Programowanie z typami zależnymi i dowodzenie twierdzeń

Daria Walukiewicz-Chrząszcz
(wykład)

Jacek Chrząszcz
(lab)

Termin: wtorek 10:15 (wykład) i 12:15 (lab)

Motywacja: poprawność programów

- Typy (statyczne typowanie)
- Dowodzenie własności programów

Zadanie z JPP

Zaimplementuj zrównoważone drzewa BST i operacje

```
...
toList :: Tree a -> [a]
fromList :: [a] -> Tree a
...
```

tak aby dla dowolnej listy xs spełnione były zależności:

```
...
Data.List.sort xs == toList (fromList xs)
...
```

Motywacja: poprawność programów

- Typy (statyczne typowanie)
- Dowodzenie własności programów

Zadanie z JPP

Zaimplementuj zrównoważone drzewa BST i operacje

```
...
toList :: Tree a -> [a]
fromList :: [a] -> Tree a
...
```

tak aby dla dowolnej listy xs spełnione były zależności:

```
...
Data.List.sort xs == toList (fromList xs)
...
```

Motywacja: poprawność programów

- Typy (statyczne typowanie)
- Dowodzenie własności programów

Zadanie z JPP

Zaimplementuj zrównoważone drzewa BST i operacje

```
...
toList :: Tree a -> [a]
fromList :: [a] -> Tree a
...
```

tak aby dla dowolnej listy xs spełnione były zależności:

```
...
Data.List.sort xs == toList (fromList xs)
...
```

Motywacja: poprawność programów

- Typy (statyczne typowanie)
- Dowodzenie własności programów

Zadanie z JPP

Zaimplementuj zrównoważone drzewa BST i operacje

```
...
toList :: Tree a -> [a]
fromList :: [a] -> Tree a
...
```

tak aby dla dowolnej listy xs spełnione były zależności:

```
...
Data.List.sort xs == toList (fromList xs)
...
```

Typy zależne

$$\texttt{Inductive } \mathit{ftree} : \mathit{nat} \to \texttt{Set} :=$$
$$\mid \mathit{Leaf} : \mathit{ftree}\ 0$$
$$\mid \mathit{Node} : \forall\ n, \mathit{ftree}\ n \to \mathit{ftree}\ n \to \mathit{ftree}\ (S\ n).$$

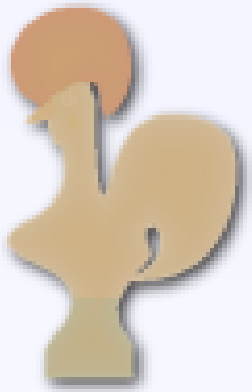
$$\mathit{tail} : \forall\ n, \mathit{list}\ (S\ n) \to \mathit{list}\ n.$$

Typy zależne

$$\texttt{Inductive}\ \mathit{ftree} : \mathit{nat} \to \texttt{Set} :=$$
$$\mid\ \mathit{Leaf} : \mathit{ftree}\ 0$$
$$\mid\ \mathit{Node} : \forall\ n,\ \mathit{ftree}\ n \to \mathit{ftree}\ n \to \mathit{ftree}\ (S\ n).$$

$$\mathit{tail} : \forall\ n,\ \mathit{list}\ (S\ n) \to \mathit{list}\ n.$$

Coq — system wspomagania dowodzenia

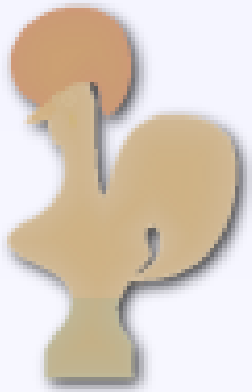


http://coq.inria.fr/

```
(***********)
(*  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

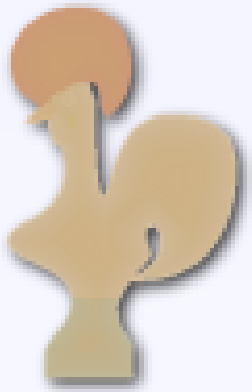


http://coq.inria.fr/

