COQ

Daria Walukiewicz-Chrząszcz
Weryfikacja Wspomagana Komputerowo

30 maja 2014

Co to jest Coq

Coq to “System wspomagania dowodzenia” (Proof assistant)

Co to jest Coq

Coq to “System wspomagania dowodzenia” (Proof assistant)

`http://coq.inria.fr/`

Co to jest Coq

Coq to “System wspomagania dowodzenia” (Proof assistant)

`http://coq.inria.fr/`

- użytkownik musi sam wymyślić metodę dowodu,

Co to jest Coq

Coq to “System wspomagania dowodzenia” (Proof assistant)

`http://coq.inria.fr/`

- użytkownik musi sam wymyślić metodę dowodu,
- system weryfikuje, czy krok zaproponowany przez użytkownika jest dopuszczalny,

Co to jest Coq

Coq to "System wspomagania dowodzenia" (Proof assistant)

`http://coq.inria.fr/`

- użytkownik musi sam wymyślić metodę dowodu,
- system weryfikuje, czy krok zaproponowany przez użytkownika jest dopuszczalny,
- oraz rozwiązuje podproblemy, dla których zaimplementowano w nim algorytmy automatycznego dowodzenia,

Co to jest Coq

Coq to "System wspomagania dowodzenia" (Proof assistant)

`http://coq.inria.fr/`

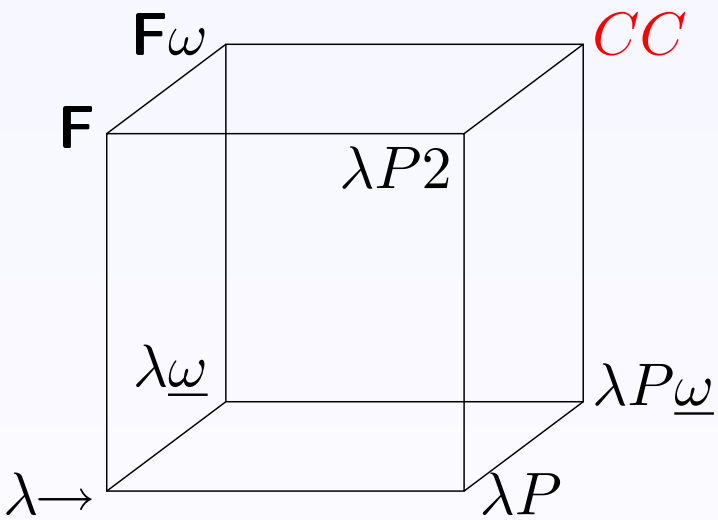
- użytkownik musi sam wymyślić metodę dowodu,
- system weryfikuje, czy krok zaproponowany przez użytkownika jest dopuszczalny,
- oraz rozwiązuje podproblemy, dla których zaimplementowano w nim algorytmy automatycznego dowodzenia,
- działanie systemu opiera się na stosowaniu reguł wnioskowania dla logiki, na której jest zbudowany.

Coq — formalizm

Coq — intuicjonistyczna logika wyższego rzędu
(rachunek konstrukcji (CC) + definicje indukcyjne)

Coq — formalizm

Coq — intuicjonistyczna logika wyższego rzędu
(rachunek konstrukcji (CC) + definicje indukcyjne)



↑ polimorfizm

↗ konstruktory typów

→ typy zależne

Coq — formalizm

reguła abstrakcji dla typów prostych:

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

Coq — formalizm

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:

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

Coq — formalizm

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 $x \notin FV(B)$

Coq — formalizm

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 $x \notin FV(B)$

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

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

Coq — formalizm

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 $x \notin FV(B)$

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

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

reguła aplikacji:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash F\,G : B}$$

Coq — formalizm

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 $x \notin FV(B)$

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

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

reguła aplikacji:

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

Izomorfizm Curry'ego-Howarda

dowód danej formuły

Izomorfizm Curry'ego-Howarda

Izomorfizm Curry'ego-Howarda

$$(A \to B \to C) \to (A \to B) \to (A \to C)$$

Izomorfizm Curry'ego-Howarda

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

Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

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

Pytanie: czy mam poprawny dowód danego lematu odpowiada pytaniu czy odpowiedni term jest danego typu

Poprawność Coqa: poprawność alorytmu sprawdzania typów

Definicje indukcyjne w Coqu

- proste typy danych
  (np. liczby naturalne unarne)

Definicje indukcyjne w Coqu

- proste typy danych
  (np. liczby naturalne unarne)
- typy polimorficzne
  (np. listy)

Definicje indukcyjne w Coqu

- proste typy danych
  (np. liczby naturalne unarne)
- typy polimorficzne
  (np. listy)
- typy zależne
  (np. wektory danej długośc lub drzewa danego rozmiaru)

Definicje indukcyjne w Coqu

- proste typy danych
  (np. liczby naturalne unarne)
- typy polimorficzne
  (np. listy)
- typy zależne
  (np. wektory danej długośc lub drzewa danego rozmiaru)
- formuły
  (prawie cała logika Coqa)

