Dependent type-ok és az Agda

Dr. Érdi Gergő
http://gergo.erdi.hu/

ELTE IK, 2009. november 10.

module *AgdaEloadas* **where**

Magasabb kind-ú típusok

Haskell: Típus függhet más típustól: pl.

$$\textbf{data}\ \mathit{List}\ \tau\ =\ \mathit{Nil}\ \mid\ \mathit{Cons}\ \tau\ (\mathit{List}\ \tau)$$

Ez mit is jelent? A *List* valójában egy típuskonstrukciós művelet: ha adott egy τ típus, akkor ahhoz létezik egy *List* τ (algebrai) típus is, a megfelelő két konstruktorral. Vagyis a *List* típusról képez le típusra:

$$\mathit{List}\ :\ \star\ \rightarrow\ \star$$

Ahol a $\star$ jelenti a "valamilyen típust".
Lényeg: típusok és értékek világa egymástól teljesen el van választva.
Ugyanez Agdában:

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where}\\ \quad \mathit{nil}\ :\ \ \ \mathit{List}\ A\\ \quad \mathit{cons}\ :\ A\ \rightarrow\ \mathit{List}\ A\ \rightarrow\ \mathit{List}\ A\end{array}$$

Dependent type-ok

Agda: Típus értéktől is függhet
Mit jelent ez? Például tegyük fel, hogy adott a természetes számok típusa:

$$\begin{aligned}&\textbf{data}\ \mathit{Nat}\ :\ \mathit{Set}\ \textbf{where}\\&\quad \mathit{zero}\ :\ \mathit{Nat}\\&\quad \mathit{succ}\ :\ \mathit{Nat}\ \rightarrow\ \mathit{Nat}\end{aligned}$$

Ekkor értelmezhetjük egy adott n-re az n hosszú vektorok típusát:

$$\begin{aligned}&\textbf{data}\ \mathit{Vec}\ (A\ :\ \mathit{Set})\ :\ \mathit{Nat}\ \rightarrow\ \mathit{Set}\ \textbf{where}\\&\quad \mathit{nil}\ \ \ :\ \mathit{Vec}\ A\ \mathit{zero}\\&\quad \mathit{cons}\ :\ \{n\ :\ \mathit{Nat}\}\ \rightarrow\ A\ \rightarrow\ \mathit{Vec}\ A\ n\ \rightarrow\ \mathit{Vec}\ A\ (\mathit{succ}\ n)\end{aligned}$$

Mire jó ez? Programhelyesség

Vessük össze az alábbi két *map* változatot:

$$\begin{aligned}
map &: \{A\ B : Set\} \to \\
&\quad (A \to B) \to List\ A \to List\ B \\
map\ f\ \ nil &= nil \\
map\ f\ \ (cons\ x\ xs) &= cons\ (f\ x)\ (map\ f\ xs)
\end{aligned}$$

$$\begin{aligned}
vmap &: \{A\ B : Set\}\ \{n : Nat\} \to \\
&\quad (A \to B) \to Vec\ A\ n \to Vec\ B\ n \\
vmap\ f\ \ nil &= nil \\
vmap\ f\ \ (cons\ x\ xs) &= cons\ (f\ x)\ (vmap\ f\ xs)
\end{aligned}$$

Abból, hogy ez típushelyes, következik, hogy tényleg ugyanolyan hosszú vektort ad vissza, mint a bemenet:

$$\begin{aligned}
vmap' &: \{A\ B : Set\}\ \{n : Nat\} \to \\
&\quad (A \to B) \to Vec\ A\ n \to Vec\ B\ n \\
vmap'\ f\ _ &= nil
\end{aligned}$$