```
(***********)
(*  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

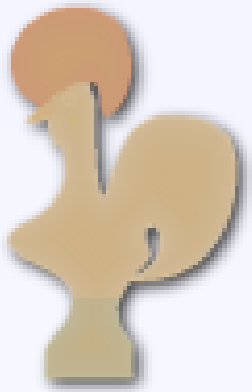


http://coq.inria.fr/

```
(***********)
(*  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

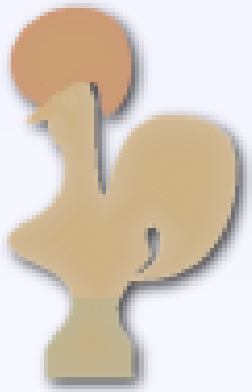


http://coq.inria.fr/

```
(***********)
(*  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

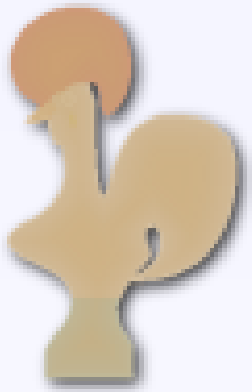


http://coq.inria.fr/

```
(***********)
(*  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



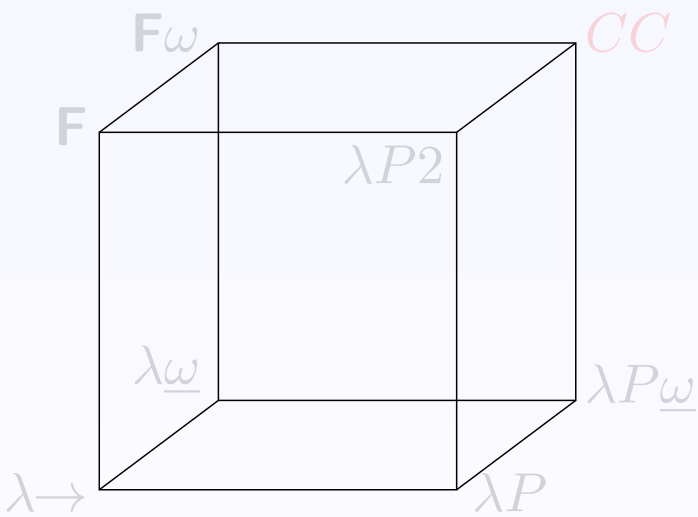
http://coq.inria.fr/

```
(***********)
(*  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)



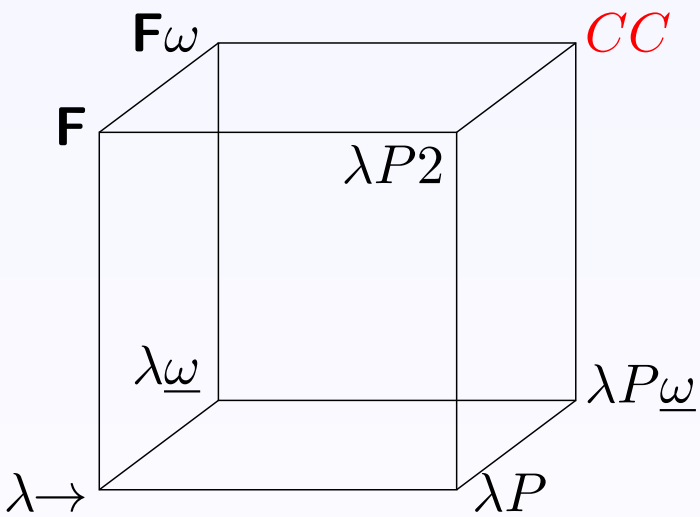
↑ polimorfizm

↗ konstruktory typów

→ typy zależne

Coq — podstawy teoretyczne

Coq — rachunek konstrukcji indukcyjnych (CIC)



