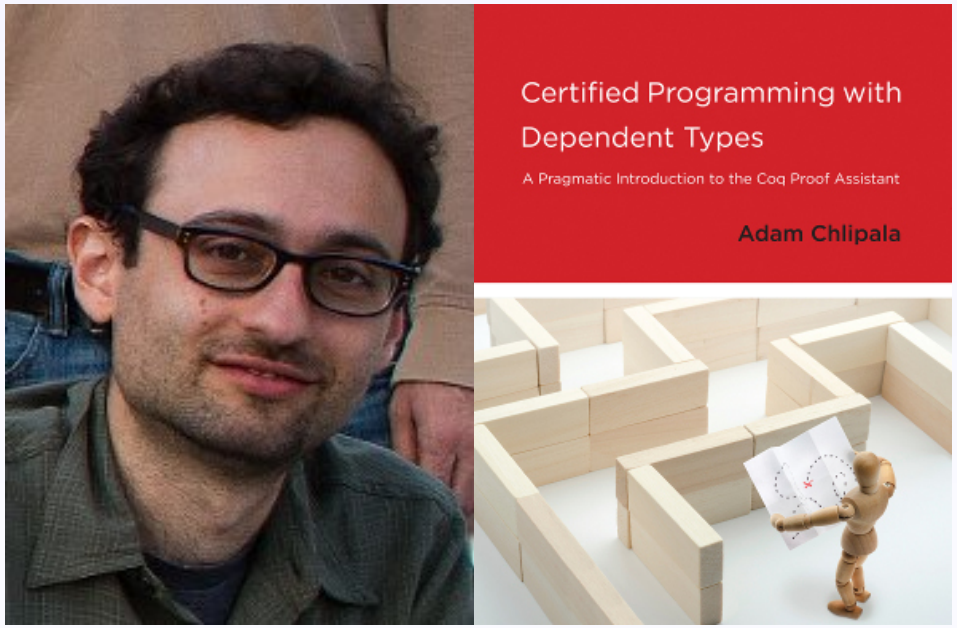
- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, w której prawa takie jak "wyłączonego środka", "przez sprzeczność" nie są prawdziwe,
- języki docelowe: O'Caml, Haskell, Scheme

# Ekstrakcja programów

- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, w której prawa takie jak "wyłączonego środka", "przez sprzeczność" nie są prawdziwe,
- języki docelowe: O'Caml, Haskell, Scheme

# Materiały do wykładu

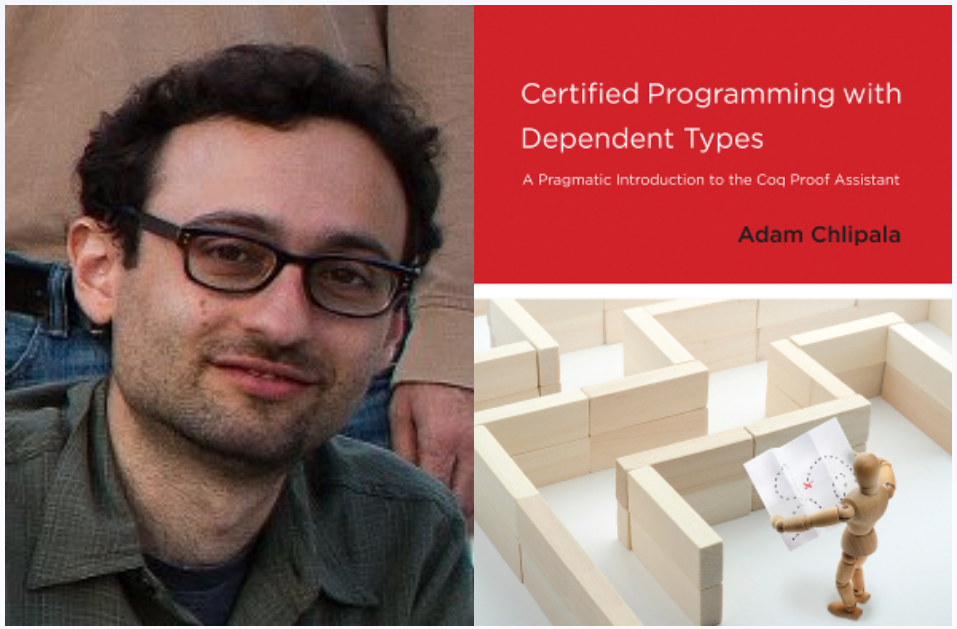
- https://coq.inria.fr/
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)