Definicje indukcyjne w Coqu

- proste typy danych
  (np. liczby naturalne unarne)
- typy polimorficzne
  (np. listy)
- typy zależne
  (np. wektory danej długośc lub drzewa danego rozmiaru)
- formuły
  (prawie cała logika Coqa)
- predykaty
  (np. równość)

Coq — typy indukcyjne

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

Coq — typy indukcyjne

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

- Typ indukcyjny:

```
bool : Type
```

Coq — typy indukcyjne

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

- Typ indukcyjny:

```
bool : Type
```

- Konstruktory:

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

Coq — typy indukcyjne

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

- Typ indukcyjny:

```
bool : Type
```

- Konstruktory:

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

- Analiza przypadków:

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

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).
```

- Rodzina predykatów indukcyjnych:

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

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).
```

- Rodzina predykatów indukcyjnych:

```
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).
```

- Rodzina predykatów indukcyjnych:

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

- Konstruktory:

```
sorted0
sorted1
sorted2
```

Architektura Coq

Zasada De Bruijna (“małe” jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:

Architektura Coq

Zasada De Bruijna (“małe” jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC

Architektura Coq

Zasada De Bruijna (“małe” jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje

Architektura Coq

Zasada De Bruijna (“małe” jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).

Architektura Coq

Zasada De Bruijna (“małe” jądro, zewnętrznie sprawdzalne termy)

- jądro / kernel (≈20KLOC), odpowiedzialne za:
  - typowanie w CIC
  - redukcje
  - środowisko (definicje, aksjomaty etc).
  - moduły

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:

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

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

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

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

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)

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

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)

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

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

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(beta) z eta-redukcją, strukturalną składnią dowodów

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 Java Card 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

Coq — język programowania

- abstrakcja i aplikacja

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja
- co-rekursja

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja
- co-rekursja
- typy zależne

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja
- co-rekursja
- typy zależne
- moduły i funktory

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja
- co-rekursja
- typy zależne
- moduły i funktory
- type classes

Coq — język programowania

- abstrakcja i aplikacja
- algebraiczne typy danych (liczby naturalne, listy, drzewa...) z pattern-matchingiem
- polimorphizm
- (structuralna) rekursja
- co-rekursja
- typy zależne
- moduły i funktory
- type classes
- ...

Coq — logika

- kwantyfikator ogólny i implikacja (“wbudowane”)

Coq — logika

- kwantyfikator ogólny i implikacja (“wbudowane”)
- spóniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)

Coq — logika

- kwantyfikator ogólny i implikacja (“wbudowane”)
- spóniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- logika wyższego rzędu

Coq — logika

- kwantyfikator ogólny i implikacja (“wbudowane”)
- spóniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- logika wyższego rzędu
- predykaty indukcyjne

Coq — logika

- kwantyfikator ogólny i implikacja (“wbudowane”)
- spóniki logiczne, fałsz, kwantyfikator egzystencjalny (zdefiniowane)
- logika wyższego rzędu
- predykaty indukcyjne
- ...

maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)

maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.

maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc tactyki: auto, intuition, etc.

maszyneria dowodowa w Coqu

- interaktywny tryb dowodowy (zarządzanie celem dowodowym i założeniami)
- wbudowane taktyki (budujące części termów dowodowych): intro, apply, etc.
- automatyczne ad-hoc tactyki: auto, intuition, etc.
- pocedury decyzyjne: omega, ring, field, tauto, etc.

maszyneria dowodowa w Coqu

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

Ekstrakcja programów

- “wyciągnięcie” programu realizującego dany algorytm z dowodu poprawności algorytmu,

Ekstrakcja programów

- “wyciągnięcie” programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją “z definicji”,

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 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,

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

Program, specifikacja i ekstrakcja

- dwie możliwości tworzenia poprawnych programów

Program, specifikacja i ekstrakcja

- dwie możliwości tworzenia poprawnych programów
  1. najpierw program, potem specyfikacja (łatwe typy + lematy)

Program, specifikacja i ekstrakcja

- dwie możliwości tworzenia poprawnych programów
  1. najpierw program, potem specyfikacja (łatwe typy + lematy)
  2. program i specyfikacja "naraz" (typy z własnościami)

Program, specifikacja i ekstrakcja

- dwie możliwości tworzenia poprawnych programów
  1. najpierw program, potem specyfikacja (łatwe typy + lematy)
  2. program i specyfikacja "naraz" (typy z własnościami)
- nowa funkcjonalność Coqa: ten drugi sposób stał się prostszy!

Przykład: sortowanie (insertion_sort.v)

Sekcja Program_Spec_later.

- Implementacja (insertion sort)
- Specyfikacja
- Dowód poprawności
- Ekstracja (program w Ocamlu/Haskellu)

Sekcja Program_and_Spec.

- Implementacja i specyfikacja razem (insertion sort)
- Dowód poprawności
- Ekstracja (program w Ocamlu/Haskellu)