Потоци. Работа с крайни и безкрайни потоци в езика Racket

Дефиниция на потоците и основни процедури за работа с потоци

Дефиниция на понятието „поток“

Потокът е редица от елементи, която може да се дефинира с помощта на основните примитивни процедури за работа с потоци както следва: ако **x** има стойност, равна на оценката на **(stream-cons a b)**, то **(stream-first x) ⟶ [a]** и **(stream-rest x) ⟶ [b]**.

Поток без елементи се означава с ***empty-stream***, а примитивен предикат за проверка за празен поток (поток без елементи) е ***stream-empty?***.

Следователно, потоците са съставен тип данни, чийто конструктор е примитивната процедура ***stream-cons*** и чийто селектори са примитивните процедури ***stream-first*** (която извлича първия елемент на даден поток) и ***stream-rest*** (която извлича опашката на даден поток, т.е. връща поток, получен от дадения поток чрез премахване на първия му елемент).

Една от важните характерни черти на потоците е, че при тях се предполага строго последователен достъп до елементите (в посока от началото на потока към неговия край). Както ще стане ясно по-нататък, потоците са структури от данни, чиято употреба е полезна най-вече в случаите, когато те имат голям брой (и дори безкрайно много) елементи.

Дефиниции на процедури за работа с потоци

Обща идея. По аналогия със списъците ще дефинираме някои процедури (включително такива от по-висок ред) за работа с потоци.

Обединяване на два потока (аналог на ***append*** с два аргумента при списъци)

```
(define (stream-append stream1 stream2)
   (if (stream-empty? stream1)
       stream2
       (stream-cons (stream-first stream1)
                    (stream-append (stream-rest
                                       stream1)
                                   stream2))))
```

Трансформиране на поток (аналог на ***map*** при списъци:
прилага дадена процедура към всеки от елементите
на даден поток и формира поток от съответните резултати)

```
(define (stream-map procedure stream)
   (if (stream-empty? stream)
       empty-stream
       (stream-cons (procedure (stream-first
                                  stream))
                    (stream-map procedure
                                (stream-rest
                                  stream)))))
```

Примери за употреба на дефинираните процедури

```
(define str1
        (stream-cons 1
           (stream-cons 2
              (stream-cons 3
                 empty-stream))))
(stream-map (lambda (x) (* 2 x)) str1)
```

В резултат се получава поток с елементи 2, 4, 6 (но не разполагаме с примитивна процедура, позволяваща извеждане отведнъж на всички елементи на потока).

Филтриране (по зададена процедура - филтър и поток връща нов поток, съставен от тези елементи на изходния, които преминават през филтъра)

```
(define (stream-filter proc stream)
   (cond [(stream-empty? stream) empty-stream]
         [(proc (stream-first stream))
          (stream-cons (stream-first stream)
                       (stream-filter proc
                          (stream-rest stream)))]
         [else (stream-filter proc (stream-rest
                                    stream))]))
```

Формиране на поток от числата, които се намират между две дадени числа **low** и **high**

```
(define (stream-enum low high)
   (if (> low high)
       empty-stream
       (stream-cons low (stream-enum (+ low 1)
                                     high))))
```

Реализация на потоците в езика Racket

Една възможност за реализация на потоците е свързана с използването на списъци (с представянето им във вид на списъци). Ако се избере този подход, тогава

stream-first съвпада с ***car***, ***stream-rest*** съвпада с ***cdr***,
stream-cons съвпада с ***cons***,
empty-stream съвпада с празния списък **() или *null***,
stream-empty? съвпада с ***null?*** .

Оказва се обаче, че списъците не са подходяща (ефективна) реализация на потоците.

Пример. Да се дефинира процедура, която връща като резултат второто просто число в интервала [10000, 1000000].

Примерно решение, което използва дефинираните по-горе процедури:

```
(define (second-prime)
   (stream-first (stream-rest (stream-filter
                        prime?
                        (stream-enum 10000 1000000)))))
```

Анализ на предложеното решение. В резултат на оценяването на обръщението към **stream-enum** (при реализация с използване на списъци) ще се формира списък от почти 1000000 елемента, като при това от този списък ще се използват само първите му десетина елемента (останалите почти 1000000 елемента изобщо няма да се използват). Разбира се, предложеният подход за решаване на тази задача далеч не е най-добрият, но той е подбран не само за да се покаже, че списъците не са подходящо средство за реализация на потоците. Идеята тук е, че много често програмистът не знае предварително каква част от дадена структура е действително необходима за решаването на конкретната задача. В такива случаи оценяването (формирането) на цялата структура може да се окаже излишно и дори крайно неефективно (когато тази структура има голям брой елементи).