Mire jó ez? Programhelyesség

Vessük össze az alábbi két *map* változatot:

$$
\begin{aligned}
&map : \{A\ B : Set\} \to \\
&\qquad (A \to B) \to List\ A \to List\ B \\
&map\ f\ \ nil \qquad\qquad = nil \\
&map\ f\ \ (cons\ x\ xs) = cons\ (f\ x)\ (map\ f\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&vmap : \{A\ B : Set\}\ \{n : Nat\} \to \\
&\qquad (A \to B) \to Vec\ A\ n \to Vec\ B\ n \\
&vmap\ f\ \ nil \qquad\qquad = nil \\
&vmap\ f\ \ (cons\ x\ xs) = cons\ (f\ x)\ (vmap\ f\ xs)
\end{aligned}
$$

Abból, hogy ez típushelyes, következik, hogy tényleg ugyanolyan hosszú vektort ad vissza, mint a bemenet:

$$
\begin{aligned}
&vmap' : \{A\ B : Set\}\ \{n : Nat\} \to \\
&\qquad (A \to B) \to Vec\ A\ n \to Vec\ B\ n \\
&vmap'\ f\ \_ = nil
\end{aligned}
$$
-- Fordítási hibát ad!

Mire jó ez? Előfeltétel-ellenőrzés

Értelmezhetjük egy adott n-re az n-nél kisebb természetes számok típusát:

```
data FNat : Nat → Set where
  zero : {n : Nat} → FNat (succ n)
  succ : {n : Nat} → FNat n → FNat (succ n)
```

Ennek a segítségével le tudjuk írni a vektor-indexelés műveletét:

```
_!!_ : {A : Set} {n : Nat} → Vec A n → FNat n → A
nil         !! ()          -- FNat zero típusnak nincs eleme!
(cons x xs) !! zero        = x
(cons x xs) !! (succ m)    = xs !! m
```

Tételbizonyítás

Példa: Természetes számok összeadása kommutál.

$$\vdash \forall n, m \in \mathbb{N} : n + m = m + n$$

Hogyan tudjuk a problémát megtámadni? Szedjük szét végtelensok résztételre:

$$\{n \in \mathbb{N}, m \in \mathbb{N}\} \vdash n + m = m + n$$

Ha van egy konkrét n és m, amiről annyit tudunk, amennyit a baloldal állít, és ebből be tudjuk rájuk bizonyítani a jobboldalt, akkor készen vagyunk.

Tételbizonyítás: Reprezentáció

$$
\begin{aligned}
\_+\_ &: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{zero} \quad &+ m = m \\
\mathit{succ}\ n &+ m = \mathit{succ}\ (n + m)
\end{aligned}
$$

A továbbiakban csakis erről az általunk definiált összeadásról tudunk bármit is állítani és bizonyítani! (modellezés fontossága)
Az állításokból típusokat csinálunk:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Eq}\ \{A : \mathit{Set}\} : A \to A \to \mathit{Set}\ \textbf{where} \\
&\quad \mathit{refl} : \{x : A\} \to \mathit{Eq}\ x\ x
\end{aligned}
$$

Tételbizonyítás: Konstruktivitás

Az előbbiek értelmében egy tételt akkor tudunk bebizonyítani, ha minden konkrét esethez létre tudjuk az állítást hozni vagyis van egy *függvény*, ami általában elő tudja állítani az állítást, azaz egy megfelelő elemét az állítást reprezentáló típusnak.
Vagyis az összeadás kommutativitásának tétele az, hogy létezik egy olyan függvény, amelynek típusa:

$$thmPlusComm\ :\ (n\ m\ :\ Nat)\ \rightarrow\ Eq\ (n + m)\ (m + n)$$

Indukció $\equiv$ rekurzió

Tételbizonyítás ≡ Bizonyításgenerálás

Kezdjük el megírni a fenti függvényt

$$
\begin{array}{lll}
\mathit{thmPlusComm}\ \mathit{zero} & \mathit{zero} = \mathit{refl} & \\
\mathit{thmPlusComm}\ \mathit{zero} & (\mathit{succ}\ m) = ?? & \text{-- } 0 + m' \equiv m' + 0 \\
\mathit{thmPlusComm}\ (\mathit{succ}\ n)\ \mathit{zero} & = ?? & \text{-- } n' + 0 \equiv 0 + n' \\
\mathit{thmPlusComm}\ (\mathit{succ}\ n)\ (\mathit{succ}\ m) & = ?? & \text{-- } n' + m' \equiv m' + n'
\end{array}
$$

