Functional Data Structures in Common Lisp: A Google Summer of Code Proposal

Andrew Baine
andrew.baine@gmail.com

March 26, 2007

1 Functional Programming

Functional programming is a programming paradigm based on composing, applying, and evaluating functions. Advantages of functional programs include the ease of reasoning about their behavior, the simplicity of parallel execution, and the availability of efficiencies such as lazy evaluation, call-by-future evaluation, and common subexpression. The critical feature of a functional program is an absence of side effects. A pure function may return new values but it must not modify its arguments. This constraint is in fact what frees functional programmers to take advantage of the above-mentioned efficiencies.

Common Lisp allows imperative programming, functional programming, object oriented programming, as well as virtually any paradigm of programming; consequently, side effects are not prohibited. Many of the built-in functions of Common Lisp are not pure functions – they modify their arguments. Common Lisp programmers thus tend to program in a style that is functional whenever possible, imperative or destructive when occasionally necessary. This Google Summer of Code proposal is about reducing the frequency of such occasions by providing a library of data structures that is entirely functional.

2 Data Structures and Functional Programming

One problem that functional programmers face is how to use conventional data structures without modifying them. For example, imperative programmers take for granted that a call such as `(heap-insert item heap)` may modify the given heap. This won't do for a functional programmer, who may rely on his ability to insert several different items into a heap whose state does not change.

So, a functional programmer may write:

```
(find-if #'(lambda (c)
             (problem-solved (heap-insert c heap)))
         '(c1 c2 c3))
```

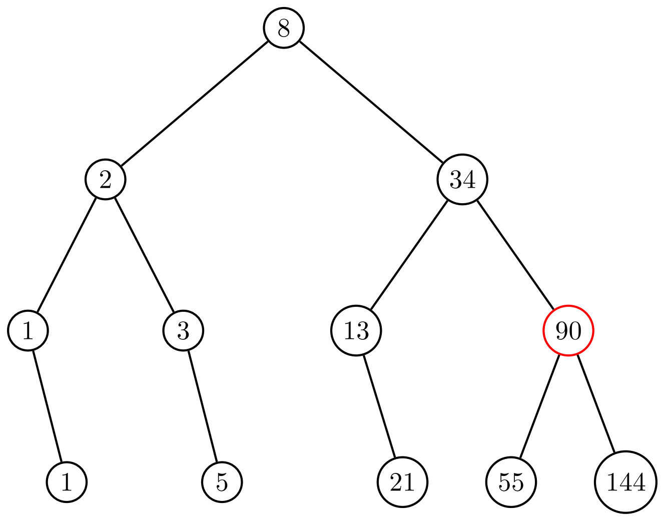
The programmer does not expect `c1` to be in the heap when he inserts `c2`. A conventional heap insertion algorithm, which modifies the heap, is unsatisfactory. What is the solution, then? Consider the following workaround:

```
procedure heap-insert (item heap)
    h := copy-heap (heap)
    conventional-heap-insert (item h)
    return h
```

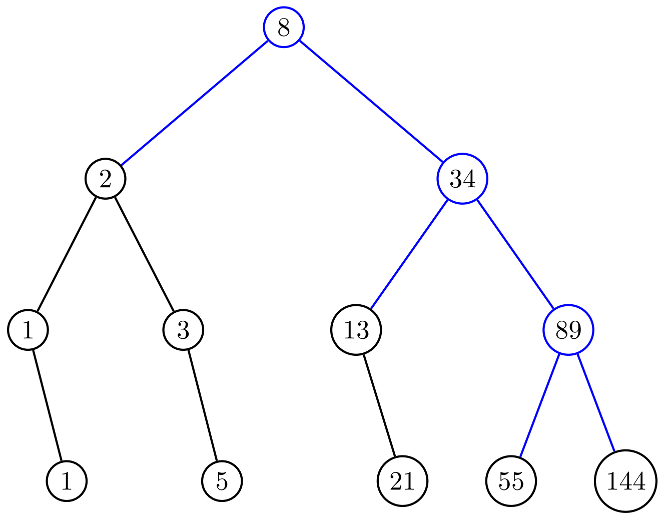
This would solve any issues of destructive modification, but it obviates the use of a heap, because copying the heap is an $O(n)$ operation in time and space. Thus the advantages of functional programming are associated with some cost. Functional programmers cannot use many conventional data structures.