Основната идея на действителната реализация на потоците, която ги прави подходящи структури за представяне на редици, съдържащи много голям брой елементи, дори безкрайно много елементи, е следната. Реализацията на потоците се основава на т. нар. ***забавени изчисления*** (***отложено оценяване***, ***delayed evaluation***), които са средство за „пакетиране“ на съответните изрази (аргументи и резултати) и позволяват тези изрази да бъдат оценявани едва тогава, когато оценяването им е наистина необходимо.

В този смисъл основната идея при реализацията на ***stream-cons*** е тази процедура да конструира потока само частично и да предава така конструирания от нея специален обект на програмата (процедурата), която го използва. Ако тази програма се опита да получи достъп до неконструираната част, то се конструира допълнително само тази част от потока, която е реално необходима за продължаване на процеса на оценяване. Оценяването (конструирането) на останалата част от потока отново се отлага до евентуално възникване на необходимост от достъп до нови елементи и т.н.

За целта се използва специалната форма ***delay***. В резултат на оценяването на обръщението **(delay <израз>)** се получава „пакетиран“ („отложен“) вариант на **<израз>**, който позволява реалното оценяване на **<израз>** да се извърши едва тогава, когато това е абсолютно необходимо. По-точно, „пакетираният“ израз, който е оценка на обръщението към ***delay***, може да бъде използван за възобновяване на оценяването на **<израз>** с помощта на процедурата ***force***.

Процедурите ***delay*** и ***force*** са вградени (примитивни). Тяхното действие може да се разглежда още и по следния начин. Формата ***delay*** „пакетира“ даден израз така, че той да бъде оценен при повикване. Следователно, може да се смята, че ***delay*** е такава специална форма, която *не оценява* аргумента си и за която

(delay <израз>) ⟶ (lambda () <израз>).

Обратно, ***force*** просто извиква процедурата без аргументи, получена от ***delay***. Следователно, ***force*** може да се реализира като процедура:

```
(define (force delayed-object)
   (delayed-object))
```

При тези дефиниции е в сила

(force (delay expr)) ⟶ (force (lambda () expr))
⟶ ((lambda () expr)) ⟶ [expr].

Тогава ***stream-cons*** може да се разглежда като специална форма (която не оценява втория си аргумент), дефинирана така, че

(stream-cons a b) ⟶ (cons a (delay b)).

Селекторите ***stream-first*** и ***stream-rest*** могат да бъдат дефинирани като процедури както следва:

```
(define (stream-first stream)
   (car stream))
(define (stream-rest stream)
   (force (cdr stream)))
```

Тази реализация на потоците вече е достатъчно ефективна в смисъла, в който проблемът беше обсъждан по-горе.

Например, ако се върнем на разгледания пример за намиране на второто просто число в интервала [10000, 1000000], процесът на оценка на израза

```
(stream-first (stream-rest (stream-filter
                prime?
                (stream-enum 10000 1000000))))
```

изглежда по следния начин.

Оценяването на **`(stream-enum 10000 1000000)`** води до оценяване на съответното обръщение към ***stream-cons***, чиято оценка е

```
(cons 10000 (delay (stream-enum 10001 1000000))).
```

Следователно, получава се поток, представен от точковата двойка с първи елемент 10000 и втори елемент - "пакетиран" израз, оценяването на който може да бъде извършено при необходимост. Този поток се филтрира от **stream-filter** (с помощта на **prime?**), като за целта се проверява дали неговият първи елемент (числото 10000) е просто число. Това в случая не е така, поради което се поражда (предизвиква) оценяване на обръщението към **stream-filter** с аргумент – опашката (***stream-rest***) на същия поток. Обръщението към ***stream-rest*** форсира оценяването на „пакетирания“ („отложения“) израз, върнат от **stream-enum**, в резултат на което се получава **(cons 10001 (delay (stream-enum 10002 1000000)))**. Тук **stream-filter** отново установява, че ***car*** (***stream-first***) от този израз (числото 10001 = 73*137) не е просто число и отново се обръща към ***stream-rest***, като го форсира да продължи оценяването и т.н.

Този процес продължава до получаване на първото просто число в разглеждания интервал (числото 10007), при което **stream-filter** съгласно дефиницията си връща

```
(stream-cons (stream-first stream)
                  (stream-filter proc
                     (stream-rest stream))).
```

При това в конкретния случай горният израз има вида

```
(cons 10007
      (delay
         (stream-filter prime?
            (cons 10008
               (delay
                  (stream-enum 10009 1000000))))))
```

Този резултат се предава като аргумент на ***stream-rest*** - а в основния израз. Сега този ***stream-rest*** форсира отложеното оценяване на **stream-filter**, който от своя страна форсира **stream-enum**, докато бъде намерено следващото просто число.

Тогава като аргумент на ***stream-first*** - а в основния израз се предава

```
(cons 10009
      (delay (stream-filter
                 prime?
                 (cons 10010
                       (delay
                          (stream-enum 10011
                                       1000000))))))
```

и крайният резултат е 10009.