Ehhez szükségünk lesz az egyenlőség kongruencia-tulajdonságára és tranzitivitására:

$$
\begin{array}{l}
\mathit{lemEqCong} : \{A\ B : \mathit{Set}\}\ \{x\ y : A\}\ (f : A \to B) \to \\
\qquad \mathit{Eq}\ x\ y \to \mathit{Eq}\ (f\ x)\ (f\ y) \\
\mathit{lemEqCong}\ e\ \mathit{refl} = \mathit{refl}
\end{array}
$$

$$
\begin{array}{l}
\mathit{lemEqTran} : \{A\ B : \mathit{Set}\}\ \{x\ y\ z : A\} \to \\
\qquad \mathit{Eq}\ x\ y \to \mathit{Eq}\ y\ z \to \mathit{Eq}\ x\ z \\
\mathit{lemEqTran}\ \mathit{refl}\ \mathit{refl} = \mathit{refl}
\end{array}
$$

Tételbizonyítás: *thmPlusComm*

$$thmPlusComm : (n\ m : Nat) \to Eq\ (n + m)\ (m + n)$$

- $0 + 0 \equiv 0 + 0$: Mindkét oldal 0-val egyenlő

```
thmPlusComm zero zero = refl
```

- $0 + m' \equiv m' + 0$: Azonnal következik
  $0 + m \equiv m + 0$-ból, mindkét oldal rákövetkezőjét véve

```
thmPlusComm zero (succ m) =
  lemEqCong succ (thmPlusComm zero m)
```

- $n' + 0 \equiv 0 + n'$: Hasonlóan

```
thmPlusComm (succ n) zero =
  lemEqCong succ (thmPlusComm n zero)
```

Tételbizonyítás: *thmPlusComm* (folyt.)

- $n' + m' \equiv m' + n'$
  Ez már keményebb dió, itt használjuk ki az egyenlőség tranzitivitását:

$$
\begin{aligned}
n' + m' &\equiv (n + m')' && (1)\\
&\equiv (m' + n)' && (2)\\
&\equiv ((m + n)')' && (3)\\
&\equiv ((n + m)')' && (4)\\
&\equiv (n' + m)' && (3')\\
&\equiv (m + n')' && (5)\\
&\equiv m' + n' \quad \square && (1')
\end{aligned}
$$

```
thmPlusComm (succ n) (succ m) =
  lemEqCong succ                                  -- (1)
    (lemEqTran {Nat} {Nat}
      (thmPlusComm n (succ m))                    -- (2)
        (lemEqTran {Nat} {Nat}
          (lemEqCong succ                         -- (3)
            (thmPlusComm m n))                    -- (4)
          (thmPlusComm (succ n) m)))              -- (5)
```

Tételbizonyítás: Mit jelent a *thmPlusComm*?

- Élő ember nem akarja meghívni ezt a függvényt
- De a megléte bizonyítja azt, hogy az $(n\ m\ :\ Nat)\ \rightarrow\ Eq\ (n + m)\ (m + n)$ típusnak van eleme!