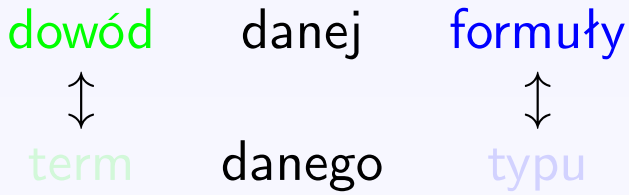
↑ polimorfizm

↗ 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

dowód	danej	formuły
↕		↕
term	danego	typu

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

Poprawność Coqa: poprawność algorytmu sprawdzania typów

Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

dowód	danej	formuły
↕		↕
term	danego	typu

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

Poprawność Coqa: poprawność algorytmu sprawdzania typów

Dowodzenie w Coqu

Izomorfizm Curry'ego-Howarda

dowód	danej	formuły
↕		↕
term	danego	typu

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

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

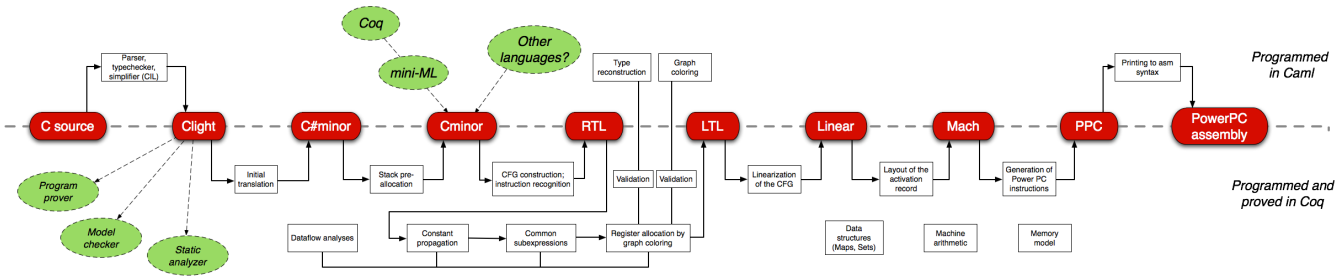
Coq — znaczące formalizacje

- 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

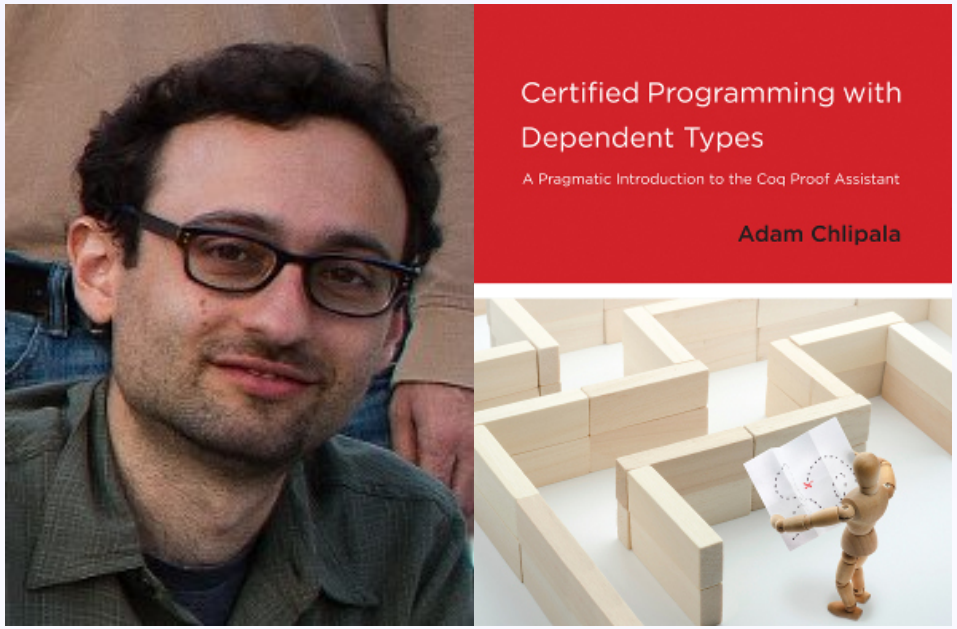
- "wyciągnięcie" programu realizującego dany algorytm z dowodu poprawności algorytmu,
- program wyekstrahowany jest poprawny ze specyfikacją *z definicji*,
- ekstrakcja polega na "wycieraniu" części logicznych z termu dowodowego,
- ekstrakcja możliwa dlatego, że mamy do czynienia z logiką intuicjonistyczną, 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

