Strumienie, czyli pochwała lenistwa

Marcin BENKE
Instytut Informatyki,
Wydział Matematyki, Informatyki i Mechaniki,
Uniwersytet Warszawski

Hasło jego brzmiało: „Poznanie nieskończoności wymaga nieskończonego czasu. Toteż wszystko jedno czy się pracuje, czy nie”.

Arkadiusz i Borys Strugaccy
Poniedziałek zaczyna się w sobotę

„Tato, jak policzyć do nieskończoności?” – takie pytanie zadał mi niedawno mój czteroletni syn. Odpowiedziałem coś w stylu „nigdy nie skończysz”. W każdym razie liczenie do nieskończoności może się relatywnie szybko znudzić, może więc to zadanie, jak wiele innych monotonnych zadań, powierzyć komputerowi? A czy komputer potrafi policzyć do nieskończoności? Łatwo sprawić, aby komputer wypisywał coraz większe liczby – może nie w nieskończoność, ale tak długo, aż przestanie działać albo znudzi nam się ta zabawa.

```
> [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,...
3647506,3647507,3647508,3647509,^CInterrupted.
```

(wydawane polecenia poprzedzane są w tym artykule znakiem „>”; posługujemy się tu językiem Haskell, głównie jego interaktywnym interpreterem `ghci`).

Może lepiej zadać jednak inne pytanie: czy komputer potrafi policzyć do nieskończoności, a potem jeszcze coś zrobić? Na przykład: stworzyć listę wszystkich liczb naturalnych, a potem wybrać z niej liczby parzyste (albo pierwsze)?

Otóż potrafi, a metodą, która pozwala na obliczenia na obiektach nieskończonych (przy czym doskonale sprawdza się też dla skończonych), jest lenistwo: obliczenia leniwe są wykonywane dopiero wtedy, gdy potrzebne są ich wyniki:

```
> let nats = [0..]
```

Oto poleciliśmy stworzyć listę wszystkich liczb naturalnych i nazwać ją `nats`; komputer „wykonał” to zadanie błyskawicznie, jako że nie musiał ich wszystkich wypisywać. Niech więc wypisze choć kilka:

```
> let few = take 5
> few nats
[0,1,2,3,4]
```

Teraz wybierzmy tylko parzyste:

```
> let evens = [x | x <- nats, even x]
> few evens
[0,2,4,6,8]
```

Haskell pozwala na budowanie list za pomocą notacji podobnej do tej, której w matematyce używa się dla zbiorów. Zbiór liczb parzystych możemy zapisać jako

$$\{x \mid x \in \mathbb{N},\ x \equiv 0 \pmod 2\}$$

albo

$$\{2x \mid x \in \mathbb{N}\}.$$

Podobnie możemy zdefiniować strumień liczb parzystych w Haskellu:

```
> [x | x <- nats, x `mod` 2 == 0]
```

Analogiczny efekt można uzyskać za pomocą funkcji *filter*, która wybiera z listy elementy spełniające podany warunek, oraz predykatu *even*, spełnianego przez liczby parzyste:

```
> let evens = filter even nats
```

Może jeszcze, żeby sprawdzić, czy lista *evens* naprawdę zawiera więcej niż pięć elementów, zażyczmy sobie wypisania jej dziesięciomilionowego elementu:

```
> evens !! (10^7)
20000000
```

Za pierwszym razem to może chwilkę potrwać, bo – jak pamiętamy – obliczenia w naszym systemie są leniwe i obliczenie dziesięciu milionów elementów strumienia wykona się właśnie teraz. System pamięta jednak raz obliczone wartości, więc za drugim razem ten sam wynik dostaniemy już błyskawicznie.

```
> evens !! (10^7)
20000000
(1.12 secs, 360844064 bytes)
> evens !! (10^7)
20000000
(0.06 secs, 521804 bytes)
```

Nieskończone listy zwykle nazywa się *strumieniami*. Co możemy jeszcze z nimi zrobić?

Możemy, na przykład, zastosować jakąś transformację do każdego elementu listy.

```
> let odds = map (+1) evens
> few odds
[1,3,5,7,9]
```

Funkcja *map* daje listę będącą wynikiem zastosowania funkcji przekazanej jako pierwszy argument do każdego elementu listy będącej drugim argumentem. Funkcja (+1), jak łatwo się domyślić, daje w wyniku swój argument plus jeden.