Tehát a fenti típus olvasható úgy is, mint egy következtetés:

$$n, m \in \mathbb{N} \vdash n + m = m + n$$

Hasonlóan a *lemEqCong* típusa:
$\{x\ y\ :\ A\}\ (f\ :\ A\ \rightarrow\ B)\ \rightarrow\ Eq\ x\ y\ \rightarrow\ Eq\ (f\ x)\ (f\ y)$ nem jelent mást, mint

$$x, y \in A, f \in A \rightarrow B \vdash (x = y) \Rightarrow (f(x) = f(y))$$

Vagyis az Agda megvalósítja a *Curry-Howard izomorfizmust*:

Igaz állítás $\leftrightarrows$ "Belakható" típus

Tételbizonyítás: Helyesség és teljesség

- *Helyesség*: Ha csak az Agda implementációban valamit nem szúrtak el, akkor nem tudunk "hamis" típushoz elemet találni: pl. nincs olyan kifejezés, aminek a típusa
  $(n : Nat) \to Eq\ n\ (succ\ n)$
  Ehhez szükséges, hogy divergens kifejezéseket ne fogadjunk el, például az alábbi Agda program hibás:

$$
\begin{aligned}
&nonsense : (n : Nat) \to Eq\ n\ (succ\ n)\\
&nonsense\ n = nonsense\ n
\end{aligned}
$$

- *Teljesség*: A fentiek alapján csak primitív rekurzív kifejezéseket használhatunk! (különben a típusellenőrzéshez meg kellene oldani a megállási problémát...) Például az Ackermann-függvényt nem tudjuk definiálni:

$$
\begin{aligned}
&Ack : Nat \to Nat \to Nat\\
&Ack\ zero\ x = x + 1\\
&Ack\ (succ\ i)\ zero = Ack\ i\ 1\\
&Ack\ (succ\ i)\ (succ\ x) = Ack\ i\ (Ack\ (i + 1)\ x)
\end{aligned}
$$

Tételbizonyítás: Helyesség és teljesség

- *Helyesség*: Ha csak az Agda implementációban valamit nem szúrtak el, akkor nem tudunk "hamis" típushoz elemet találni: pl. nincs olyan kifejezés, aminek a típusa
  $(n : Nat) \to Eq\ n\ (succ\ n)$
  Ehhez szükséges, hogy divergens kifejezéseket ne fogadjunk el, például az alábbi Agda program hibás:

$$nonsense : (n : Nat) \to Eq\ n\ (succ\ n)$$
$$nonsense\ n = nonsense\ n$$

- *Teljesség*: A fentiek alapján csak primitív rekurzív kifejezéseket használhatunk! (különben a típusellenőrzéshez meg kellene oldani a megállási problémát...) Például az Ackermann-függvényt nem tudjuk definiálni:

$$\begin{aligned} &Ack : Nat \to Nat \to Nat \\ &Ack\ zero\ x = x + 1 \\ &Ack\ (succ\ i)\ zero = Ack\ i\ 1 \\ &Ack\ (succ\ i)\ (succ\ x) = Ack\ i\ (Ack\ (i + 1)\ x) \end{aligned}$$

Tételbizonyítás: Helyesség és teljesség

- *Helyesség*: Ha csak az Agda implementációban valamit nem szúrtak el, akkor nem tudunk "hamis" típushoz elemet találni: pl. nincs olyan kifejezés, aminek a típusa
  $(n : Nat) \to Eq\ n\ (succ\ n)$
  Ehhez szükséges, hogy divergens kifejezéseket ne fogadjunk el, például az alábbi Agda program hibás:

$$
\begin{aligned}
&nonsense : (n : Nat) \to Eq\ n\ (succ\ n)\\
&nonsense\ n = nonsense\ n
\end{aligned}
$$

- *Teljesség*: A fentiek alapján csak primitív rekurzív kifejezéseket használhatunk! (különben a típusellenőrzéshez meg kellene oldani a megállási problémát...) Például az Ackermann-függvényt nem tudjuk definiálni:

$$
\begin{aligned}
&Ack : Nat \to Nat \to Nat\\
&Ack\ zero\ x = x + 1\\
&Ack\ (succ\ i)\ zero = Ack\ i\ 1\\
&Ack\ (succ\ i)\ (succ\ x) = Ack\ i\ (Ack\ (i + 1)\ x)
\end{aligned}
$$