3 Trees to the Rescue

The problem above stemmed from the task of copying the heap. This step took $O(n)$ time and space because each element had to be copied. If we stored the heap in a tree, though, only $log(n)$ elements would be copied. Consider the following binary tree, which is supposed to hold the first twelve fibonacci numbers.



How can we correct the tree without modifying it? We could naively copy every node in the tree, correcting the data in the one incorrect node in the new tree; but this requires copying all n nodes and is precisely the $O(n)$ operation we're trying to avoid. All we really need to change is the offending node and its ancestors, like this:



This operation satisfies the requirement of functional programming, that the original tree is unmodified, and it is also feasible for use because it takes only $O(log(n))$ time and space.

4 A Prototype: Fixed Length Array

To demonstrate the use of trees to implement data types, I present a simple implementation of a fixed length array-like data structure. I am interested in implementing a data structure supported by the following operations:

operation	arguments	return value
`make-f-array`	`size`	a new functional array
`f-array-elt`	`a index`	the item at a[index]
`f-array-replace`	`a index item`	an array b with b[index] = item

Using a tree structure to implement this type yields some gains and losses. The disadvantage is that querying the item at a given index takes $O(log(n))$ time instead of $O(1)$ time. But the advantage is that instead of $O(n)$ replacement, we have $O(log(n))$ replacement.

The source code for this example can be found in the Appendix. The following interaction shows how it works. Consider again the trees of fibonacci numbers presented in 3. We can create a functional array with these numbers as follows:

```
> (defvar f-array
          (make-f-array 12 :initial-contents (fibs 12)))
↦ F-ARRAY

> (defvar mistake (f-array-replace f-array 10 90))
↦ MISTAKE

> (defvar correction (f-array-replace mistake 10 89))
↦ CORRECTION

> (in-order-print mistake)
1 1 2 3 5 8 13 21 34 55 90 144
↦ NIL

> (in-order-print correction)
1 1 2 3 5 8 13 21 34 55 89 144
↦ NIL

> (show-difference mistake correction)
 index |  old  |  new
     5 |    8  |    8
     8 |   34  |   34
    10 |   90  |   89
↦ NIL
```

The last three interactions show the improvement due to the tree representation. The in-order printing of the two structures *after the correction* shows that `mistake` is unmodified. And most important, only three new nodes are created, as opposed to twelve. As is evident, of the 12 elements in `mistake` only 3 new nodes were created during the call to `f-array-replace`. The advantage of $O(log(n))$ over $O(n)$ only grows with n. In the following examle, only 7 new nodes are created when an element in array of size 1000 is replaced. This is less than 1% of those the space that would be allocated with a list or an array instead of a tree.

```
> (let ((a (make-f-array 1000)))
       (show-difference a (f-array-replace a 693 t)))
 index |  old  | new
   499 |  NIL  |  NIL
   749 |  NIL  |  NIL
   624 |  NIL  |  NIL
   686 |  NIL  |  NIL
   717 |  NIL  |  NIL
   701 |  NIL  |  NIL
   693 |  NIL  |  T
NIL
```

5 Lazy Evaluation?

The problem of data structures for functional programmers is well known and has been documented in Chris Okasaki's paper, "Purely Functional Data Structures." Okasaki shows how lazy evaluation can allow suitable functional data structures. This proposal, however, is not about implementing lazy-evaluation-style data structures. The structures that I code would be strict call-by-value structures. Their usefulness would be entirely due to the underlying trees that back them.

