Pennant Lines: An Alternative to Finger Trees in Dynamic Environments

Daniel Jay Haskin
dan@djhaskin.com

ABSTRACT

Pennant Lines are presented, a novel data structure for storing persistent sequences in Common Lisp suitable as a drop-in replacement for normal lists, but with much better runtimes for most operations. These sequences support at least logarithmic time when implementing a version of the Common Lisp Sequences API persistently, and better time in other key cases. All are simpler to implement within a dynamic setting such as that of Common Lisp than Finger Trees, and play nicer with the larger Lisp ecosystem.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**.

KEYWORDS

lisp, datastructure, sequences, dynamic, untyped

ACM Reference Format:
Daniel Jay Haskin. 2024. Pennant Lines: An Alternative to Finger Trees in Dynamic Environments. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MOTIVATION

When in a dynamically typed language like Lisp, it is often convenient to work with a data structure in specific ways in one moment, such as for operating on sets, and then perform simple list operations on them the next. Alists are nice because at the end of the day, "they are just lists" – they can be fed into functions that expect an array of items, like `print`. Object type is fluid in Lisp, as is object purpose. In Lisp, data structures can be used for their characteristics and ergonomics, rather than for their intended type. Lisp is a highly dynamic environment.

In the dynamic environment of Common Lisp, lists are used for *everything*, from dictionaries to sets to arrays. All the tooling works better with these data structures than with anything else. Getting the nth element, performing set union or intersection, stack operations, and syntax abstractions on lists are all defined in the Common Lisp standard.

However, lists present poor run time characteristics when used for anything more complicated than a stack. Often lists are used simply because of their ease of use with little regard for runtime. This is fine until the runtime becomes important; then ergonomics is sacrificed and the code is rewritten data structures which are faster, but less ergonomic.

We wish to implement the Common Lisp list and sequence API in a way which is ergonomic, persistent by default, and dynamic. This means the resulting data structure must be usable as a list first, but sometimes also as a set or dictionary. By implementing the list API, we make it easy for people to use the structure, since all the functions they are used to using with lists exist and work just as they do for lists, they just exist in a different package.

As a first attempt at this task, Finger Trees[**?**] were first examined. The problem with Finger Trees is that, while they are easy to implement in Haskell, with its algebraic sum types and pattern matching, they are clunky to implement in a dynamically typed language like Lisp. Implementing Finger Trees involves polymorphic recursion. Every level in a finger tree contains deeper 2-3 trees than the level above it, and values are only at the leaves. The spine of the tree is different than the leaves.

This creates a lot of case-by-case work in Lisp. The CLOS is well designed, but can be slow, especially in the "hot path" in which data structure operations often find themselves. Without using the CLOS, we are left with the choice of using `defstruct` and `typecase`, which greatly complicates the implementation. This is especially true in a language without built-in support for pattern matching like Lisp.

It would be ideal if the data structure were relatively homogeneous – it looked the same throughout. This would greatly decrease case logic.

Finally, we need a structure that is tolerant to and implements functionality for non-persistent updates along side the persistent ones. This mirrors how lists are used in Common Lisp – persistent by default, but changeable if need be.

So, we need to create a data structure that is relatively homogeneous, with few cases to manage. It must be able to be used dynamically for different purposes. It must be persistent by default, but changeable if needed. Most important, it should have good running time when implementing the Common Lisp Standard's List API is implemented.

2 DEVELOPMENT

In this section we first lay out the API that needs to be implemented and summarize what functionality needs to exist for that API. We then develop an answer to that API with weight-balanced binary trees. Finally, develop even better run times by adding pennant lines to the mix.

Here is the API for lists which we will implement, or in some cases, reuse:

- `subst`, `subst-if`, `subst-if-not`
- `sublis`

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- atom
- copy-tree
- tree-equal
- copy-list
- list-length
- list, list*, make-list
- listp
- push
- pop
- nth
- null
- butlast
- pairlis
- assoc
- subsetp
- union
- intersection
- set-difference
- set-exclusive-or
- pushnew
- first..tenth
- cons
- acons
- endp
- ldiff
- tailp
- rest
- member, member-if, member-if-not
- map* functions

Also mutating versions:

- rplaca, rplacd
- nconc
- nsubst, nsubst-if, nsubst-if-not
- nbutlast

We also must deal with the generic sequences API. Here are the functions we choose to implement from that API:

Functions that modify the original structure:

- delete, delete-if, delete-if-not
- nsubstitute-if, nsubstitute-if-not
- fill
- map-into
- replace
- nsubstitute

Functions that don't care what the sequence is, they just need one (and therefore traversal needs to be kinda fast, hopefully somewhat locality-respecting).

