Játék a rekurzióval

Feladat: Adott egy `p` függvény, ami egy végtelen bitsorozatból kiszámol egy logikai értéket. Döntsük el róla, hogy kielégíthető-e, azaz hogy van-e olyan bitsorozat, amire igazat ad vissza!

Apró betűs rész

- Az általunk megírandó függvény különféle bitsorozatokon tesztelheti a `p` függvényt (és csak ezt tudja, tehát pl. nem olvashatja `p` kódját).
- A `p` függvényről feltesszük, hogy minden bemenetre véges időn belül lefut és hogy tiszta (*pure*), azaz a visszatérési értéke csak a paraméterétől függ és egyéb mellékhatása sincs.
- Az általunk megírandó függvények is tisztának kell lennie és minden `p`-re véges időben le kell futnia.

Végtelen bitsorozatok

- Végtelen bitsorozat alatt most tetszőleges olyan függvényt értünk, ami nemnegatív egész számokhoz rendel logikai értéket (és minden nemnegatív egészre véges időben lefut).
- Lehetne ezeket máshogyan reprezentálni (pl. Haskell lustán kiértékelt végtelen listák), de ez bizonyos szempontból hatékonyabb (az első megoldás működne végtelen listákkal is, de a másodikban majd kihasználjuk, hogy függvényeink vannak).

Haskell szintaxis

- A függvények meghívását egyszerű egymás után írás jelöli: „`f x`” felel meg *f*(*x*)-nek.
- Ez nagyon erősen köt és balra asszociatív, azaz `f x*y` ≡ `(f x)*y` és `f x y` ≡ `f x) y`
- Az egyváltozós függvény típusa „*paraméter típusa* `->` *eredmény típusa*”
- A többváltozós függvények valójában olyan függvények, amik elfogadnak egy paramétert és visszaadják azt a függvényt, ami várja a többit, tehát pl. egy kétváltozós függvény típusa `a -> (b -> c)` (itt a zárójelek elhagyhatóak).

A felhasznált típusok

```
type Natural = Integer
```

{- „`type`” a typedef Haskell megfelelője, `Integer` a „nagyon” nagy egészek típusa, amire itt bevezetünk egy másik nevet, hogy ezzel is jelezzük, hogy a program mindenütt csak nemnegatív értékekkel számol -}

```
type Bitseq = Natural -> Bool
```

{- a bitsorozatok olyan függvények, amik természetes számból igaz/hamis értéket csinálnak -}

```
forsome :: (Bitseq -> Bool) -> Bool
```

{- itt deklaráljuk, a megírandó függvényünk típusát: a függvényünk egy `Bitseq -> Bool` függvényből egy `Bool`-t (logikai értéket) számol ki -}

A kód eleje

Minden Haskell program „`module` *Név* `where`”-rel kezdődik

```
module Main where
type Natural = Integer
type Bitseq = Natural -> Bool
```

Definiáljuk a „`#`” operátort, ami egy logikai értéket egy Bitseq elejére illeszt

```
(#) :: Bool -> Bitseq -> Bitseq
x # a = \i -> if i==0 then x else a (i-1)
```

A visszaper a „λ” ASCII alakja

Az if-then-else olyan, mint a C/C++-os `?:` operátor

```
forsome :: (Bitseq -> Bool) -> Bool
{- ezt a függvényt akarjuk megírni -}
```

forsome definíciója

```
module Main where
type Natural = Integer
type Bitseq = Natural -> Bool
(#) :: Bool -> Bitseq -> Bitseq
x # a = \i -> if i==0 then x else a (i-1)

forsome :: (Bitseq -> Bool) -> Bool
forsome p = p (find p)

find :: (Bitseq -> Bool) -> Bitseq
```

{- elegendő írnunk egy olyan `find` függvényt, ami megpróbál találni egy a paraméterül kapott tulajdonságot kielégítő bitsorozatot, és ha talál olyat, akkor visszaadja azt, egyébként pedig visszaad bármi mást -}

`find` definíciója

```
module Main where
type Natural = Integer
type Bitseq = Natural -> Bool
(#) :: Bool -> Bitseq -> Bitseq
x # a = \i -> if i==0 then x else a (i-1)

forsome :: (Bitseq -> Bool) -> Bool
forsome p = p (find p)

find :: (Bitseq -> Bool) -> Bitseq
find p = if forsome (\a -> p (False # a))
  then False # find (\a -> p (False # a))
  else True # find (\a -> p (True # a))
```

Megjegyzés: Lusta kiértékelés

- Nem kellően lusta szemlélő számára úgy tűnhet, hogy a `find` és a `forsome` függvények együtt végtelen ciklust csinálnak, de:
- A `find` függvény mindig meghívja a `forsome` függvényt.
- A `forsome` függvény viszont nem mindig hívja meg a `find` függvényt: ha a `p` függvénynek nincs szüksége a paraméterére (tehát az egy konstans függvény), akkor `forsome` nem fogja kiszámolni `(find p)`-t mert nincs szükség az értékére.