(Ackermann-függvény másodrendű primitív rekurzív)

$$
\begin{aligned}
&Ack' \;:\; Nat \;\to\; Nat \;\to\; Nat \\
&Ack'\ zero \;=\; \lambda\, x \;\to\; x + 1 \\
&Ack'\ (succ\ i) \;=\; ack' \\
&\quad \textbf{where}\ ack' \;:\; Nat \;\to\; Nat \\
&\qquad ack'\ zero \;=\; Ack'\ i\ 1 \\
&\qquad ack'\ (succ\ x) \;=\; Ack'\ i\ (ack'\ x)
\end{aligned}
$$

Összefoglalva: Hogyan lesz ebből programhelyesség?

- Ha ügyesek a típusaink, a függvényeink típusa jól leírja a specifikációjukat (ld. *map* vs. *vmap*). Ha átmegy a típusellenőrzőn, akkor megfelel a specifikációnak.
- A függvényeink tulajdonságait tételként is megfogalmazhatjuk, és az imént látott módon bizonyíthatjuk őket. Például ha adott az *id* függvény:

$$
\begin{aligned}
&id \;:\; \{A \;:\; Set\} \;\rightarrow\; A \;\rightarrow\; A \\
&id\ x \;=\; x
\end{aligned}
$$

akkor ennek a lényeges tulajdonsága:

$$
\begin{aligned}
&thmIdId \;:\; \{A \;:\; Set\}\,\{x \;:\; A\} \;\rightarrow\; Eq\ x\ (id\ x) \\
&thmIdId \;=\; refl
\end{aligned}
$$

Köszönöm a figyelmet

További információ:

- Ulf Norell: *Dependently Typed Programming in Agda*
- *Agda wiki*
- Nordström, Petersson, Smith: *Programming in Martin-Löf's Type Theory*

Példa

Az alábbiakban egy nemtriviális tulajdonságot mutatok be: azt, hogy egy lista rendezett.

Példa: Rendezés

```
Rel : Set → Set₁
Rel A = A → A → Set

data _≡_ {A : Set} : Rel A where
  refl : {x : A} → x ≡ x

lem-cong : forall {A B x y} (f : A → B) →
           x ≡ y → (f x) ≡ (f y)
lem-cong p refl = refl

data Reflexive {A : Set} (_∼_ : Rel A) : Set where
  refl : ((x : A) → x ∼ x) → Reflexive _∼_

data Transitive {A : Set} (_∼_ : Rel A) : Set where
  tran : ({x y z : A} → x ∼ y → y ∼ z → x ∼ z) → Transitive _∼_

data Antisymmetric {A : Set} (_∼_ : Rel A) : Set where
  asym : ({x y : A} → x ∼ y → y ∼ x → x ≡ y) → Antisymmetric _∼

data Ordering {A : Set} (_≤_ : Rel A) : Set where
  ord : Reflexive _≤_ → Transitive _≤_ →
        Antisymmetric _≤_ → Ordering _≤_
```

Példa: $\mathbb{N}$ rendezése

$$\begin{aligned}
&\textbf{data}\ \cdot \leq_{\mathbb{N}} \cdot\ :\ \mathit{Nat}\ \to\ \mathit{Nat}\ \to\ \mathit{Set}\ \textbf{where}\\
&\quad \mathit{least}\ :\ (n\ :\ \mathit{Nat})\ \to\ \mathit{zero}\ \leq_{\mathbb{N}}\ n\\
&\quad \mathit{cong}\ :\ \{n\ m\ :\ \mathit{Nat}\}\ \to\ n\ \leq_{\mathbb{N}}\ m\ \to\ n'\ \leq_{\mathbb{N}}\ m'
\end{aligned}$$