wydawnictwo: MIT Press 2013
dostępne: http://adam.chlipala.net/cpdt/

# Materiały do wykładu

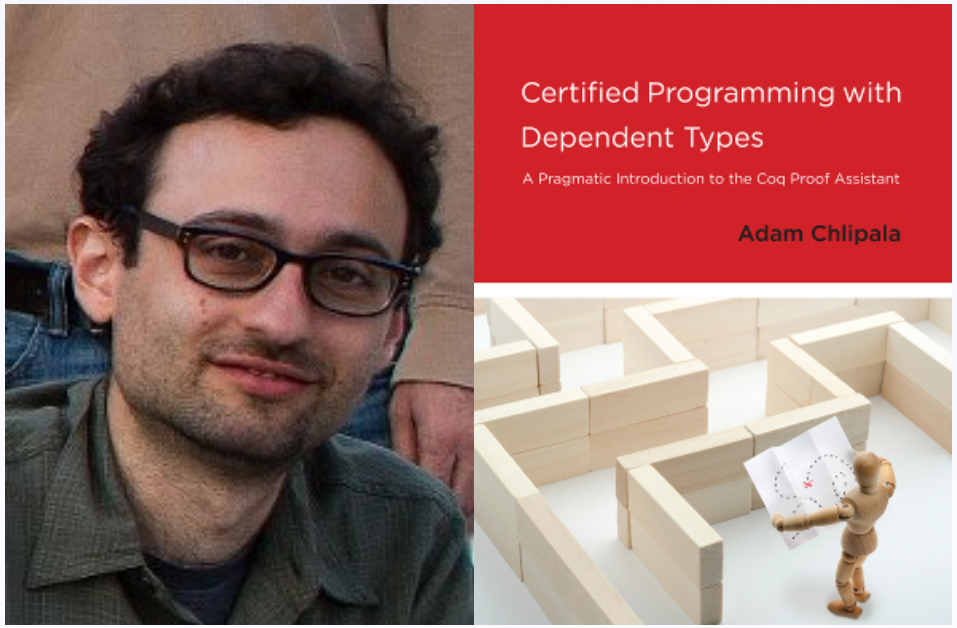
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)




wydawnictwo: MIT Press 2013
dostępne: `http://adam.chlipala.net/cpdt/`

# Materiały do wykładu

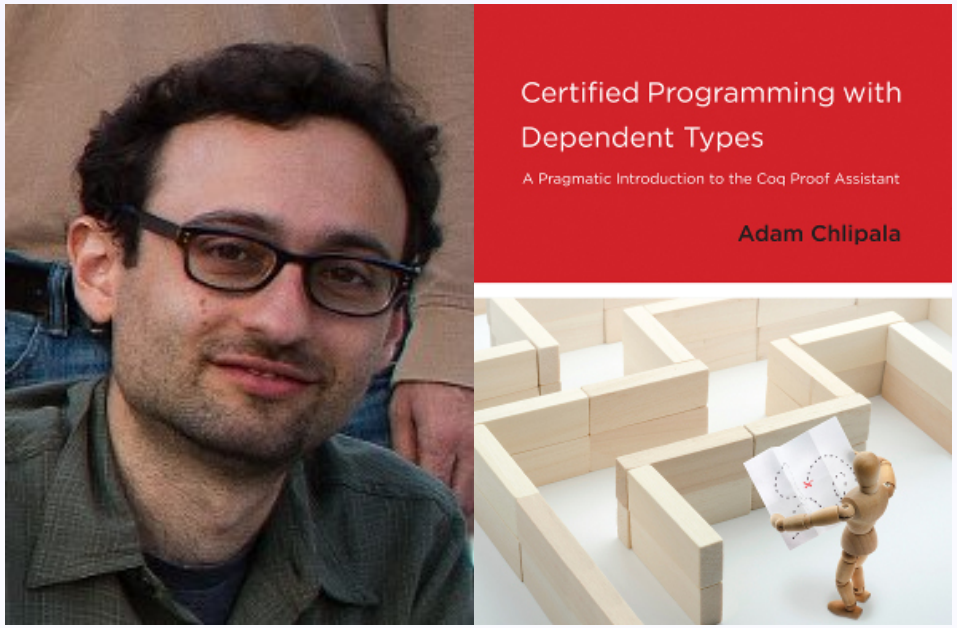
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)