Listy definiujemy zwykle w terminach głowy (pierwszy element listy) i ogona (cała reszta). Listę złożoną z głowy x i ogona xs zapisujemy jako $x : xs$. W językach funkcyjnych definicje zapisujemy w postaci ciągów równań (często rekurencyjnych). W ten sposób możemy zapisać definicję strumienia liczb naturalnych:

```
> let nats = from 0 where
  from n = n:from (n+1)
```

Strumienie możemy łączyć, np.:

```
> let zipped = zip evens odds
> few zipped
[(0,1),(2,3),(4,5),(6,7),(8,9)]
```

Funkcja *zip* niejako „spina” dwie listy, łącząc ich elementy w pary (podobnie do działania zamka błyskawicznego – stąd jej nazwa):

```
> let zip (a:as) (b:bs) = (a,b) : zip as bs
```

Podobnie możemy zdefiniować funkcję, która doda dwa strumienie, element po elemencie:

```
> let add (a:as) (b:bs) = (a+b) : add as bs
> few (add evens odds)
[1,5,9,13,17]
```

Do tej pory „nieskończoności” nie bardzo było widać, większość powyższych operacji działała w rzeczywistości na kilkuelementowych listach. Zbiory i listy nieskończone mają natomiast pewne ciekawe własności, np. lista nieskończona jest równoliczna ze swoim ogonem i możemy je połączyć:

```
> let odds = add nats (tail nats)
> few odds
[1,3,5,7,9]
```

Tym sposobem możemy uzyskać nowe definicje `nats` oraz `evens`:

```
> let nats = 0:map (+1) nats
> let evens = add nats nats
```

Dla listy liczb naturalnych głową jest 0, ogon zaś jest taki jak cała lista, „podwyższona” o 1.

Analogicznie możemy zdefiniować strumień liczb Fibonacciego: zachodzi równość

$$f + (tail\ f) = drop\ 2\ f,$$

gdzie *drop* n f pomija pierwszych n elementów listy

```
> let fibs = 0:1:add fibs (tail fibs)
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
> fibs !! 200
280571172992510140037611932413038677189525
```

Funkcja *drop* jest niejako dualna do funkcji *take*, która wybiera z listy pierwszych n elementów, pomijając resztę; użyte wcześniej *few* zdefiniowałem na potrzeby tego artykułu jako *take* 5.

Pierwsze dwa elementy to 0 i 1, po czym następuje suma ciągu ze swoim ogonem.

Tak na marginesie, tę definicję zapisuje się zwykle nie za pomocą *add*, ale wbudowanej funkcji *zipWith* (i taką definicję można najczęściej spotkać):

```
> let fibs = 0:1:zipWith (+) fibs (tail fibs)
```

Funkcja *zipWith* łączy listy za pomocą wskazanej funkcji

```
> let zipWith f (a:as) (b:bs) =
  f a b : zipWith f as bs
```

Stąd *zipWith* (+) oznacza po prostu sumę dwóch list.

Liczby pierwsze budzą zwykle więcej emocji niż liczby Fibonacciego; tak jest i w środowisku funkcyjnym – napisano na ten temat wiele artykułów. Tu ograniczymy się do kilku prostych sposobów.

Najpierw zdefiniujmy relację „x nie jest podzielne przez y”:

```
> let x -/ y = x `mod` y > 0
```

Pierwsza próba może wyglądać tak:

- pierwszym elementem listy jest 2,
- dalej z ciągu liczb naturalnych większych niż 2 wybieramy liczby, które nie są podzielne przez żadną z wcześniejszych.

```
> let primes1 = sieve [2..] where
  sieve (p:xs) = p : sieve [x | x <- xs, x -/ p]

> primes1 !! 10000
104743
(6.29 secs, 4296827576 bytes)
```

Można próbować usprawnić tę definicję, zauważając, że powyżej 2 możemy się ograniczyć do liczb nieparzystych:

```
> let primes2 = 2:sieve [3,5..] where
  sieve (p:xs) = p : sieve [x | x <- xs, x -/ p]
```

... ale okazuje się, że usprawnienie jest mniejsze, niż by się można spodziewać:

```
> primes2 !! 10000
104743
(6.20 secs, 4286105612 bytes)
```

Rzeczywiste usprawnienie możemy uzyskać, zauważając, że dla sprawdzenia pierwszości x wystarczy sprawdzać dzielniki nie większe niż $\sqrt{x}$:

```
> let primes3 = 2:[x | x <- [3,5..], isPrime x] where
  isPrime x = all (x -/) (factorsToTry x) where
  factorsToTry x = takeWhile (\p -> p*p <= x) primes3

> primes3 !! 10000
104743
(0.04 secs, 24043288 bytes)
```

Definicja *primes3* jest nieco bardziej złożona i zapewne wymaga kilku wyjaśnień. Użyta w definicji *isPrime* funkcja *all* sprawdza, czy podany predykat (tu „x nie jest podzielne przez...”) jest spełniony przez wszystkie elementy listy. Zatem liczbę x uznajemy za pierwszą, gdy nie dzieli się ona przez żaden z potencjalnych dzielników. Listę tych ostatnich uzyskujemy zaś, biorąc (*takeWhile*) kolejne liczby pierwsze tak długo, jak ich kwadraty są mniejsze od x.