Miért működik ez?

- Tudjuk, hogy p-nek minden bitsorozatra véges sok időn belül le kell futnia.
- Ez azt jelenti, hogy minden a bitsorozathoz van egy N_a szám, úgy, hogy p az a sorozatot csak az 1, 2, …, N_a helyeken értékelte ki.
- Bebizonyítjuk, hogy ebből következik, hogy van egy olyan N szám, hogy $N_a = N$ minden a bitsorozatra.

A Cantor-halmaz

- $\mathcal{C}$ = Cantor-halmaz = [0,1] intervallumnak töröljük a középső harmadát (mint nyílt intervallumot), majd ezt ismételjük a végtelenségig a darabokra
- Egy (0 és 1 közti) szám pontosan akkor van a Cantor-halmazban, ha felírható hármas számrendszerben csak a 0 és 2 jegyek felhasználásával.
- Ez alapján a bitsorozatok azonosíthatók a Cantor-halmaz elemeivel (0 ↔ False, 2 ↔ True).

Folytonosság

- **Lemma:** Egy $p : \mathcal{C} \to \mathbb{R}$ függvény pontosan akkor folytonos x-ben, ha minden $\varepsilon>0$-hoz van olyan $N_{\varepsilon,x}$ pozitív egész, hogy ha x és y első $N_{\varepsilon,x}$ db hármas számrendszerbeli jegye megegyezik, akkor $|p(x)-p(y)|<\varepsilon$.
- Ezek szerint ha a `p :: Bitseq -> Bool` (tiszta, minden inputra véges idő alatt lefutó) függvényt egy $p : \mathcal{C} \to \{0,1\}$ függvénynek tekintjük, akkor az folytonos lesz egész $\mathcal{C}$-n.

Egyenletes folytonosság

- **Def.:** $f : A \to \mathbb{R}$ egyenletesen folytonos, ha minden $\varepsilon>0$-hoz van olyan $\delta_\varepsilon>0$, hogy minden $x,y \in A$-ra, ha $|x-y| < \delta_\varepsilon$, akkor $|f(x)-f(y)|<\varepsilon$.
- **Tétel:** Egy kompakt halmazon értelmezett folytonos függvény egyenletesen folytonos.
- A Cantor-halmaz közismert, hogy kompakt, de így a p-k egyenletesen folytonosak lesznek.
- Ezt visszaalakítva δ_ε-ról a megegyező jegyek N_ε számára éppen $\mathbb{N}$ létezését kapjuk (pl. $N_{1/2}$ jó lesz $\mathbb{N}$-nek).

Egyenletes folytonosság

- **Def.:** $f : A \to \mathbb{R}$ egyenletesen folytonos, ha minden $\varepsilon>0$-hoz van olyan $\delta_\varepsilon>0$, hogy minden $x,y \in A$-ra, ha $|x-y| < \delta_\varepsilon$, akkor $|f(x)-f(y)|<\varepsilon$.
- **Tétel:** Egy kompakt halmazon értelmezett folytonos függvény egyenletesen folytonos.
- A Cantor-halmaz közismert, hogy kompakt, de így a p-k egyenletesen folytonosak lesznek.
- Ezt visszaalakítva δ_ε-ról a megegyező jegyek N_ε számára éppen $\mathbb{N}$ létezését kapjuk (pl. $N_{1/2}$ jó lesz $\mathbb{N}$-nek).

Miért jó ez nekünk?

- Kaptuk, hogy minden `p :: Bitseq -> Bool` „szép” függvényhez létezik olyan $N=N_p$ pozitív egész, amire a `p` függvény csak az első N_p helyen értékeli ki a paraméterként kapott bitsorozatot.
- A `p' = (\a -> p (False # a))` és `p'' = (\a -> p (True # a))` függvényekre $N_{p'}=N_{p''}=N_p-1$ lesz.
- N_p szerinti indukcióval bizonyítható, hogy `forsome` és `find` véges időn belül lefut.

További függvények

```
{- igaz-e a tulajdonság minden bitsorozatra? -}
forevery :: (Bitseq -> Bool) -> Bool
forevery p = not (forsome (not . p))
{- próbáljunk találni jó bitsorozatot, vagy jelezzük, hogy nincs -}
search :: (Bitseq -> Bool) -> Maybe Bitseq
search p = if forsome p
  then Just (find p)
  else Nothing
{- az ilyen típusú függvények egyenlőségét el tudjuk dönteni -}
instance (Eq a) => Eq ((Bitseq->Bool)->a) where
  f == g = forevery (\a -> f a == g a)
```

A `Maybe Bitseq` típus lehetséges értékei: `Nothing` és `Just x`, ahol `x` egy bitsorozat

N kiszámítása

{- két bitsorozat azonos-e az első n biten -}

```
eqstart :: Natural -> Bitseq -> Bitseq -> Bool
eqstart n a b = and [a i == b i | i <- [0..n-1]]
```

Az `and` függvény a kapott lista elemeit össze és-eli.