6 Proposal

I propose to implement a library of functional data structures that are suitable for use in computationally complex programs. The array example presented in this paper is but a toy compared to what I would like to produce. In particular, I would like to implement a a binary tree, a self-balancing tree, a fixed size list, an adjustable size list, a queue, a stack, a priority queue, and a dictionary. I find that these are the abstract data types that I rely on most in my day-to-day programming. Each of these will be backed by some sort of tree, so that insertion and deletion does not take $O(n)$ time. The nature of the simplified example of this proposal allowed the use of a binary tree that was not self-balancing, because the structure was of a fixed size. The complex data structures that I plan to implement will be of variable size, and will thus require balancing. Every function related to these data structures will be pure – the structures will never be modified and there will be no other side effects.

There are already a few open source Common Lisp data structure libraries. None is suitable for functional programming, because all rely on the modification of the structure to perform insertions and deletions. This project would fill a need in the Common Lisp community.

6.1 Methodology

Coding the example data structure for this proposal was a lot of fun, but it made me realize that there are some difficult issues to think about to make these data structures work. I would adhere to the following schedule for the summer:

Weeks 1–2 Determine precisely which abstract data types to implement. At a minimum, this will consist of a binary tree, a self-balancing tree, a fixed size list, an adjustable size list, a queue, a stack, a priority queue, and a dictionary. I will research implementations of these data structures in lisp, in other languages, and as abstract data types in computer science texts. After completing the research, I will specify the abstract interface to be implemented.

Weeks 3–4 Implement a prototype of a purely functional self-balancing tree. Since all the data structures will be backed by trees, this is probably the most important data structure of the project. By the end of this stage, I will have a preliminary version of the tree that satisfies the abstract interface.

Weeks 5–6 Using the tree from the previous stage, implement prototypes of the other data structures in the project. Refine the tree.

Weeks 7–8 Refine all data structures in the project. For each data structure, optimize the prototype, possibly by adding type specialization and other efficiencies.

Weeks 9–10 For each data structure, add additional functionality that Lisp programmers expect from data structures. For example, programmers are accustomed to calling function `map` on their `sequence` instances and function `mapcar` on their `list` instances. They may well expect to be able to call `map-queue` on a queue, in addition to the conventional queue API.

Weeks 11–12 Prepare documentation to demonstrate the library's use. I will be documenting the source code throughout the project, but in this stage, I will prepare a guide to using the data structures, and include examples like using the priority queue to solve the 15-puzzle and the functional array to solve a sudoku. At the end of this stage, I will deliver the source code and documentation in accordance with the Google Summer of Code protocol.

6.2 Conclusion

I am very excited about this project and hopeful that Google and my mentor will feel that it is worthwhile. I can reached by e-mail at `andrew.baine@gmail.com` to answer any questions about the project. I would also offer as references, the following three faculty members at George Washington University: Poorvi Vora `<poorvi@gwu.edu>`, Dave Christian `<dave_christian@prodigy.net>`, and Mark Happel `<mhappel@gwu.edu>`. These professors know my programming ability; they also know I love Common Lisp, because I've taken to turning in a version of my programming projects in Common Lisp in addition to whatever language is required. I am still waiting for somebody to assign a project and say, "You must complete this in Common Lisp." I hope Google will this summer.

