Common Lisp and Introduction to Functional Programming
Lecture 8: Functional Data Structures 1/2

Yuri Zhykin

Apr 8, 2021

Avoiding Explicit Mutable State 1/2

- **Lexical bindings** allow to avoid assignment operations and provide a cleaner way of maintaining **local** state.
- **Recursion and TCO** allow to define iterative processes in a stateless way, replacing state with function arguments.
- Generally, in a lot of cases explicit state can be replaced with function arguments:
  - this way, functions remain mathematical functions, as they still depend solely on their arguments (the goal),
  - state has a clearly defined "flow" through the program and can be viewed as data.

Avoiding Explicit Mutable State 2/2

- Example:

```
(defun fib-1 (n)
  (if (or (= n 0) (= n 1))
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

(defun fib-2 (n)
  (let ((a 0)
        (b 1))
    (loop
      while (> n 0)
      do (multiple-value-setq (a b) (values b (+ a b)))
         (decf n))
    a))

(defun fib-3 (n)
  (labels ((%fib (n a b)
             (if (= n 0)
                 a
                 (%fib (- n 1) b (+ a b)))))
    (%fib n 0 1)))
```

Data Structures

- **Data structure** is
  - a collection of values,
  - the relationships among these values,
  - the functions or operations that can be applied to the data.
- Data structures store state in a **semi-implicit** way: it is not immediately obvious that a data structure contains components that are modified.

Mutable Data Structures

- Generally, most useful data structures available in modern programming languages are **mutable** for **efficiency reasons**.
- **Mutable data structure** is a data structure that can be modified in-place by its operations.
- Operations that modify the data structure in-place are called **destructive operations** - they “destroy” the existing data structure and replace it in memory with the new structure.

Immutable Data Structures 1/2

- **Immutable data structure** or **persistent data structure** is a data structure that preserves the previous version of itself when modified.
- **Mutable data structure operations** return only the relevant results, while all required state changes are performed in-place in the data structure object.
- **Immutable data structure operations** return both the relevant results and the new version of the data structure that contains the new version of internal data.
- The old version of the data structure remains intact for any parts of the program that hold the reference to it.

Immutable Data Structures 2/2

- Advantages:
  - no need to track which program components modify the data structure - the new versions of the structure are created by its operation, returned as results and propagated through the program,
  - programs that work with immutable data structures are trivially **parallelizable**.
- Drawbacks:
  - new version of the data structure is **a copy** of the previous object with some modifications, the old object still takes memory,
  - copying the data structure on every operations consumes a lot of memory,
  - this approach relies heavily on efficient **memory management** and **garbage collection**.

Copy-on-Write

- **Copy-on-write** (**CoW**) approach distinguishes between **read** and **write** operations and is based on the following idea
  - if a part of the structure is only being **read**, it can be shared between multiple readers,
  - if a part of the structure is modified, a new of the structure is created with all the parts **except** the one being **written** shared with the original structure,
  - the new copy of structure is than modified in-place with no effect on the original structure.

Structure Sharing

- **Structure sharing** is an approach to implementing immutable data structures that partially resolves the problem of memory consumption.
- **Structure sharing** is a variant of the **CoW**: we only need to copy the parts of the data structure that are being modified.
- If **structure sharing** is implemented for primitive data structures provided by the language, more complex data structures can be constructed from the primitive ones in a way that preserves structure sharing benefits.

Common Lisp's Lists 1/2

- **Common Lisp's list** is one of the simplest examples of structure sharing:

```
CL-USER> (setf list-1 (list 1 2 3))
(1 2 3)
CL-USER> (setf list-2 (cons 0 list-1))
(0 1 2 3)
CL-USER> list-1
(1 2 3)
CL-USER> (eq (cdr list-2) list-1)
T
CL-USER> (setf list-3 (list 'a (list 'b 'c)))
(4 5 6)
CL-USER> (setf list-4 (append list-3 list-1))
(a (b c) 1 2 4)
CL-USER> (eq (cdr (cdr (cdr list-4) list-1)))
T
```

Common Lisp's Lists 2/2

- Internal representation:

```
        +---+   +---+   +---+
list-3: | a |-->| * |-->|NIL|
        +---+   +--\+   +---+
                    \
                     \+---+   +---+   +---+
                      | b |-->| c |-->|NIL|
                     /+---+   +---+   +---+
                    /
        +---+   +--/+
list-4: | a |-->| * |
        +---+   +--\+
                    \
                     \+---+   +---+   +---+   +---+
list-1:               | 1 |-->| 2 |-->| 3 |-->|NIL|
                     /+---+   +---+   +---+   +---+
                    /
             +---+ /
list-2:      | 0 |/
             +---+
```

Building on Lists: 1/4

- Lists are great at representing sequences, but can we use them to represent associative maps (associative arrays or key-value structures)?
- **P-lists** (**plists**, **property lists**) are lists that store keys and values in a flat structure:

```
CL-USER> (setf plist '(:a 1 :b 2 :c 3))
(:a 1 :b 2 :c 3)
CL-USER> (getf plist :b)
2
CL-USER> (setf (getf plist :b) 22) ;; destructive
22
CL-USER> plist
(:a 1 :b 22 :c 3)
```

Building on Lists 2/4

- **A-lists (alists, associative lists)** are lists that store keys and values as `cons`-pairs:

```
CL-USER> (setf alist '((:a . 1) (:b . 2) (:c . 3)))
((:a . 1) (:b . 2) (:c . 3))
CL-USER> (assoc :b alist)
(:b . 2)
CL-USER> (setf (cdr (assoc :b alist)) 22) ;; destructive
22
CL-USER> alist
((:a . 1) (:b . 22) (:c . 3))
```

Building on Lists: 3/4

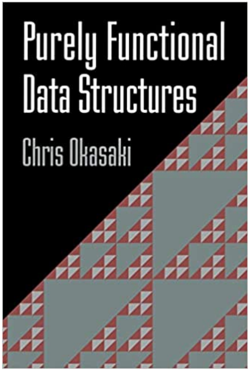
- **Associative lists** can be easily adapted to be functional data structures (i.e. update operations can be implemented in a non-destructive manner).
- **Problem with associative lists** is that they are very inefficient in general:
  - lookup operations take $O(n)$ time, where n is the number of elements in the list,
  - update operations take $0(n)$ time and $0(n)$ space in the worst case scenario,
  - for comparison, both lookup and update operations on a mutable `hash-table` take $O(1)$ time (update also takes $0(1)$ space on average).

Building on Lists: 4/4

- Can we have `hash-table`-like performance on an immutable data structure?
- The answer is **trees**.
- **Clojure** language heavily relies on structure-sharing trees implementation for most of its primitive data structures.
- In **Common Lisp**, structure-sharing trees can be easily implemented on top of built-in structure-sharing lists.

Useful Resources

- **Purely Functional Data Structures** by Chris Okasaki



The End

Thank you!