Обща бележка. По този начин действително се получи максимална ефективност, тъй като не се оцени нищо, което не е необходимо за формирането на крайния резултат. При това, моделът на оценяване чрез заместване е валиден и в този случай.

В средата на DrRacket разгледаните процедури за работа с потоци са достъпни като части от библиотеките **racket/stream** (основните примитивни процедури за работа с потоци) и **racket/promise** (процедурите ***delay*** и ***force***).

Примери

```
(require racket/stream)
(define a (stream-cons 'a
            (stream-cons 'b empty-stream)))
> a
#<stream>
> (stream-first a)
'a
> (stream-first (stream-rest a))
'b
> (stream-rest (stream-rest a))
#<stream>
```

Работа с безкрайни потоци

Основни идеи

Демонстрираната по-горе техника на реализация на потоците действително позволява те да бъдат използвани като ефективни структури за представяне на редици, които могат да имат голям брой елементи и дори безброй много елементи.

Използват се два основни подхода при конструирането на безкрайни потоци – т. нар. ***неявно (индиректно)*** и ***явно (директно)*** конструиране. При първия подход се използват специални процедури - генератори, а при втория - рекурсия върху съответния генериран поток (по отношение на вече генерираните части на потока).

Неявно (индиректно) конструиране на безкрайни потоци

Пример 1. Генериране на безкраен поток от положителните цели числа.

```
(define (integers-from n)
   (stream-cons n (integers-from (+ 1 n))))
(define integers (integers-from 1))
```

Пример 2. Генериране на безкраен поток от числата на Фибоначи.

```
(define (fibgen a b)
   (stream-cons a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))
```

Обяснение на решението от Пример 1. Предложената дефиниция е смислена, тъй като **integers** има за стойност точкова двойка с глава - числото 1 и опашка – „пакетиран“ израз, който може да генерира поток от целите числа, започващи от 2. Така генерираният поток е безкраен, но във всеки момент може да се работи само с негова крайна част. В този смисъл нашата програма никога няма да „знае“, че целият безкраен поток не е наличен (тъй като никога няма да се наложи да бъде разглеждан целият поток).

Дефиниции на някои полезни функции
за работа с (безкрайни) потоци

Намиране на n-тия елемент на безкраен поток

```
(define (nth-of-stream n stream)
   (if (= n 1)
       (stream-first stream)
       (nth-of-stream (- n 1)
                      (stream-rest stream))))
```

Събиране на елементите на два потока

```
(define (add-streams s1 s2)
   (cond [(stream-empty? s1) s2]
         [(stream-empty? s2) s1]
         [else (stream-cons
                  (+ (stream-first s1)
                     (stream-first s2))
                  (add-streams (stream-rest s1)
                               (stream-rest s2)))]))
```

Забележка. Ако потоците **s1** и **s2** са безкрайни, проверките на двете гранични условия не са необходими (не са необходими проверките в първите две клаузи на ***cond***-а).

Умножаване на всички елементи на даден поток с константа

```
(define (scale-stream const stream)
   (stream-map (lambda (x) (* x const))
               stream))
```

Сливане на два сортирани потока в резултантен сортиран поток
(дублиращите се елементи се включват в резултата
в един екземпляр)

```
(define (merge-streams s1 s2)
   (cond [(stream-empty? s1) s2]
         [(stream-empty? s2) s1]
         [else (let ([h1 (stream-first s1)]
                     [h2 (stream-first s2)])
```

```
(cond [(< h1 h2)
       (stream-cons h1
          (merge-streams
             (stream-rest s1) s2))]
      [(> h1 h2)
       (stream-cons h2
          (merge-streams s1
             (stream-rest s2)))]
      [else (stream-cons h1
             (merge-streams
              (stream-rest s1)
               (stream-rest s2)))]))]))
```

Явно (директно) конструиране на безкрайни потоци

Обща идея. По-горе генерирахме потоци с помощта на специални процедури - генератори. Възможен е и друг, директен вариант на дефиниране на потоци, при който се използват предимствата, които дава отложеното оценяване (използва се рекурсия върху самия генериран поток, а не се използват процедури - генератори).

Пример 1. Дефиниране на безкраен поток от единици (чрез рекурсия по отношение на дефинирания поток от единици).

