Obliczenia i wnioskowanie w systemie Coq

Małgorzata Biernacka

Instytut Informatyki UWr

Wykład 3
12.03.2013

Podsystemy CoC

$$(\Gamma \vdash T, U : s \quad s \in \{Set, Prop\})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash (\mathtt{fun}\ x : T \Rightarrow t) : T \to U}$$

$$\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

- rachunek lambda z typami prostymi
- minimalny intuicjonistyczny rachunek zdań

Podsystemy CoC

$$\frac{\Gamma \vdash T : \mathit{Set} \quad \Gamma, x : T \vdash U : s \quad s \in \{\texttt{Set}, \texttt{Prop}\}}{\Gamma \vdash (\texttt{forall}\ x : T, U) : s}$$

$$\frac{\Gamma \vdash (\texttt{forall}\ x : T, U) : s \quad s \in \{\mathit{Set}, \mathit{Prop}\} \quad \Gamma, x : T \vdash t : U}{\Gamma \vdash (\texttt{fun}\ x : T \Rightarrow t) : (\texttt{forall}\ x : T, U)}$$

$$\frac{\Gamma \vdash t : (\texttt{forall}\ x : U, T) \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T[x \mapsto u]}$$

- rachunek lambda z typami zależnymi
- minimalny intuicjonistyczny rachunek predykatów

Podsystemy CoC

$$\frac{\Gamma, x : \texttt{Prop} \vdash U : \texttt{Prop}}{\Gamma \vdash (\texttt{forall}\ x : \texttt{Prop}, U) : \texttt{Prop}} \qquad \frac{\Gamma, x : \texttt{Set} \vdash U : \texttt{Set}}{\Gamma \vdash (\texttt{forall}\ x : \texttt{Set}, U) : \texttt{Type}}$$

$$\frac{\Gamma \vdash (\texttt{forall}\ x : T, U) : s \quad \Gamma, x : T \vdash t : U}{\Gamma \vdash (\texttt{fun}\ x : T \Rightarrow t) : (\texttt{forall}\ x : T, U)}$$

$$\frac{\Gamma \vdash t : (\texttt{forall}\ x : U, T) \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T[x \mapsto u]}$$

- polimorficzny rachunek lambda (system F)
- intuicjonistyczna logika zdań 2. rzędu

Wykonywanie obliczeń w Coqu

- obliczenie to normalizacja termu według reguł redukcji
- typy redukcji
  - β-redukcja: aplikacja funkcji do argumentu (podstawienie)

$$E\ ;\ \Gamma \vdash (\texttt{fun}\ x : T\ \Rightarrow\ t)\ u \rhd_\beta t[x \mapsto u]$$

  - δ-redukcja: zastąpienie identyfikatora przez jego definicję (*unfolding*)

$$E\ ;\ \Gamma \vdash x \rhd_\delta t \quad \text{jeśli } x := t : T \in \Gamma \text{ lub } x := t : T \in E$$

  - ζ-redukcja: podstawienie lokalnej definicji w konstrukcji `let`

$$E\ ;\ \Gamma \vdash \texttt{let}\ x := u\ \texttt{in}\ t \rhd_\zeta t[x \mapsto u]$$

  - ι-redukcja: dla definicji indukcyjnych

Redukcja i normalizacja

- niech $\triangleright$ oznacza $\triangleright_{\beta\delta\zeta\iota}$
- term t redukuje się **w jednym kroku** do termu u, jeśli u otrzymujemy przez zastosowanie jednej z reguł redukcji w dowolnym podtermie t (redukcja w kontekście)

$$t \triangleright u$$

- domknięcie zwrotne i przechodnie redukcji jednokrokowej generuje relację redukcji **wielokrokowej**

$$t \triangleright^* u$$

- **postać normalna** to term, w którym nie da się zastosować redukcji
- **normalizacja** termu to proces sprowadzania termu do postaci normalnej (o ile istnieje) poprzez ciąg redukcji
- termy t i u są **konwertowalne**, jeśli istnieje term r taki, że $t \triangleright^* r$ i $u \triangleright^* r$ (ozn. $t = u$)
- w Coqu ważną rolę odgrywa **czołowa postać normalna** (czoło termu jest identyfikatorem, abstrakcją lub produktem; ciało termu nie jest redukowane)

Własności redukcji w Coqu

- **silna normalizacja** – każdy ciąg redukcji danego termu jest skończony
- **konfluencja** – jeśli $t \rhd^* t_1$ i $t \rhd^* t_2$, to istnieje term u taki, że $t_1 \rhd^* u$ i $t_2 \rhd^* u$
- **redukcja zachowuje typy** – jeśli term t typu T redukuje się do termu u, to u ma typ T
- w konsekwencji każdy term ma dokładnie jedną postać normalną i konwertowalność jest rozstrzygalna

