Module 4a: Self-Reference

CPSC 110

Peyton Seigo

2018-09-26

Module 4a: Self-Reference

Learning goals

- Be able to use list mechanisms to construct and destruct lists.
- Be able to identify problem domain information of arbitrary size that should be represented using lists and lists of structures.
- Be able to use the HtDD, HtDF and Data Driven Templates recipes with such data.
- Be able to explain what makes a self-referential data definition well formed and identify whether a particular self-referential data definition is well-formed.
- Be able to design functions that consume and produce lists and lists of structures.
- Be able to predict and identify the correspondence between self-references in a data definition and natural recursions in functions that operate on the data.

Notes

- Lists
  - A list is an itemization containing `atomic-distinct` and `compound` data.
  - Lists are **self referential**.
- The self-reference template rule puts a natural recursion in the template that corresponds to the self-reference in the type comment.
  - *The `dd-template-type` is called `self-ref`*
- `(check-expect)`s for lists
  - Examples shold include base and self-referential cases.
  - Have one or more tests with a list of 2 or more elements
- Remember, constants support "single-point of control" in that it's very easy to change their values later on!

Terminology

- **Arbitrary-sized information**: information that we don't know the size of in advance.
  - *A program that can display any number of cows is operating with abitrary-sized information.*
- **Self-reference**: the relationship between an itemization's case that refers to the data definition itself and the data definition
  - Causes Natural Recursion in template
- **Well-formed self referential data definition**:
  - At least one base case (allows self referential case to end)
  - At least one self referential case

- **Self reference rule**: if an itemization data definition has a `one-of` case that refers to itself, put a natural recursion in the template that corresponds to the self-reference in the type comment
  - `@dd-template-rules` rule is called `self-ref`
- **Natural recursion**: the relationship within a self-referencing data type's template where the template actually refers to itself!
  - i.e. a template's function named `fn-for-los` will call itself within its own definition using `(fn-for-los (rest los))`
  - Caused by self-reference rule

The `cons` primitive

The primitive `cons` is a two element constructor that constructs a list:

```
(cons x y) -> list?
  x : any/x
  y : list?
```

`cons` can be used to produce lists with more than one type of data; but we will not do that (our data definitions do not let us talk about that very well).

Other primitives that operate on lists

Lists have functions that are SIMILAR to `struct` selectors:

- `(first <list>)`: first element in list
- `(rest <list>)`: list with front popped off
  - *Note: `rest` expects a non-empty list*
  - `(first (rest L2))`: produces element in `<list>`
    - pops element off the front of `L2`, then gets the first element in the new list
    - `(second <list>)` also exists, but popping and getting the first element as shown above is VERY useful in things like recursion and using accumulators! It's mostly useful because the procedure is generalized.
- `(empty? <list>)`: produce true if argument is the empty list
- `(length <list>)`: evaluates number of items on a list

Lists containing primitive data

Example of a list data definition containing primitive data:

```
(require spd/tags)
```

```
(@HtDD ListOfNumber)
;; ListOfNumber is one-of:
;;  - empty
;;  - (cons Number ListOfNumber)
;; interp. a list of numbers
(define LON1 empty)
(define LON2 (cons 12 empty))
(define LON3 (cons 6 (cons 12 empty)))

(@dd-template-rules one-of          ; 2 cases
                    atomic-distinct ; empty
                    compound        ; (cons Number ListOfNumber)
                    self-ref)       ; (rest lon) is ListOfNumber
(define (fn-for-lon lon)
    (cond [(empty? lon) (...)]
          [else
            (... (first lon)
                 (fn-for-lon (rest lon)))]))
```

And a function implementing our new list:

```
(@HtDF sum)
(@signature ListOfNumber -> Number)
;; produce the sum of the given list
(check-expect (sum empty) 0)
(check-expect (sum (cons 5 empty)) 5)
(check-expect (sum (cons 10 (cons 5 empty))) 15)

;(define (sum lon) 0) ; stub

(@template ListOfNumber)
(define (sum lon)
    (cond [(empty? lon) 0]
          [else
            (+ (first lon)
               (sum (rest lon)))]))
```