Laboratorio de Lenguajes de Programación

Evaluación perezosa Parte I

Manuel Soto Romero

Universidad Nacional Autónoma de México
Facultad de Ciencias

5 de octubre de 2016

Objetivos

Objetivos

- Definir el concepto de evaluación perezosa y sus diferencias entre la semántica de la evaluación glotona.

Objetivos

- Definir el concepto de evaluación perezosa y sus diferencias entre la semántica de la evaluación glotona.
- Definir el concepto de sucesión.

Objetivos

- Definir el concepto de evaluación perezosa y sus diferencias entre la semántica de la evaluación glotona.
- Definir el concepto de sucesión.
- Implementar una sucesión usando las ventajas de la evaluación perezosa.

Definición

Definición

La evaluación perezosa se define como:

Definición

La evaluación perezosa se define como:

«Una estrategia de evaluación dónde los parámetros en una llamada de un procedimiento no son evaluados hasta que sean usados en el cuerpo del procedimiento».

(Felleinsen et al., 2009)

Definición

La evaluación perezosa se define como:

«Una estrategia de evaluación dónde los parámetros en una llamada de un procedimiento no son evaluados hasta que sean usados en el cuerpo del procedimiento».

(Felleinsen et al., 2009)

En pocas palabras... Evaluamos hasta que sea estrictamente necesario

Ejemplo

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (! (* n n)))
```

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (! (* n n)))
```

No ocurre ningún error

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (! (* n n)))
```

No ocurre ningún error

Si usamos la variante `plai` con evaluación glotona con la misma expresión:

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (! (* n n)))
```

No ocurre ningún error

Si usamos la variante `plai` con evaluación glotona con la misma expresión:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (* n n))
```

Ejemplo

Racket tiene una variante que trabaja con evaluación perezosa. Podemos escribir las siguiente expresión usando esta variante, con `!` forzamos la evaluación:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (! (* n n)))
```

No ocurre ningún error

Si usamos la variante `plai` con evaluación glotona con la misma expresión:

```
(let ([n 5] [nunca (factorial 1000000000)])
   (* n n))
```

Se consume demasiada memoria

Sucesiones

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Para representar una lista usaremos sucesiones (*streams*).

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Para representar una lista usaremos sucesiones (*streams*).

Una sucesión es una lista con valores que siguen un patrón específico. Por ejemplo:

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Para representar una lista usaremos sucesiones (*streams*).

Una sucesión es una lista con valores que siguen un patrón específico. Por ejemplo:

```
'(1 3 7 11 13)
```

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Para representar una lista usaremos sucesiones (*streams*).

Una sucesión es una lista con valores que siguen un patrón específico. Por ejemplo:

```
'(1 3 7 11 13)
```

Cada valor de la sucesión es un número impar. Para encontrar cada elemento n en la sucesión debemos seguir el patrón: $2n + 1$

Sucesiones

Analizaremos ahora, un ejemplo que ilustra cómo, mediante evaluación perezosa, podemos implementar listas posiblemente infinitas.

Para representar una lista usaremos sucesiones (*streams*).

Una sucesión es una lista con valores que siguen un patrón específico. Por ejemplo:

```
'(1 3 7 11 13)
```

Cada valor de la sucesión es un número impar. Para encontrar cada elemento n en la sucesión debemos seguir el patrón: $2n + 1$

Si el conjunto de elementos que forma una sucesión no tiene último elemento decimos que es infinita: `'(0 1 2 3 4 5 ...)`.

Procesadores

Procesadores

Queremos definir una sucesión mediante evaluación perezosa.

Procesadores

Queremos definir una sucesión mediante evaluación perezosa.

Podemos simular este tipo de evaluación usando procesadores (*thunks*).

Procesadores

Queremos definir una sucesión mediante evaluación perezosa.

Podemos simular este tipo de evaluación usando procesadores (*thunks*).

Un procesador es una función que no recibe parámetros, es decir tiene la siguiente forma:

Procesadores

Queremos definir una sucesión mediante evaluación perezosa.

Podemos simular este tipo de evaluación usando procesadores (*thunks*).

Un procesador es una función que no recibe parámetros, es decir tiene la siguiente forma:

(λ () <cuerpo>)

Procesadores

Queremos definir una sucesión mediante evaluación perezosa.

Podemos simular este tipo de evaluación usando procesadores (*thunks*).

Un procesador es una función que no recibe parámetros, es decir tiene la siguiente forma:

(λ () <cuerpo>)

Podemos simular este tipo de evaluación usando procesadores (*thunks*).

Listas infinitas

Listas infinitas

Si recordamos, una lista se define de forma recursiva como sigue:

Listas infinitas

Si recordamos, una lista se define de forma recursiva como sigue:

- La lista vacía es una lista y se representa por `'()`.
- Si `x` es un elemento de un conjunto cualquier y `xs` es una lista, entonces `(cons x xs)` es una lista. Llamamos a `x` la cabeza de la lista y a `xs` el resto de la lista.
- Son todas.

Listas infinitas

Si recordamos, una lista se define de forma recursiva como sigue:

- La lista vacía es una lista y se representa por `'()`.
- Si `x` es un elemento de un conjunto cualquier y `xs` es una lista, entonces `(cons x xs)` es una lista. Llamamos a `x` la cabeza de la lista y a `xs` el resto de la lista.
- Son todas.