- map
- reduce
- count-if, count-if-noT
- find-if, find-if-not
- position-if, position-if-not
- substitute-if, substitute-if-not
- remove-if, remove-if-not,
- search
- sort, stable-sort (sets sorted flag)

Table 1: Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
π	1 in 5	Common in math
$	4 in 5	Used in business
Ψ_1^2	1 in 40,000	Unexplained usage

Functions that can be implemented persistently and therefore need to be "kinda fast":

- elt
- copy-seq
- make-sequence
- subseq
- count
- length
- reverse
- find (check if sorted, check the sorted flag)
- position (check if sorted)
- mismatch (check if sorted)
- substitute (
- concatenate
- merge
- remove
- remove-duplicates (tree's left arm is <=, multiset)

(We will likely have to pare this down a bit, but it's still an interesting idea)

These APIs boils down to a persistent-by-default structure supporting the following:

- Efficient random insertion, deletion, and access (IDA)
- Efficient IDA at the ends
- Efficient traversal
- Efficient concatenation
- Efficient splitting
- Efficient set operations

It would also be nice to reuse some of the functions, like tree-equal and copy-tree.

3 TABLES

Immediately following this sentence is the point at which Table **??** is included in the input file; compare the placement of the table here with the table in the printed output of this document.

To set a wider table, which takes up the whole width of the page's live area, use the environment **table*** to enclose the table's contents and the table caption. As with a single-column table, this wide table will "float to a location deemed more desirable. Immediately following this sentence is the point at which Table **??** is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed output of this document.

Always use midrule to separate table header rows from data rows, and use it only for this purpose. This enables assistive technologies to recognise table headers and support their users in navigating tables more easily.

Table 2: Some Typical Commands

Command	A Number	Comments
`\author`	100	Author
`\table`	300	For tables
`\table*`	400	For wider tables

4 MATH EQUATIONS

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

4.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the **math** environment, which can be invoked with the usual `\begin . . . \end` construction or with the short form `$ . . . $`. You can use any of the symbols and structures, from α to ω, available in LaTeX [**?**]; this section will simply show a few examples of in-text equations in context. Notice how this equation: $\lim_{n\to\infty} x = 0$, set here in in-line math style, looks slightly different when set in display style. (See next section).

4.2 Display Equations

A numbered display equation—one set off by vertical space from the text and centered horizontally—is produced by the **equation** environment. An unnumbered display equation is produced by the **displaymath** environment.

Again, in either environment, you can use any of the symbols and structures available in LaTeX; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n\to\infty} x = 0 \tag{1}$$

Notice how it is formatted somewhat differently in the **displaymath** environment. Now, we'll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty} x_i = \int_0^{\pi+2} f \tag{2}$$

just to demonstrate LaTeX's able handling of numbering.

5 FIGURES

so important, please see https://www.acm.org/publications/taps/describing-figures/.

6 CITATIONS AND BIBLIOGRAPHIES

ACKNOWLEDGMENTS

Thanks is due to Dr. Kimball Germane of Brigham Young University, Provo, Utah for previewing drafts and offering feedback.

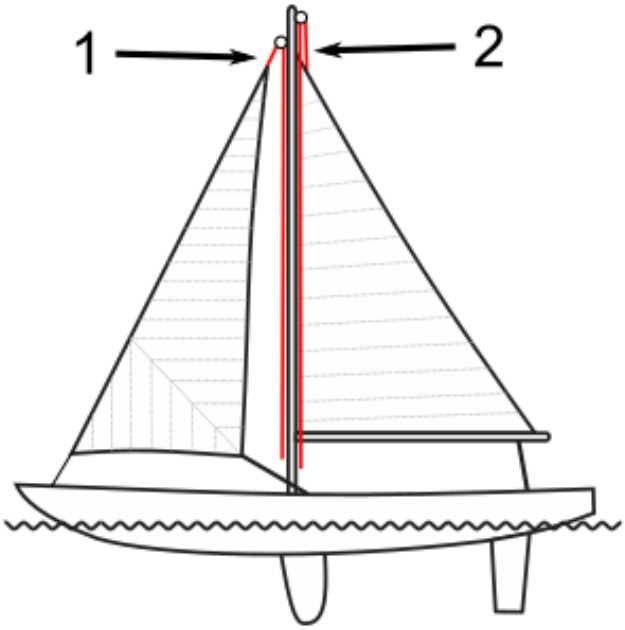

Figure 1: 1907 Franklin Model D roadster. Photograph by Harris & Ewing, Inc. [Public domain], via Wikimedia Commons. (https://goo.gl/VLCRBB).

A RESEARCH METHODS

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

B ONLINE RESOURCES

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit

congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.