{- logikai implikáció operátora -}

```
(-->) :: Bool -> Bool -> Bool
p --> q = not p || q
```

{- ez a függvény kiszámolja az `N` értéket a paraméterként kapott `p`-hez, vegyük észre, hogy gyakorlatilag a matematikai definíciót írtuk le, és ez működő Haskell kód lett -}

```
used_bits :: (Bitseq -> Bool) -> Natural
used_bits p = head [n | n <- [0..],
   forevery (\a ->
   forevery (\b ->
   (eqstart n a b --> p a == p b)))]
```

Vesszük azon `n`-ek (végtelen) listáját, ahol `n` természetes szám és igaz a belőle kapott logikai érték, majd ennek a `head` beépített függvénnyel vesszük a legelső elemét.

Egy hatékonyabb megvalósítás I.

```
module Main where
type Natural = Integer
type Bitseq = Natural -> Bool

branch :: Bool -> Bitseq -> Bitseq -> Bitseq
branch x left right n
  | n == 0     = x
  | odd n      = left ((n-1) `div` 2)
  | otherwise = right ((n-2) `div` 2)

findBit :: (Bool -> Bool) -> Bool
findBit p = if p False then False else True
```

Egy hatékonyabb megvalósítás II.

```
forsome :: (Bitseq -> Bool) -> Bool
forsome p = p (find p)   {- ez változatlan -}

find :: (Bitseq -> Bool) -> Bitseq
find p = branch x0 l0 r0 where
  x0 = findBit (\x ->
       forsome (\l ->
       forsome (\r -> p (branch x l r))))
  l0 = find (\l ->
       forsome (\r -> p (branch x0 l r)))
  r0 = find (\r -> p (branch x0 l0 r))
```

Futtatás

```
GHCi, version 7.4.1: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( 2.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +s
*Main> forsome (\a -> True)
True
(0.00 secs, 552736 bytes)
*Main> forsome (\a -> a 999)
True
(0.01 secs, 513840 bytes)
*Main> forsome (\a -> and [a i == odd i | i <-[1..20]])
True
(0.35 secs, 39473776 bytes)
```

A GHCi (a GHC fordítóhoz tartozó interaktív parancsértelmező) ezután kiírja a futásidőt és a memóriahasználatot (az adatok csak nagyságrendi jellegűek, egymás után többszöri futtatásra is előfordult eltérő eredmény).

Az első és huszadik bitek között pontosan a páratlan indexűek legyenek igazak

Kicsit nagyobb számok…

```
*Main> forsome (\a -> a (2014^2014))
True
(0.56 secs, 164318536 bytes)
*Main> forsome (\a -> a (2014^2014) && not (a
(2015^2015)))
True
(1.78 secs, 490614680 bytes)
*Main> forsome (\a -> a (2014^2014) && not (a (2015^2015))
&& a (2016^2016))
True
(4.46 secs, 1294872760 bytes)
*Main> forsome (\a -> a (2^2^2^2^2))
True
(3.16 secs, 1202031344 bytes)
```

Ez hatványozás,
nem bitenkénti XOR!

Alternatív feladatértelmezés

Mi van, ha a `p` függvényről nem követeljük meg azt, hogy minden bitsorozatra véges időn belül lefusson, csak azt, hogy minden kiszámítható bitsorozatra véges időben lefusson? Igaz-e, hogy a `forsome` függvény akkor is mindig véges idő alatt lefut, ha egy ezt a gyengébb feltételt kielégítő `p`-t adunk neki paraméterként?

(Egy függvényt akkor nevezünk kiszámíthatónak, ha van olyan Turing-gép, ami éppen azt a függvényt számolja ki, természetesen `p`-ről is feltesszük, hogy kiszámítható.)

Haskell propaganda

A Haskell nyelv következő tulajdonságai miatt tudtuk ennyire tömören megírni a programunkat:

- Képesek vagyunk „függvényeket” dinamikusan létrehozni lambda-kifejezésként (ez modern C++-ban is megjelent)
- Lusta kiértékelés van – ezt lehet utánozni lambdákkal, de az eléggé csúnya lesz
- Nagy számok kezelése nagyon könnyű
- Matematikaközeli jelölések

Forrás, további olvasnivaló

Az előadás a következő anyag alapján készült:

http://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/

Ennek a folytatása:

http://math.andrej.com/2008/11/21/a-haskell-monad-for-infinite-search-in-finite-time/

Ebben le van írva egy olyan `S` monád konstrukciója, amire `S a` lehetséges értékei az `a` típus értékeinek olyan részhalmazainak felel meg, amikben „lehet egy tulajdonság kielégíthetőségét tesztelni”. Az ott definiált műveletekkel könnyen kijön egy `S [Bool]` típusú objektum, ami az összes bitsorozat (mint végtelen lista) halmazának felel meg.