El comportamiento perezoso se logra, postergando la creación o evaluación del resto de la sucesión usando procesadores. El procesador actuará como el patrón que genera cada elemento de la sucesión y lo hará hasta que sea estrictamente necesario.

Listas infinitas

Si recordamos, una lista se define de forma recursiva como sigue:

- La lista vacía es una lista y se representa por `'()`.
- Si `x` es un elemento de un conjunto cualquier y `xs` es una lista, entonces `(cons x xs)` es una lista. Llamamos a `x` la cabeza de la lista y a `xs` el resto de la lista.
- Son todas.

El comportamiento perezoso se logra, postergando la creación o evaluación del resto de la sucesión usando procesadores. El procesador actuará como el patrón que genera cada elemento de la sucesión y lo hará hasta que sea estrictamente necesario.

```
(define-type Stream
   [sempty]
   [scons (head any?) (tail zero-arity?)])
```

Ejemplos

Ejemplos

Función que genera una sucesión de números naturales desde n:

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

```
(define (genera-factoriales n)
   (scons (fact n) (λ () (genera-naturales (+ n 1)))))
```

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

```
(define (genera-factoriales n)
   (scons (fact n) (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de fibonaccies desde n:

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

```
(define (genera-factoriales n)
   (scons (fact n) (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de fibonaccies desde n:

```
(define (genera-fibonaccies n)
   (scons (fibo n) (λ () (genera-naturales (+ n 1)))))
```

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

```
(define (genera-factoriales n)
   (scons (fact n) (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de fibonaccies desde n:

```
(define (genera-fibonaccies n)
   (scons (fibo n) (λ () (genera-naturales (+ n 1)))))
```

Ninguna de estas funciones recursivas tiene un caso base.

Ejemplos

Función que genera una sucesión de números naturales desde n:

```
(define (genera-naturales n)
   (scons n (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de factoriales desde n:

```
(define (genera-factoriales n)
   (scons (fact n) (λ () (genera-naturales (+ n 1)))))
```

Función que genera una sucesión de fibonaccies desde n:

```
(define (genera-fibonaccies n)
   (scons (fibo n) (λ () (genera-naturales (+ n 1)))))
```

Ninguna de estas funciones recursivas tiene un caso base.

Sólo estamos definiendo el patrón en esta etapa.

Operaciones básicas de cabeza y cola

Operaciones básicas de cabeza y cola

Encontrar el primer elemento consiste simplemente en regresar el parámetro `head` del constructor:

Operaciones básicas de cabeza y cola

Encontrar el primer elemento consiste simplemente en regresar el parámetro `head` del constructor:

```
(define (shead s)
   (if (sempty? s)
      (error 'shead "empty stream")
      (scons-head s)))
```

Operaciones básicas de cabeza y cola

Encontrar el primer elemento consiste simplemente en regresar el parámetro `head` del constructor:

```
(define (shead s)
   (if (sempty? s)
      (error 'shead "empty stream")
      (scons-head s)))
```

Calcular el resto de la sucesión no consiste sólo de regresar el parámetro `tail`. Encontrar el resto de la sucesión consiste en aplicar la función que regresa el procesador, dicha aplicación genera otra sucesión que representa el resto de la original.

Operaciones básicas de cabeza y cola

Encontrar el primer elemento consiste simplemente en regresar el parámetro `head` del constructor:

```
(define (shead s)
   (if (sempty? s)
      (error 'shead "empty stream")
      (scons-head s)))
```

Calcular el resto de la sucesión no consiste sólo de regresar el parámetro `tail`. Encontrar el resto de la sucesión consiste en aplicar la función que regresa el procesador, dicha aplicación genera otra sucesión que representa el resto de la original.

```
(define (stail s)
   (match s
      [(sempty) (error 'stail "empty stream")]
      [(scons h t) (t)]))
```

Otras funciones útiles

Otras funciones útiles

Función para encontrar en n-ésimo elemento de la sucesión:

Otras funciones útiles

Función para encontrar en n-ésimo elemento de la sucesión:

```
(define (snth s n)
   (if (= n 0)
      (shead s)
      (snth (stail s) (- n 1))))
```

Otras funciones útiles

Función para encontrar en n-ésimo elemento de la sucesión:

```
(define (snth s n)
   (if (= n 0)
      (shead s)
      (snth (stail s) (- n 1))))
```

La siguiente función es la más importante, pues genera los primeros n elementos de la sucesión. Para generar los elementos y poder visualizarlos usamos una lista:

Otras funciones útiles

Función para encontrar en n-ésimo elemento de la sucesión:

```
(define (snth s n)
   (if (= n 0)
      (shead s)
      (snth (stail s) (- n 1))))
```

La siguiente función es la más importante, pues genera los primeros n elementos de la sucesión. Para generar los elementos y poder visualizarlos usamos una lista:

```
(define (stake s n)
   (if (<= n 0)
      empty
      (match s
         [(sempty) empty]
         [(scons h t) (cons h (stake (t) (- n 1)))])))
```

Ejercicios

Los ejercicios se encuentran en
`sites.ciencias.unam.mx/ldp171/practicas`

Enviar al correo `manu+ldp@ciencias.unam.mx` **un archivo** `equipo_sesion8.rkt` **con asunto** `[LDP-Sesión 8]` **a más tardar a las 18:59:59.**

Incluir el nombre de los integrantes en el cuerpo del correo.

Sólo pueden entregar aquellos alumnos que aparezcan en la lista de asistencia de la sesión. No es válido apuntar a miembros del equipo que no estén presentes.