Programming Languages

Streams and Memoization

Adapted from Dan Grossman's PL class,
U. of Washington

Review

- A thunk is a function of no arguments used to explicitly delay a result.
- **`(delay expression)`** => returns a thunked version of expression
  - has to be implemented as a special form so that **`expression`** won't be evaluated until we **`force`** it.
  - Once forced, later forces won't re-evaluate **`expression`**, but rather the same value will be returned for every subsequent force.
  - Called a promise. (in that we say delay returns a promise)
- **`(force promise)`** => returns the value of the original delayed expression, either by evaluating it, or saving 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.
- Stream == List
  - 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`
`'()`	`the-empty-stream`
`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-empty-stream)))
    ```
- 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.

Two common stream idioms

- Consider these two versions of an infinite stream of ones:

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

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

  ```
  (define ones-alt (make-constant-stream 1))
  ```

Next examples

- Create an infinite stream of integers, starting at zero and increasing by one.
  - Hint: define a function that takes an argument x and returns a stream of integers starting from x.
- Define a function `stream-map` that duplicates the functionality of map for streams.
- Define an alternate version of the infinite stream of integers starting from zero by using `stream-map` and an infinite stream of ones.
- Define a function `stream-filter` that duplicates filter.
- 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 an infinite stream of prime numbers.
  - Hint: use not-divisible-by on a stream of the ints from 2.
- Define an infinite stream of the Fibonacci numbers.