Certified Programming with Dependent Types

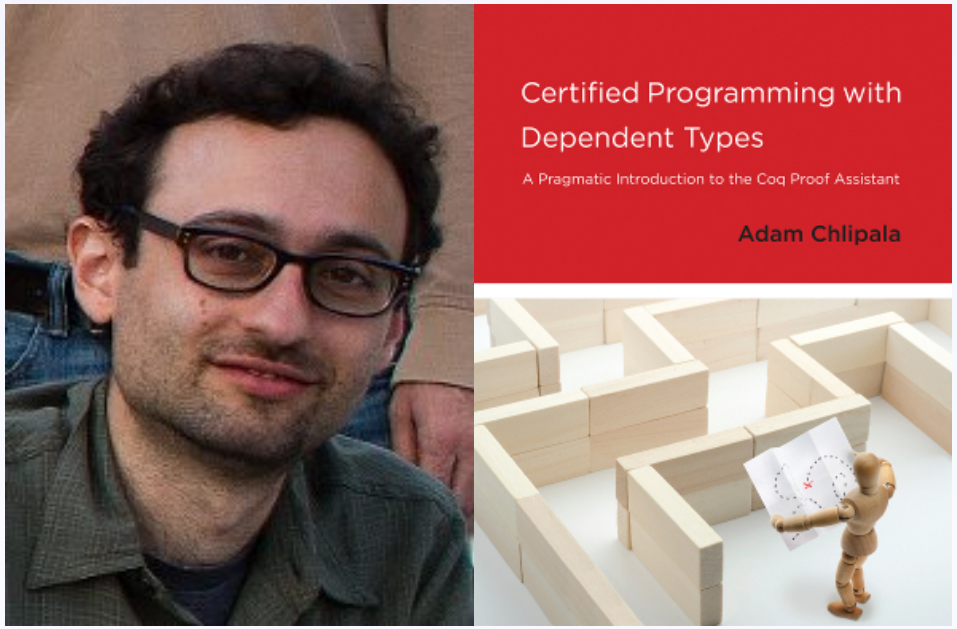
Adam Chlipala (MIT)



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

Certified Programming with Dependent Types

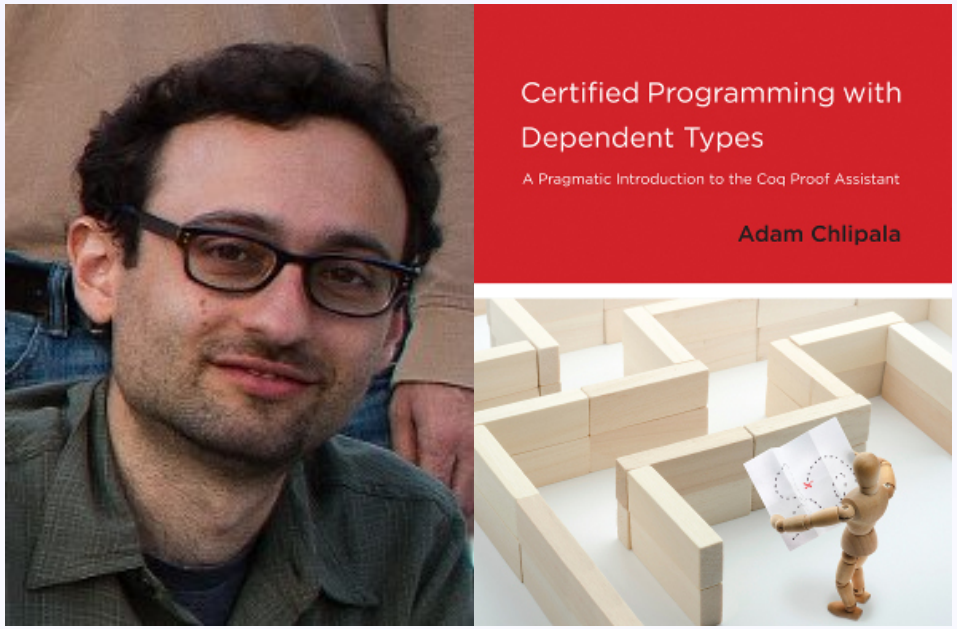
Adam Chlipala (MIT)



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

Certified Programming with Dependent Types

Adam Chlipala (MIT)



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

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Check Const 42.

Check Binop Plus (Const 2) (Const 2).