wydawnictwo: MIT Press 2013
dostępne: `http://adam.chlipala.net/cpdt/`

# Materiały do wykładu

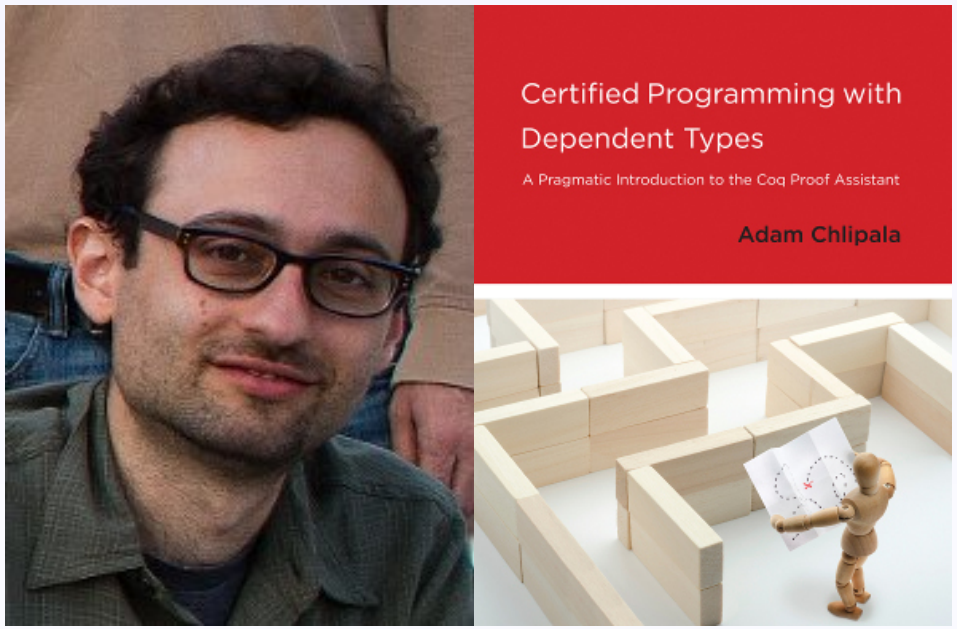
- `https://coq.inria.fr/`
- Coq Art, Yves Bertot, Pierre Castéran
- Certified Programming with Dependent Types, Adam Chlipala (MIT)




wydawnictwo: MIT Press 2013
dostępne: `http://adam.chlipala.net/cpdt/`

# Materiały do wykładu

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




wydawnictwo: MIT Press 2013
dostępne: `http://adam.chlipala.net/cpdt/`

# Typowanie

$$\texttt{środowisko} \vdash \texttt{term} : \texttt{typ}$$

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, $\texttt{Set} : \texttt{Type}_1$

ale nie wszystko jest typem np. `fun n:nat => n`

# Typowanie

$$\texttt{środowisko} \vdash \texttt{term} : \texttt{typ}$$

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, $\texttt{Set} : \texttt{Type}_1$

ale nie wszystko jest typem np. `fun n:nat => n`

# Typowanie

$$\texttt{środowisko} \vdash \texttt{term} : \texttt{typ}$$

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, $\texttt{Set}:\texttt{Type}_1$

ale nie wszystko jest typem np. `fun n:nat => n`

# Typowanie

$$\texttt{środowisko} \vdash \texttt{term} : \texttt{typ}$$

środowisko: globalne i lokalne deklaracje i definicje

typy też są termami i mają typ, np. `nat:Set`, $\texttt{Set}:\texttt{Type}_1$

ale nie wszystko jest typem np. `fun n:nat => n`

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: abstrakcja i aplikacja

reguła abstrakcji dla typów prostych:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M \ : \ A \to B}$$

reguła abstrakcji dla typów zależnych:

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

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

$\lambda n{:}\mathrm{nat}.M \ : \ \forall n{:}\mathrm{nat}.\mathrm{ftree}(n)$ to w Coqu:

```
fun n:nat => M : forall n:nat, ftree n
```

reguła aplikacji:

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

# Coq — reguły typowania: sorty

- Sorty w Coqu:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type_1 : Type_2 : \ldots$$

- Kumulatywność (lub podtypowanie):

$$Prop \leq Set \leq Type_1 \leq Type_2 \leq \ldots$$

# Coq — reguły typowania: sorty

- Sorty w Coqu:

$$\begin{matrix} Prop \\ Set \end{matrix} : Type_1 : Type_2 : \ldots$$

- Kumulatywność (lub podtypowanie):

$$Prop \leq Set \leq Type_1 \leq Type_2 \leq \ldots$$

# Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli $s_1$ i $s_2$ spełniają . . .

- $s_1 \leq s_2$, or
- $s_2 = Prop$

reguła podtypowania

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

# Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli $s_1$ i $s_2$ spełniają . . .

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

reguła podtypowania

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

# Coq — produkty

reguła typowania dla typu produktowego

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.B : s_2}$$ jeśli $s_1$ i $s_2$ spełniają ...