Source Code

```
(defpackage :fds}
(in-package :fds)

(defstruct (bt (:print-function print-bt))
  'A binary tree.'
  index
  data
  left
  right)

(defun make-f-array (size &key (initial-contents nil))
  'A new functional array of the given size.'
  (labels ((f (start end)
             (cond ((= start (1- end))
                    (make-bt :index start
                             :data (when initial-contents
                                     (elt initial-contents start))
                             :left nil
                             :right nil))
                   ((= start end) nil)
                   (t (let ((midpoint (floor (+ end start -1) 2)))
                        (make-bt :index midpoint
                                 :data (when initial-contents
                                           (elt initial-contents midpoint))
                                 :left (f start midpoint)
                                 :right (f (1+ midpoint) end)))))))
    (f 0 size)))

(defun f-array-replace (f-array index data)
  'A functional array identical to the given array except that the
data at index is changed.  This is an O(log(n)) function in time and
space.'
  (cond ((< index (bt-index f-array))
         (make-bt :index (bt-index f-array)
                  :data (bt-data f-array)
                  :left (f-array-replace (bt-left f-array) index data)
                  :right (bt-right f-array)))
        ((= (bt-index f-array) index)
         (make-bt :index index
                  :data data
                  :left (bt-left f-array)
                  :right (bt-right f-array)))
        (t (make-bt :index (bt-index f-array)
                    :data (bt-data f-array)
                    :left (bt-left f-array)
                    :right (f-array-replace (bt-right f-array) index data)))))

(defun f-array-get (f-array index)
  'The data at the given index of the given functional array.'
  (cond ((= index (bt-index f-array)) (bt-data f-array))
        ((< index (bt-index f-array)) (f-array-get (bt-left f-array) index))
        (t (f-array-get (bt-right f-array) index))))

(defun f-array-position (item f-array)
  'The index at which the given item is found in the given functional
array, or nil if the item is not found.'
  (when f-array
    (or (f-array-position item (bt-left f-array))
        (when (eql item (bt-data f-array))
          (bt-index f-array))
        (f-array-position item (bt-right f-array)))))

(defun f-array-difference (old new)
  'A list of the data in nodes that are structurally different in the
given functional arrays old and new.'
  (when old
    (concatenate 'list
                 (when (not (eq old new))
                   (list (list (bt-index old) (bt-data old) (bt-data new))))
                 (f-array-difference (bt-left old) (bt-left new))
                 (f-array-difference (bt-right old) (bt-right new)))))

;;; SIDE EFFECTS (PRINTING)

(defun show-difference (old new &optional (depth 0))
  (when old
    (when (zerop depth)
        (format t "~&␣index␣|␣␣old␣␣|␣new"))
    (unless (eq old new)
      (format t "~&~6d␣|␣␣~3d␣␣|␣␣~3d" (bt-index old) (bt-data old)
              (bt-data new)))
    (show-difference (bt-left old)
                     (bt-left new) (1- depth))
    (show-difference (bt-right old)
                     (bt-right new) (1- depth))))

(defun print-bt (bt stream depth)
  (when (zerop depth)
        (format stream "#BT"))
  (when bt
      (if (> depth -2)
          (format stream "((~A,~A)~A~A)"
                  (bt-index bt)
                  (bt-data bt)
                  (format nil "~:[␣x~;␣~A~]"
                          (bt-left bt)
                          (print-bt (bt-left bt) nil (1- depth)))
                  (format nil "~:[␣x~;␣~A~]"
                          (bt-right bt)
                          (print-bt (bt-right bt) nil (1- depth))))
          (format stream "..."))))

(defun pre-order-print (bt)
  (walk-pre-order bt #'print-data))

(defun print-data (node)
  (format t "~A␣" (bt-data node)))

(defun in-order-print (bt)
  (walk-in-order bt #'print-data))

(defun walk-pre-order (bt f)
  'Apply function f to each node of the given binary tree in
pre-order fashion.'
  (when bt
    (funcall f bt)
    (walk-pre-order (bt-left bt) f)
    (walk-pre-order (bt-right bt) f)))

(defun walk-in-order (bt f)
  'Apply function f to each node of the given binary tree in
in-order fashion.'
```

```
  (when bt
    (walk-in-order (bt-left bt) f)
    (funcall f bt)
    (walk-in-order (bt-right bt) f)))

;;; This is just for the example

(defun fibs (n &optional (a 1) (b 1))
  ’A list of the first n numbers in the fibonacci sequence.’
  (if (zerop n)
      nil
      (cons a (fibs (1- n) b (+ a b)))))
```