Check Binop Times (Binop Plus (Const 2) (Const 2)) (Const 7).
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Source language denotation

```
Inductive binop : Set := Plus | Times.

Inductive exp : Set :=
| Const : Z → exp
| Binop : binop → exp → exp → exp.

Definition binopDenote (b : binop) : Z → Z → Z :=
  match b with
  | Plus ⇒ fun x y ⇒ x + y
  | Times ⇒ fun x y ⇒ x × y
  end.

Fixpoint expDenote (e : exp) : Z :=
  match e with
  | Const n ⇒ n
  | Binop b e1 e2 ⇒ (binopDenote b) (expDenote e1) (expDenote e2)
  end.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Check iConst 42 :: nil.

Check iConst 2 :: iConst 2 :: iBinop Plus :: nil.

Check iConst 7 :: iConst 2 :: iConst 2 :: iBinop Plus :: iBinop Times :: nil.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack :=
  match i with
  | iConst n ⇒ Some (n :: s)
  | iBinop b ⇒
    match s with
    | arg1 :: arg2 :: s' ⇒ Some ((binopDenote b) arg1 arg2 :: s')
    | _ ⇒ None
    end
  end.
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack := ...
```

Single sort: Target language denotation

```
Inductive instr : Set :=
| iConst : Z → instr
| iBinop : binop → instr.

Definition prog := list instr.

Definition stack := list Z.

Definition instrDenote (i : instr) (s : stack) : option stack := ...

Fixpoint progDenote (p : prog) (s : stack) : option stack :=
  match p with
    | nil ⇒ Some s
    | i :: p' ⇒ match instrDenote i s with
                  | None ⇒ None
                  | Some s' ⇒ progDenote p' s'
                end
  end.
```

Single sort: Compilation

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Single sort: Compilation correctness

```
Fixpoint compile (e : exp) : prog :=
  match e with
  | Const n ⇒ iConst n :: nil
  | Binop b e1 e2 ⇒ compile e2 ++ compile e1 ++ iBinop b :: nil
  end.
```

```
Theorem compile_correct :
  ∀ e, progDenote (compile e) nil = Some (expDenote e :: nil).
```

```
Lemma compile_correct' : ∀ e p s,
  progDenote (compile e ++ p) s = progDenote p (expDenote e :: s).
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
             (MZConst 7) : mexp Mbool.
```

Many sorts: Source language

```
Inductive sort : Set := Mint | Mbool.

Inductive mbinop : sort → sort → sort → Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : ∀ s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.

Inductive mexp : sort → Set :=
| MZConst : Z → mexp Mint
| MBConst : bool → mexp Mbool
| MBinop : ∀ s1 s2 s, mbinop s1 s2 s → mexp s1 → mexp s2 → mexp s.

Check MBinop MPlus (MZConst 2) (MZConst 2) : mexp Mint.

Check MBinop (MEq Mint) (MBinop MPlus (MZConst 2) (MZConst 2))
                        (MZConst 7) : mexp Mbool.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with
    | Mint ⇒ Z
    | Mbool ⇒ bool
  end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ b e1 e2 ⇒
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
      (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
      (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Source language denotation

```
Definition sortDenote (s : sort) : Set :=
  match s with Mint ⇒ Z | Mbool ⇒ bool end.

Definition mbinopDenote arg1 arg2 res (b : mbinop arg1 arg2 res)
  : sortDenote arg1 → sortDenote arg2 → sortDenote res :=
  match b with
    | MPlus ⇒ Z.add
    | MTimes ⇒ Z.mul
    | MEq Mint ⇒ Z.eqb
    | MEq Mbool ⇒ Bool.eqb
    | MLt ⇒ Z.ltb
  end.

Fixpoint mexpDenote s (e : mexp s) : sortDenote s :=
  match e with
    | MZConst n ⇒ n
    | MBConst b ⇒ b
    | MBinop _ _ _ b e1 e2 ⇒
        (mbinopDenote b) (mexpDenote e1) (mexpDenote e2)
  end.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
    mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
   mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language

```
Definition sstack := list sort.

Inductive minstr : sstack → sstack → Set :=
| MiZConst : ∀ ss, Z → minstr ss (Mint :: ss)
| MiBConst : ∀ ss, bool → minstr ss (Mbool :: ss)
| MiBinop : ∀ arg1 arg2 res ss,
   mbinop arg1 arg2 res → minstr (arg1 :: arg2 :: ss) (res :: ss).

Inductive mprog : sstack → sstack → Set :=
| MNil : ∀ ss, mprog ss ss
| MCons : ∀ ss1 ss2 ss3, minstr ss1 ss2 → mprog ss2 ss3 → mprog ss1 ss3.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with
    | nil ⇒ unit
    | s :: ss' ⇒ sortDenote s × vstack ss'
  end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with
    | nil ⇒ unit
    | s :: ss' ⇒ sortDenote s × vstack ss'
  end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Target language denotation

```
Fixpoint vstack (ss : sstack) : Set :=
  match ss with nil ⇒ unit | s :: ss' ⇒ sortDenote s × vstack ss' end.