Reguła typowania z konwersją

- aby uwzględnić konwersję przy typowaniu, dorzucamy dodatkową regułę typowania
- nowa reguła uwzględnia też hierarchię uniwersów – w tym celu definiujemy porządek $\leq_{\beta\delta\zeta\iota}$:
  - jeśli $E\ ;\ \Gamma \vdash t =_{\beta\delta\zeta\iota} u$, to $E\ ;\ \Gamma \vdash t \leq_{\beta\delta\zeta\iota} u$
  - $E\ ;\ \Gamma \vdash \mathtt{Type}_i \leq_{\beta\delta\zeta\iota} \mathtt{Type}_j$ dla dowolnych $i \leq j$
  - $E\ ;\ \Gamma \vdash \mathtt{Set} \leq_{\beta\delta\zeta\iota} \mathtt{Type}_i$ dla dowolnego i
  - $E\ ;\ \Gamma \vdash \mathtt{Prop} \leq_{\beta\delta\zeta\iota} \mathtt{Set}$ dla dowolnego i
  - jeśli $E\ ;\ \Gamma \vdash T =_{\beta\delta\zeta\iota} U$ oraz $E\ ;\ \Gamma, x : T \vdash T' \leq_{\beta\delta\zeta\iota} U'$, to

$$E\ ;\ \Gamma \vdash (\mathtt{forall}\ x : T, T') \leq_{\beta\delta\zeta\iota} (\mathtt{forall}\ x : U, U')$$

- reguła dla konwertowalności

$$\frac{E\ ;\ \Gamma \vdash U : s \quad E\ ;\ \Gamma \vdash t : T \quad E\ ;\ \Gamma \vdash T \leq_{\beta\delta\zeta\iota} U}{E\ ;\ \Gamma \vdash t : U}$$

- konwersje przeprowadzane w dowodzie są "transparentne" dla dowodu (nie są uwzględniane w wygenerowanym termie dowodu)

Taktyki dot. konwersji

- `change term`, `change term with term` – zastosowanie konwersji
- `red`, `hnf`, `simpl` – taktyki wykonujące obliczenia w bieżącym celu
- `unfold id` – wykonuje δ-redukcję
- komenda `Eval ... in t` – obliczenie wartości wyrażenia `t` wg zadanej strategii (`compute`, `cbv`, `lazy`, `cbv beta`, itp.)

Definicje indukcyjne a Coq

- indukcyjne typy danych/predykaty są definiowalne w CoC, ale taki sposób reprezentacji ma spore wady
- CIC jest rozszerzeniem CoC, w którym definicje indukcyjne są pojęciem pierwotnym

  (Ch. Paulin-Mohring "Inductive Definitions in the System Coq. Rules and Properties")

Definicje indukcyjne typów

- elementy typu ind. są definiowane za pomocą konstruktorów w skończonej liczbie kroków
- dowolny element typu ind. można otrzymać wyłącznie za pomocą konstruktorów
- konstruktory są parami różne (termy z nich zbudowane są różne)
- konstruktory są różnowartościowe (tj. jeśli `c t = c s`, to `t = s`)
- użycie: konstruowanie termów, eliminacja termów (definiowanie funkcji za pomocą rekursji prostej), dowodzenie własności przez indukcję

Definicje indukcyjne typów

- zbiór termów reprezentujących liczby naturalne możemy zapisać w składni Ocamla tak:

```
type nat = O | S of nat
```

- w Coqu:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

- wygodnie jest myśleć o definicji indukcyjnej używając reguł wnioskowania:

$$\frac{}{\vdash O : nat} \qquad \frac{\vdash t : nat}{\vdash St : nat}$$

Definicje indukcyjne typów w Coqu

Definicja użytkownika:

- nazwa typu jest stałą (termem) zdefiniowaną w Coqu o typie końcowym `Set` lub `Type` (`nat : Set`, `bool : Set`, `list : Type -> Type`, `array : nat -> Set`)
- każdy konstruktor typu T jest stałą (termem) o typie końcowym T

Warunek pozytywnych wystąpień (*strict positivity condition*) w definicjach indukcyjnych

- typ T spełnia warunek pozytywnych wystąpień dla identyfikatora X w termie T :
  - T postaci $XM_1 \ldots M_n$: X nie występuje w M_i
  - T postaci `forall` $x : V, U$: X występuje ściśle pozytywnie w V oraz U spełnia war. poz. wyst. dla X
- X występuje ściśle pozytywnie w T, gdy:
  - X nie występuje w T
  - $T = XM_1 \ldots M_n$ i X nie występuje w M_i
  - $T =$ `forall` $x : V, U$ i X nie występuje V oraz występuje ściśle pozytywnie w U
  - w pewnych przypadkach w parametrach typu indukcyjnego T (dokładniej: patrz Manual, 4.5.3)

Warunek pozytywnych wystąpień (*strict positivity condition*) w definicjach indukcyjnych

Negatywne wystąpienia identyfikatora typu definiowanego indukcyjnie są zabronione w typach argumentów konstruktorów tego typu

```
Inductive A : Set :=
| c : (A -> B) -> A.
```

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

Dlaczego musimy przestrzegać warunku pozytywnych wystąpień?