- $s_1 \leq s_2$, or
- $s_2 = \mathit{Prop}$

reguła podtypowania

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

# Coq — produkty

- typ funkcyjny $(\lambda n : nat.n + 1)$

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

- polimorfizm $(\lambda \alpha : \texttt{Set}.\lambda x : \alpha.x)$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Set}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}}$$

- typ predykatu $(\texttt{Even})$

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

# Coq — produkty

- typ funkcyjny $(\lambda n : nat.n + 1)$

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

- polimorfizm $(\lambda \alpha : \texttt{Set}.\lambda x : \alpha.x)$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Set}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}}$$

- typ predykatu (Even)

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

# Coq — produkty

- typ funkcyjny $(\lambda n : nat.n + 1)$

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

- polimorfizm $(\lambda \alpha : \texttt{Set}.\lambda x : \alpha.x)$

$$\frac{\Gamma \vdash \texttt{Set} : \texttt{Type} \qquad \Gamma, \alpha : \texttt{Set} \vdash \alpha \to \alpha : \texttt{Set}}{\Gamma \vdash \forall \alpha : \texttt{Set}.\alpha \to \alpha \ : \ \texttt{Type}}$$

- typ predykatu $(\texttt{Even})$

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

# Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

- impredykatywność (numerały Churcha)

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

# Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

- impredykatywność (numerały Churcha)

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

# Coq — produkty (cd.)

- konstructor typu (`List`)

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

- typ zależny (`ftree`)

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

- impredykatywność (numerały Churcha)

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

# Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — redukcje

- beta

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

- eta

  $\lambda x{:}A.Mx \longrightarrow_\eta M \quad$ if $x \notin FV(M)$

- delta

  (rozwijanie definicji)

- zeta

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

- iota

  (związane z typami indukcyjnymi)

# Coq — konwersja

reguła konwersji

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

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

# Coq — konwersja

reguła konwersji

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

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

# Coq — konwersja

reguła konwersji

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

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

# Coq — typy indukcyjne — wprowadzenie

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Typ indukcyjny:

```
bool : Set
```

- Konstruktory:

```
true : bool
false : bool
```

- Analiza przypadków:

```
match n with
| true => ...
| false => ...
end
```

# Coq — typy indukcyjne — wprowadzenie

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Typ indukcyjny:

```
bool : Set
```

- Konstruktory:

```
true : bool
false : bool
```

- Analiza przypadków:

```
match n with
| true => ...
| false => ...
end
```

# Coq — typy indukcyjne — wprowadzenie

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Typ indukcyjny:

```
bool : Set
```

- Konstruktory:

```
true : bool
false : bool
```

- Analiza przypadków:

```
match n with
| true => ...
| false => ...
end
```

# Coq — typy indukcyjne — wprowadzenie

```
Inductive bool : Set :=
  true : bool
| false : bool.
```

- Typ indukcyjny:

```
bool : Set
```

- Konstruktory:

```
true : bool
false : bool
```

- Analiza przypadków:

```
match n with
| true => ...
| false => ...
end
```

# Coq — typy indukcyjne z parametrami

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Typ indukcyjny z parametrem:

```
list : Set -> Set
```

- Konstruktory:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Definicja rekurencyjna po budowie listy

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

# Coq — typy indukcyjne z parametrami

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Typ indukcyjny z parametrem:

```
list : Set -> Set
```

- Konstruktory:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Definicja rekurencyjna po budowie listy

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

# Coq — typy indukcyjne z parametrami

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Typ indukcyjny z parametrem:

```
list : Set -> Set
```

- Konstruktory:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Definicja rekurencyjna po budowie listy

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

# Coq — typy indukcyjne z parametrami

```
Inductive list (A:Set): Set :=
| nil : list A
| cons : A -> list A -> list A.
```

- Typ indukcyjny z parametrem:

```
list : Set -> Set
```

- Konstruktory:

```
nil : forall A:Set, list A
cons : forall A:Set, A -> list A -> list A
```

- Definicja rekurencyjna po budowie listy

```
Fixpoint length (A:Set)(l:list A) : nat :=
match l with
| nil => 0
| cons hd tl => S (length A tl)
end.
```

# Coq — predykaty indukcyjne

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Predykat indukcyjny:

```
sorted : list Z -> Prop
```

- Konstruktory:

```
sorted0
sorted1
sorted2
```

# Coq — predykaty indukcyjne

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Predykat indukcyjny:

```
sorted : list Z -> Prop
```

- Konstruktory:

```
sorted0
sorted1
sorted2
```

# Coq — predykaty indukcyjne

```
Inductive sorted : list Z -> Prop :=
| sorted0 : sorted nil
| sorted1: forall z:Z, sorted (z :: nil)
| sorted2: forall (z1 z2 : Z) (l : list Z),
    z1 <= z2 -> sorted (z2 :: l) -> sorted (z1 :: z2 :: l).
```

- Predykat indukcyjny:

```
sorted : list Z -> Prop
```

- Konstruktory:

```
sorted0
sorted1
sorted2
```