PROGRAMANDO EN HASKELL

Capítulo 15 – Evaluación Postergada

Introducción

Las expresiones en Haskell se evalúan utilizando una técnica llamada evaluación postergada (lazy evaluation), la cual:

- Evita hacer evaluaciones innecesarias;
- Asegura la terminación cuando es posible;
- Soporta la programación con listas infinitas;
- Permite a los programas ser más modulares.

Evaluando Expresiones

```
square n = n * n
```

Ejemplo:

```
  square (1+2)
=
  square 3
=
  3 * 3
=
  9
```

También es posible otro orden de evaluación:

```
  square (1+2)
= (1+2) * (1+2)
= 3 * (1+2)
= 3 * 3
= 9
```

primero aplica square.

Cualquier forma de evaluar la misma expresión dará el mismo resultado, siempre y cuando termine.

Estrategias de Evaluación

Hay dos estrategias principales para decidir que expresión reducible (redex) considerar a continuación:

- Elegir el redex íntimo, en el sentido que no contiene otro redex;

- Elegir el redex del extremo, en el sentido que no está contenido en otro redex.

Terminación

```
infinity = 1 + infinity
```

Ejemplo:

```
fst (0, infinity)
```

Evaluación del redex íntimo.

=

```
fst (0, 1 + infinity)
```

=

```
fst (0, 1 + (1 + infinity))
```

=

⋮

```
fst (0, infinity)
```

=

```
0
```

Evaluación del redex extremo.

Nota:

- La evaluación del extremo puede dar un resultado cuando la evaluación del íntimo falla en terminar;

- Si cualquier secuencia de evaluación termina, también lo hará la evaluación del extremo, con el mismo resultado.

Número de Reducciones

Íntimo:

```
  square (1+2)
=
  square 3
=
  3 * 3
=
  9
```

3 pasos.

Extremo:

```
  square (1+2)
=
  (1+2) * (1+2)
=
  3 * (1+2)
=
  3 * 3
=
  9
```

4 pasos.

Nota:

- La versión del extremo es ineficiente, porque el argumento 1+2 se duplica cuando square se aplica, siendo evaluado dos veces.

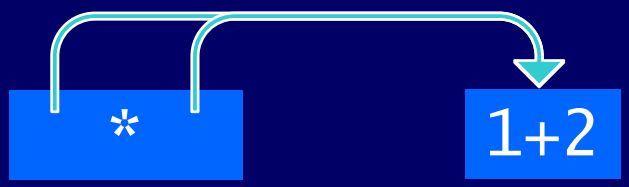
- Debido a tal duplicación, la evaluación del extremo podría requerir más pasos que la evaluación del íntimo.

- Podemos evitar este problema usando punteros para indicar el compartimiento de argumentos.

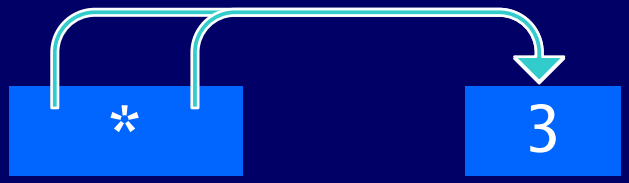
Ejemplo:

```
square (1+2)
```

=



=



=

```
9
```

El argumento compartido evaluado una sola vez.

Esto nos da una nueva estrategia de evaluación:

evaluación postergada = evaluación del extremo + compartir argumentos

Nota:

- La evaluación postergada asegura la terminación cuando es posible, pero nunca requiere más pasos que la evaluación del íntimo y algunas veces requiere menos pasos.

Listas Infinitas

```
ones = 1 : ones
```

Ejemplo:

```
  ones
= 1 : ones
= 1 : (1 : ones)
= 1 : (1 : (1 : ones))
= ⋮
```

Una lista infinita de unos.

¿Qué pasa si seleccionamos el primer elemento?

Íntimo:

```
  head ones
=
  head (1:ones)
=
  head (1:(1:ones))
=
  ⋮
```

No termina.

Postergada:

```
  head ones
=
  head (1:ones)
=
  1
```

¡Termina en 2 pasos!

Nota:

- En la evaluación postergada, solo el primer elemento de la lista es producido, porque no se requiere el resto.

- En general, con la evaluación postergada las expresiones solo se evalúan conforme son requeridas por el contexto en el que están siendo utilizadas.

- Así, ones es potencialmente una lista infinita en realidad.

Programación Modular

La evaluación postergada nos permite escribir programas mas modulares al separar el control de los datos

```
> take 5 ones
[1,1,1,1,1]
```

La parte de los datos ones solo se evalúa conforme lo va requiriendo la parte de control take 5.