Check (5, (true, (false, ()))) : vstack (Mint::Mbool::Mbool::nil).

Definition minstrDenote ss ss' (i : minstr ss ss') : vstack ss → vstack ss' :=
  match i with
    | MiZConst _ n ⇒ fun vs ⇒ (n, vs)
    | MiBConst _ b ⇒ fun vs ⇒ (b, vs)
    | MiBinop _ _ _ _ b ⇒ fun vs ⇒
      let '(v1, (v2, vs')) := vs in ((mbinopDenote b) v1 v2, vs')
  end.

Fixpoint mprogDenote ss ss' (p : mprog ss ss') : vstack ss → vstack ss' :=
  match p with
    | MNil _ ⇒ fun vs ⇒ vs
    | MCons _ _ _ i p' ⇒ fun vs ⇒ mprogDenote p' (minstrDenote i vs)
  end.
```

Many sorts: Compilation

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.
```

```
Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).
```

```
Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.

Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).

Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

Many sorts: Compilation correctness

```
Fixpoint mcompile s (e : mexp s) (ss : sstack) : mprog ss (s :: ss) :=
  match e with
    | MZConst n ⇒ MCons (MiZConst _ n) (MNil _)
    | MBConst b ⇒ MCons (MiBConst _ b) (MNil _)
    | MBinop _ _ _ b e1 e2 ⇒ mconcat (mcompile e2 _)
      (mconcat (mcompile e1 _) (MCons (MiBinop _ b) (MNil _)))
  end.
```

```
Theorem mcompile_correct : ∀ s (e : mexp s),
  mprogDenote (mcompile e nil) () = (mexpDenote e, ()).
```

```
Lemma mcompile_correct' : ∀ s (e : mexp s) ss (s : vstack ss),
  mprogDenote (mcompile e ss) s = (mexpDenote e, s).
```

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 — indeksowana rodzina typów

```
Inductive mbinop : sort -> sort -> sort -> Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : forall s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.
```

- Indeksowana rodzina typów:

```
mbinop : sort -> sort -> sort -> Set
```

- Analiza przypadków:

```
Definition mbinopDenote arg1 arg2 res (b:mbinop arg1 arg2 res):
  sortDenote arg1 -> sortDenote arg2 -> sortDenote res :=
match b with
| MPlus => Z.add
| MTimes => Z.mul
| MEq Mint => Z.eqb
| MEq Mbool => Bool.eqb
| MLt => Z.ltb
end.
```

Coq — indeksowana rodzina typów

```
Inductive mbinop : sort -> sort -> sort -> Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : forall s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.
```

- Indeksowana rodzina typów:

```
mbinop : sort -> sort -> sort -> Set
```

- Analiza przypadków:

```
Definition mbinopDenote arg1 arg2 res (b:mbinop arg1 arg2 res):
  sortDenote arg1 -> sortDenote arg2 -> sortDenote res :=
match b with
| MPlus => Z.add
| MTimes => Z.mul
| MEq Mint => Z.eqb
| MEq Mbool => Bool.eqb
| MLt => Z.ltb
end.
```

Coq — indeksowana rodzina typów

```
Inductive mbinop : sort -> sort -> sort -> Set :=
| MPlus : mbinop Mint Mint Mint
| MTimes : mbinop Mint Mint Mint
| MEq : forall s, mbinop s s Mbool
| MLt : mbinop Mint Mint Mbool.
```

- Indeksowana rodzina typów:

```
mbinop : sort -> sort -> sort -> Set
```

- Analiza przypadków:

```
Definition mbinopDenote arg1 arg2 res (b:mbinop arg1 arg2 res):
  sortDenote arg1 -> sortDenote arg2 -> sortDenote res :=
match b with
| MPlus => Z.add
| MTimes => Z.mul
| MEq Mint => Z.eqb
| MEq Mbool => Bool.eqb
| MLt => Z.ltb
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
```