W tym miejscu kuszące mogłoby być użycie funkcji *filter*, ale, niestety, taki sposób wybierania podzbiorów wymaga nieskończonego czasu, nawet gdy rzeczony podzbiór jest skończony (skąd mamy wiedzieć, czy może jeszcze gdzieś tam daleko w ciągu pojawi się element spełniający warunek). Zauważmy jednak, że wszystkie

nasze dotychczasowe strumienie są rosnące, dzięki czemu podzbiór ograniczony możemy zawsze wybrać w ograniczonym czasie.

Ogólniej, strumienie rosnące nadają się do reprezentacji nieskończonych zbiorów liczb: w ograniczonym czasie potrafimy rozstrzygnąć, czy dana liczba x jest elementem zbioru:

```
> let member x (y:ys)
  | x == y = True
  | x < y = False
  | x > y = member x ys

> member 104743 primes3
True
```

Przeglądamy kolejne elementy strumienia; gdy napotkamy liczbę większą od x, to wiemy, że x już dalej się nie pojawi (strumień jest rosnący, zatem wszystkie dalsze elementy są większe od x).

Podobnie możemy zdefiniować sumę (a także przecięcie i różnicę) zbiorów:

```
> let union (x:xs) (y:ys)
  | x < y = x:union xs (y:ys)
  | x == y = x:union xs ys
  | x > y = y:union (x:xs) ys
```

Możemy pójść o krok dalej i rozważać strumienie wyższego rzędu, czyli nieskończone strumienie nieskończonych strumieni. Spróbujmy, na przykład, stworzyć rosnący strumień liczb postaci p^n, gdzie p jest liczbą pierwszą. Gdybyśmy potrafili obliczyć strumień liczb stanowiący sumę teoriomnogościową strumienia strumieni, rozwiązanie mogłoby być proste:

```
> let primePowers = mergeAll [powers p | p <- primes]
```

gdzie *powers p* jest strumieniem kolejnych potęg p, zaś *mergeAll* jest funkcją, która scala strumień strumieni. Tylko czy potrafimy ją zdefiniować? Okazuje się, że tak (zainteresowanych odsyłam do modułu **Data.List.Ordered**; zob. `http://hackage.haskell.org/packages/archive/data-ordlist/0.4.5/doc/html/Data-List-Ordered.html`), jednakże pod pewnymi warunkami:

1. strumień głów strumieni składowych musi być niemalejący (w naszym przypadku jest to strumień liczb pierwszych);
2. każdy element wyniku będzie powtórzony tyle razy, ile razy łącznie powtarza się we wszystkich strumieniach wejściowych; funkcja *mergeAll* realizuje sumę multizbiorów, zaś sumę zbiorów, gdy strumienie wejściowe są parami rozłączne, który to przypadek także zachodzi dla naszego problemu.

```
> take 20 primePowers
[2,3,4,5,7,8,9,11,13,16,17,19,
                        23,25,27,29,31,32,37,41]
(0.00 secs, 527380 bytes)
```

W pół drogi do nieskończoności

Maciej LISICKI
doktorant, Instytut Fizyki Teoretycznej,
Wydział Fizyki, Uniwersytet Warszawski

Chmura składa się z obłoków złożonych z obłoków,
które składają się obłoków,
które wyglądają jak chmury.
Ale kiedy zbliżasz się do chmury,
nie widzisz gładkości,
tylko nieregularności w drobniejszej skali.
Benoît Mandelbrot (1924–2010)

W termodynamice charakteryzujemy równowagowy układ za pomocą pewnego zbioru parametrów (np. ciśnienia, objętości i temperatury) i związków pomiędzy nimi (np. równania stanu gazu). Wielkości te są funkcjami stanu, a więc są jednoznacznie przyporządkowane danemu stanowi układu i nie zmieniają się w czasie. Jeśli jednak spojrzymy na układ z punktu widzenia jego struktury mikroskopowej, na cząsteczki gazu w ciągłym ruchu, powstaje pytanie: jak powiązać charakterystyki termodynamiczne układu z jego mikroskopową dynamiką? Odpowiedzi na to pytanie udziela mechanika statystyczna, w myśl której układ wielu cząsteczek podlega prawom statystycznym, a parametry makroskopowe są średnimi wartościami odpowiednich wyrażeń mikroskopowych. Przykładowo, średnia energia kinetyczna cząsteczek gazu doskonałego, w którym prędkości cząsteczek opisane są rozkładem Maxwella, jest