Sin la evaluación postergada las partes de control y de los datos necesitarían ser combinadas en una sola:

```
replicate :: Int → a → [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

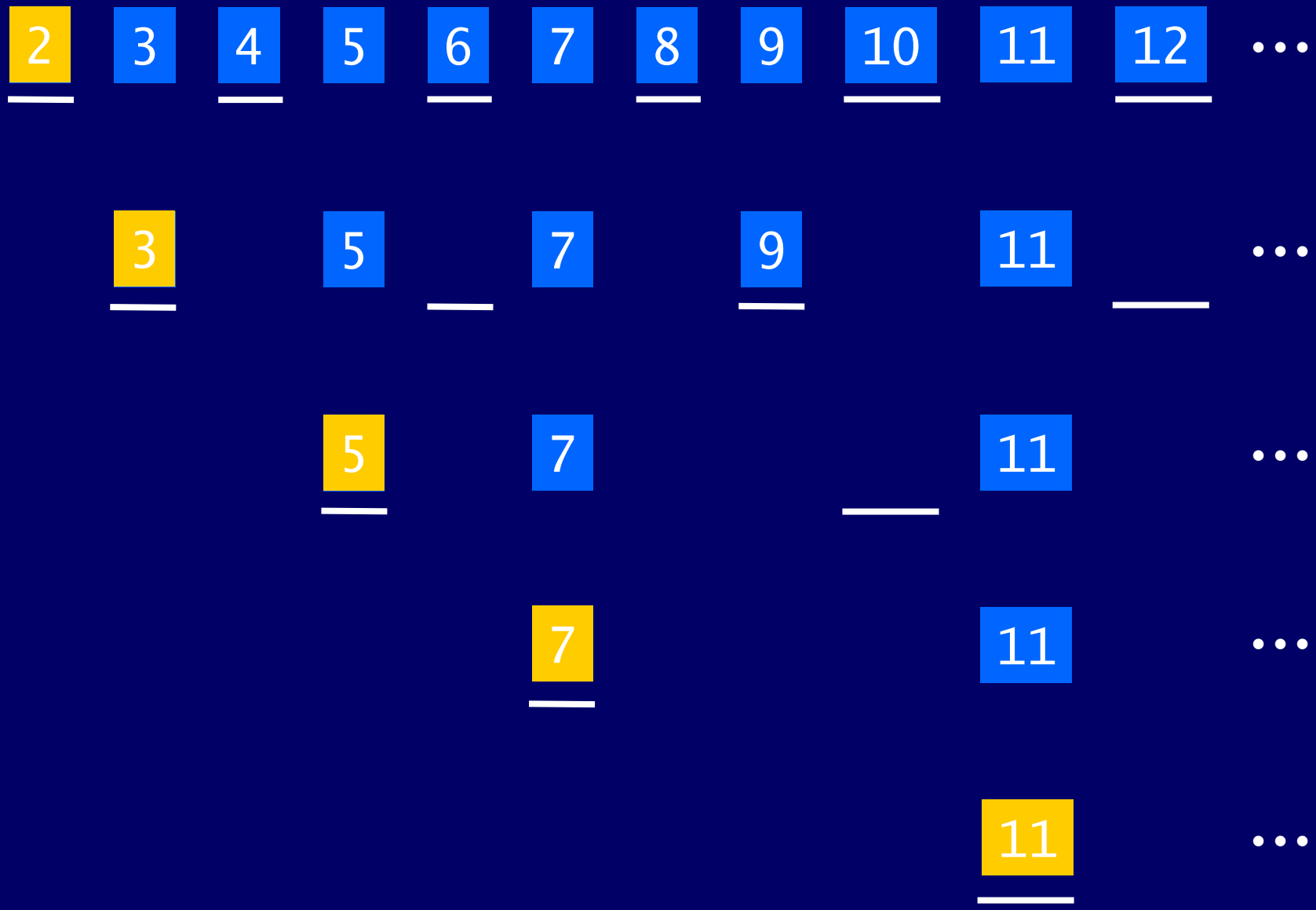
Ejemplo:

```
> replicate 5 1
[1,1,1,1,1]
```

Generando Primos

Para generar una secuencia infinita de primos:

1. Escribe la secuencia infinita 2, 3, 4, …;
2. Marca el primer número p como primo;
3. Borra todos los múltiplos de p en la secuencia;
4. Regresa al Segundo paso.


2 3 4 5 6 7 8 9 10 11 12 ...
3 5 7 9 11 ...
5 7 11 ...
7 11 ...
11 ...

¡Esta idea se traduce directamente al programa que genera una lista infinita de números primos!

```
primes :: [Int]
primes = sieve [2..]
```

```
sieve :: [Int] → [Int]
sieve (p:xs) =
    p : sieve [x | x ← xs, mod x p /= 0]
```

Ejemplos:

```
> primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,…
```

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

```
> takeWhile (< 10) primes
[2,3,5,7]
```

Podemos usar primes para generar una lista (¿infinita?) de primos gemelos que difieren precisamente por dos.

```
twin :: (Int,Int) → Bool
twin (x,y) = y == x+2
```

```
twins :: [(Int,Int)]
twins = filter twin (zip primes (tail primes))
```

```
> twins
[(3,5),(5,7),(11,13),(17,19),(29,31),…
```

Ejercicio

(1) La secuencia Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

comienza con 0 y 1, cada número siguiente es la suma de los dos previos. Usando una lista comprendida, define una expresión

```
fibs :: [Integer]
```

que genera esta secuencia infinita.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com