W przeciwnym razie

- nie zachodziłaby własność silnej normalizacji termów
- konwersja i sprawdzanie typów nie byłyby rozstrzygalne
- system byłby logicznie sprzeczny

```
Inductive A : Prop/Set :=
| c : (A -> B) -> A.

Definition F (t:A) : B :=
match t with
| c f => f t
end.

F (c F) -> ...
```

Definicje indukcyjne typów w Coqu

Każda definicja indukcyjna typu `T` powoduje dodanie do środowiska:

- definicji stałej `T` odpowiedniego typu
- deklaracji konstruktorów typu `T` jako stałych odpowiedniego typu (postaci `c : forall ..., T`)
- definicji stałych `T_ind`, `T_rect`, `T_rec` wygenerowanych automatycznie na podstawie definicji typu `T`
- `T_ind` – schemat indukcji strukturalnej dla `T`
- `T_rec` – schemat rekursji prostej dla `T`

Dopasowanie wzorca

- składnia `match`

```
Definition iszero (n:nat) : Prop :=
match n with
| O => True
| S m => False
end
```

- definicja za pomocą `match` musi obejmować wszystkie przypadki

Definicje rekurencyjne

- za pomocą komendy `Fixpoint`
- lub za pomocą wyrażenia `fix` (gdy definiujemy funkcję rekurencyjną lokalnie)

```
Fixpoint fact (n:nat) {struct n} : nat :=
match n with
| O => 1
| S m => n * fact m
end.

Definition fact2 := fix fact (n:nat) : nat :=
match n with
| O => 1
| S m => n * fact m
end.
```

- obliczenie musi się zatrzymać – wywołanie rekurencyjne musi dotyczyć właściwego podtermu wskazanego argumentu

ι-REDUKCJA

- `match` + konstruktory typu wprowadzają nowy rodzaj redeksu
- ι-redukcja dla typu T o konstruktorach c_i (każdy o k_i argumentach), $1 \leq i \leq n$:

$$
\begin{aligned}
&\texttt{match } c_i\ a_1 \texttt{ ... } a_{k_i} \texttt{ with} \\
&\texttt{| } c_1\ p_1 \texttt{ ... } p_{k_1} \texttt{ => } t_1 \\
&\texttt{| ...} \\
&\texttt{| } c_n\ p_1 \texttt{ ... } p_{k_n} \texttt{ => } t_n \\
&\texttt{end} \\
&\longrightarrow t_i[a_1/p_1; a_2/p_2; \ldots; a_{k_i}/p_{k_i}]
\end{aligned}
$$

Zasada indukcji

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n))
       -> forall n : nat, P n
```

```
nat_rect =
fun (P : nat -> Type) (f : P 0)
                      (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

Taktyki dla eliminacji typów indukcyjnych

- `induction n`
- `elim` *t*
- `case` *t*
- `destruct H`
- `discriminate H`
- `injection H`

Taktyka `induction`

- `induction t` – wykonuje indukcję ze względu na typ termu `t` (musi być indukcyjny):
  - bieżący cel jest traktowany jako parametr `P` w definicji zasady indukcji
  - taktyka generuje nowe podcele dla każdego z konstruktorów typu
  - taktyka wprowadza do przesłanek odpowiednie hipotezy indukcyjne

Taktyka `elim`

- `elim t` – prostsza niż `induction`
  - wyszukuje odpowiednią zasadę eliminacji dla typu `t`
  - generuje nowe podcele dla każdego z konstruktorów typu
  - nie modyfikuje kontekstu
  - działa jak `apply ...`

Taktyka `case`

- `case t` – dowód przez rozpatrywanie przypadków; typ termu `t` musi być indukcyjny
- działa podobnie do `elim t`

Taktyka `destruct`

- `destruct t` – dowód przez rozpatrywanie przypadków; typ termu `t` musi być indukcyjny
- działa podobnie do `induction t`, ale nie generuje hipotezy indukcyjnej

Taktyka `discriminate`

- taktyka wykorzystuje fakt, że dwa termy zbudowane za pomocą dwóch różnych konstruktorów tego samego typu nie mogą być równe
- `discriminate H` pozwala udowodnić dowolny cel, gdy w przesłance `H : t1 = t2` postaci normalne termów `t1` i `t2` mają w pewnym podtermie różne konstruktory na tym samym miejscu
- działanie `discriminate` można zasymulować za pomocą prostszych taktyk

Taktyka `injection`

- taktyka wykorzystuje fakt, że konstruktory typu są różnowartościowe
- jeśli `c t1 ... tn = c s1 ... sn`, to `ti = si` dla $i = 1, \ldots, n$
- `injection H` generuje cel `t1 = s1 -> ... -> tn = sn -> G`, gdy `H : c t1 ... tn = c s1 ... sn`
- działanie `injection` można zasymulować za pomocą prostszych taktyk

Taktyki wprowadzania typów indukcyjnych

- `constructor` – zastosowanie kontruktora do zbudowania termu typu indukcyjnego
- `apply c` – j.w. (c jest konstruktorem)