CS 360
Programming Languages
Day 15 – Streams II

Review

- A ***thunk*** is a function of no arguments used to explicitly delay a computation.
  - No special syntax, not specific to Racket.

- A ***promise*** is a data type that holds a thunk and also caches the result of the computation.
  - Not specific to Racket.
  - **`(delay`** ***`expr`*** **`)`** => returns promise for ***`expr`***
  - has to be implemented as a special form so that ***`expr`*** won't be evaluated until we **`force`** it.
  - Once forced, later forces won't re-evaluate ***`expr`***, but rather the same value will be returned for every subsequent **`force`**.
  - **`(force promise)`** => returns the value of the original ***`expr`***, either by evaluating it, or by retrieving the cached value.

Example

```
(define x 1)
(define y (delay x))
(force y)
(set! x 2)
(force y)
```

Streams

- One common use for promises is to create a new data type called a ***stream***.
- Streams and lists are almost identical in functionality and implementation.
  - Only difference is the car of a stream is eager (evaluated normally), but the cdr is lazy (implemented as a promise).
  - (Car and cdr of normal lists are eager.)

- Create a stream with **`stream-cons`**:

```
(define-syntax-rule (stream-cons first rest)
  (cons first (delay rest)))
```

- This code creates a special form that literally replaces every call to stream-cons with the line (cons <first arg> (delay <2nd arg>)).
- A normal function wouldn't work because it would evaluate both arguments, but we want to delay evaluation of the rest argument.

Useful stream functions

Most of these are just the list functions we know and love with the prefix "**`stream-`**"

List version	**Stream version**
`'()`	`'()`
`null?`	`stream-null?`
`car`	`stream-car`
`cdr`	`stream-cdr`
	`stream->list`
`list-ref`	`stream-ref`
	`stream-enumerate`

Finite Streams

- Not any more useful than lists.

  -
    ```
    (stream-cons 1
     (stream-cons 2
      (stream-cons 3 '())))
    ```

- The power of streams comes from making infinite streams.
  - Impossible to do with lists.
  - Easy with streams because we don't explicitly represent all the values (since there are an infinite number of them).
  - Instead, we represent the first one explicitly, and then promise to provide the next one as soon as it's needed.

Our first infinite stream

- Let's create an infinite stream of a fixed constant value. What would that look like as cons cells?

- How could we write a function that takes one argument (the fixed value) and returns an infinitely long stream of that value?

-
  ```
  (define (make-constant-stream val)
    (stream-cons val (make-constant-stream val))
  ```

- A different way:

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

Another infinite stream

- Let's create an infinite stream of integers increasing from a fixed starting integer. What would that look like as cons cells?

- How could we write a function that takes one argument (the fixed value) and returns an infinitely long stream of that value?

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

- Possible to create the stream '(1 2 3 ...)) in one line of code, but we need some more functions first.

Streams and higher-order functions

- Let's duplicate the map function to work with streams (finite or infinite).

- List version of map:

```
(define (map func lst)
  (if (null? lst) '()
    (cons (func (car lst)) (map func (cdr lst)))))
```

- Stream version:

-
  ```
  (define (stream-map func stream)
    (if (stream-null? stream) '()
      (stream-cons (func (stream-car stream))
                   (map func (stream-cdr stream)))))
  ```

Using stream-map

- If we already have

```
(define ints-from-1 (ints-from 1))
```

- How would we:
  - Define a stream of the multiples of 5?
  - Define a stream of the powers of 2?

- Define a function `stream-filter` that is analogous to filter.
  - Use stream-filter and ints-from-1 to make a stream of only even numbers.
- Define a new stream of integers increasing from 1 by using stream-map.
  - Do not use a function; do this (recursively) in one line.
- Define a function `stream-map2` that works like map2 on project 2 (takes a function of two args and two streams).
  - Define a new stream of ints increasing from 1 by using stream-map2 and a constant stream of 1s.
  - Do not use a function; do this recursively in one line.
- Define a function called **`partial-sums`** that takes a stream and returns the partial sums of the stream.
  - Ex: the partial sums of ints-from-1 are 1, 3, 6, 10, 15...
- Create a stream of the numbers '(4, -4/3, 4/5, -4/7, 4/9...) any way you want.
  - Hint: This will go faster if you use decimals rather than fractions.
  - Find the partial sums of the previous stream. What are they approaching?
- Define a function `not-divisible-by` that takes a stream of integers and an integer n and removes all the integers that are divisible by n from the stream.
- Define function that returns an infinite stream of prime numbers.
  - Hint: Recursively use `not-divisible-by` on a stream of the ints from 2.
- Define an infinite stream of the Fibonacci numbers.