$$\begin{aligned}
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{refl}\ :\ \mathit{Reflexive}\ \cdot \leq_{\mathbb{N}} \cdot\\
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{refl}\ =\ \mathit{refl}\ \mathit{refl}'\\
&\quad \textbf{where}\ \mathit{refl}'\ :\ (n\ :\ \mathit{Nat})\ \to\ n\ \leq_{\mathbb{N}}\ n\\
&\qquad \mathit{refl}'\ \mathit{zero}\ =\ \mathit{least}\ \mathit{zero}\\
&\qquad \mathit{refl}'\ y'\ =\ \mathit{cong}\ (\mathit{refl}'\ y)\\
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{tran}\ :\ \mathit{Transitive}\ \cdot \leq_{\mathbb{N}} \cdot\\
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{tran}\ =\ \mathit{tran}\ \mathit{tran}'\\
&\quad \textbf{where}\ \mathit{tran}'\ :\ \{x\ y\ z\ :\ \mathit{Nat}\}\ \to x\ \leq_{\mathbb{N}}\ y\ \to y\ \leq_{\mathbb{N}}\ z\ \to\ x\ \leq_{\mathbb{N}}\ z\\
&\qquad \mathit{tran}'\ \{z\ =\ z\}\ (\mathit{least}\ _)\ q\ \ =\ \mathit{least}\ z\\
&\qquad \mathit{tran}'\ \ (\mathit{cong}\ p)\ (\mathit{cong}\ q)\ =\ \mathit{cong}\ (\mathit{tran}'\ p\ q)\\
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{asym}\ :\ \mathit{Antisymmetric}\ \cdot \leq_{\mathbb{N}} \cdot\\
&\mathit{thm}-\mathit{nat}-\leqslant-\mathit{asym}\ =\ \mathit{asym}\ \mathit{asym}'\\
&\quad \textbf{where}\ \mathit{asym}'\ :\ \{x\ y\ :\ \mathit{Nat}\}\ \to\ x\ \leq_{\mathbb{N}}\ y\ \to\ y\ \leq_{\mathbb{N}}\ x\ \to\ x\ \equiv\ y\\
&\qquad \mathit{asym}'\ (\mathit{least}\ .0)\ (\mathit{least}\ .0)\ =\ \mathit{refl}\\
&\qquad \mathit{asym}'\ (\mathit{cong}\ p)\ \ (\mathit{cong}\ q)\ \ =\ \mathit{lem}-\mathit{cong}\ \ \cdot'\ \ (\mathit{asym}'\ p\ q)\\
&\mathit{thm}-\mathit{nat}-\leqslant\ :\ \mathit{Ordering}\ \cdot \leq_{\mathbb{N}} \cdot\\
&\mathit{thm}-\mathit{nat}-\leqslant\ =\ \mathit{ord}\ \mathit{thm}-\mathit{nat}-\leqslant-\mathit{refl}\ \mathit{thm}-\mathit{nat}-\leqslant-\mathit{tran}\ \mathit{thm}-\mathit{nat}-\leqslant-\mathit{asym}
\end{aligned}$$

Példa: Rendezett sorozatok

$$
\begin{aligned}
&\textbf{data}\ \mathit{Sorted}\ \{A : \mathit{Set}\}\ \{\cdot \leq \cdot : A \to A \to \mathit{Set}\}\ (\mathit{ord} : \mathit{Ordering}\ \cdot \leq \cdot) \\
&\qquad \{n : \mathit{Nat}\} \to \mathit{Vec}\ A\ n \to \mathit{Set}\ \textbf{where} \\
&\quad \mathit{trivial0} : \mathit{Sorted}\ \mathit{ord}\ [\,] \\
&\quad \mathit{trivial1} : (x : A) \to \mathit{Sorted}\ \mathit{ord}\ x : [\,] \\
&\quad \mathit{inherit} : \{n : \mathit{Nat}\}\ \{x\ y : A\}\ \{\mathit{ys} : \mathit{Vec}\ A\ n\} \\
&\qquad \to x \leqslant y \to \mathit{Sorted}\ \mathit{ord}\ y : \mathit{ys} \\
&\qquad \to \mathit{Sorted}\ \mathit{ord}\ x : y : \mathit{ys}
\end{aligned}
$$