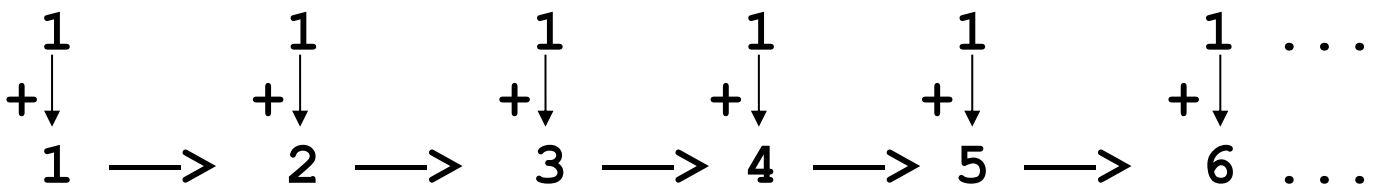
```
(define ones (stream-cons 1 ones))
```

Така **ones** се дефинира като точкова двойка, чийто ***car*** е 1 и чийто ***cdr*** е „пакетиран“ израз, при форсирането на който ще се получи **[ones]**. Оценяването на ***cdr*** от тази точкова двойка отново дава 1 и „пакетиран“ израз от описания вид и т.н. С други думи, вече генерираната част от потока се използва за дефиниране на следващите му елементи.

Пример 2. Генериране на безкраен поток от целите положителни числа.

Идея. Всеки елемент на потока (от втория включително нататък) е равен на предишния, увеличен с 1.

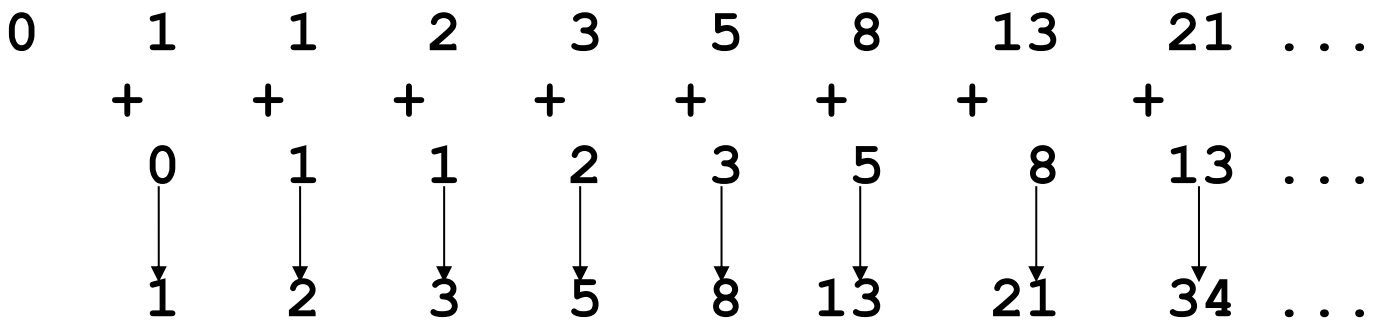
```
(define ints
   (stream-cons 1
                (add-streams ones ints)))
```



Пример 3. Генериране на безкраен поток от числата на Фибоначи.

Идея. Всеки елемент на потока (от третия включително нататък) е сума на предишните два.

```
(define fibs
   (stream-cons 0
      (stream-cons 1
         (add-streams (stream-rest fibs) fibs))))
```



Допълнителни бележки върху реализацията на процедурите delay и force в езика Racket

По-горе отбелязахме, че ***delay*** е специална форма, която не оценява аргумента си и за която

(delay <израз>) ⟶ (lambda () <израз>).

„Обратната“ ѝ функция ***force*** може да се дефинира по следния начин:

```
(define (force delayed-object)
   (delayed-object))
```

Тази реализация на ***delay*** и ***force*** е задоволителна и работи коректно, но може да бъде оптимизирана. В много случаи се налага един и същ отложен обект да се форсира повече от един път. Това може да доведе до голяма неефективност например при рекурсивни програми, които работят с потоци. Едно възможно решение е съответният отложен обект да се построи така, че в резултат на първото прилагане на ***force*** да се съхрани получената при форсирането стойност. При по-късни прилагания на ***force*** просто ще се връща съхранената стойност, без да се повтарят всички извършени изчисления. Следователно, новата идея е да се реализира ***delay*** като процедура със специална памет, предназначена за съхраняване на посочените стойности. Един от начините да се осъществи тази идея е да се използва дефинираната по-долу процедура **memo-proc**, която има като аргумент процедура без аргументи и връща като резултат съответната процедура с памет.

При първото си стартиране тази процедура запомня намерения резултат и при следващите извиквания просто го връща:

```
(define (memo-proc proc)
   (let ([already-run? #f] [result '()])
      (lambda ()
         (if (not already-run?)
             (begin (set! result (proc))
                    (set! already-run? #t)
                    result)
             result))))
```

Тогава ***delay*** е специална форма, която се дефинира така, че **`(delay <израз>)`** е еквивалентно на:

`(memo-proc (lambda () <израз>))`,

а дефиницията на ***force*** може да остане непроменена.

Това съответства в максимална степен на действителната реализация на процедурите ***delay*** и ***force*** в Racket.