

# Tuning Data and Control Structures for Incremental Computation

Kyle Headley\*

University of Colorado Boulder<sup>†</sup>

kyle.headley@colorado.edu

Incremental computation permeates modern software. A computation is incremental if repeating it with a changed input is faster than from-scratch re-computation. Programmers implement most contemporary incremental functionality in an ad-hoc manner, built into the efficient implementation of the system being designed. I have been working on language- and library-based incremental computation methods in order to increase the efficiency of code in a more systematic way.

**Background** The state of the art in general purpose, or language based, incremental computation involves building dynamic dependency graphs of program control flow and maintaining memoization tables of prior results. Dependency graphs allow a program to mark all functions affected by an input change, and memoization tables provide results of prior computations that may be used to update an affected function’s output without recomputing it. Dependency graphs may maintain a total order for efficiency, or a partial order for flexibility. Memoization tables often use keys with some computed value based on a function’s arguments, to allow irrelevant minor changes to be ignored. Two of the standards currently competing on general purpose incremental computation performance are SAC and Adapton. “Self adjusting computation” or SAC [2] is a set of techniques based around a totally ordered dependency graph and annotated function calls. (Nominal) Adapton [3] is an incremental computation engine based on a partially ordered dependency graph and user-defined “names” to key its memoization tables.

**Limitations** General purpose incremental computation requires a lot of overhead in memory and time. Storing dependency graphs and memoization tables consumes memory. Creating these features and the running the algorithms that search them consume

time. Incremental algorithms produce data with localized differences for different inputs, but the extent of the locality may not be known, requiring dependency graphs to mark and then clean functions that are not affected by a change.

**Approach** Programmers can group up functions and data used in an incremental setting so that dependency tracking and memoization act on groups rather than individual items. This technique puts the burden of the incremental system on the programmer, opening the door to human error. I design data structures and higher-order functions in the Rust language that contain additional grouping markers and grouping code respectively. A user then runs these functions over these data structures, which implicitly perform the groupings based on the markers. The data structure defines data markers by keeping data in multiple separate arrays (Rust uses the term “vector”), and function markers by flags in its nodes. User code for processing data runs on the vectors through Rust aggregate features. User code affecting a data structure node is only memoized if a flag is present. Non-memoized code is grouped with the memoized function that calls it.

## Gauged Incremental Random Access Zipper

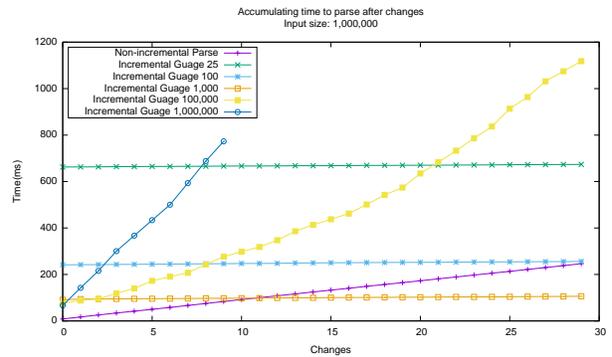
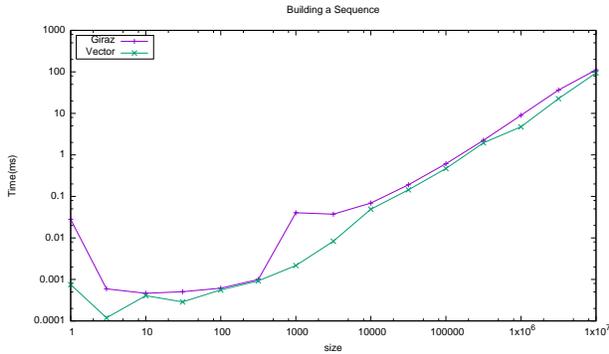
I am designing and implementing an incremental form of the Random Access Zipper (Raz) [4]. The Raz represents a sequence with a tree structure internally, one that rebalances only on the path from an edited node to the root. Algorithms written this way ensure compatibility with incremental computation. My enhancement replaces individual elements with vectors of elements, with the expected size of the vector called the “gauge”. The Raz is also enhanced with markers for function memoization and names for use with Adapton. Lets call this new version of the Raz a “Giraz”.

The Giraz does not exclusively perform incremental computations. When not involved in an incremental computation, or while preparing for one, the Gi-

---

\*ACM Student Member 1004593

<sup>†</sup>Work done as 2nd year graduate student, Spring ‘17



raz has similar performance to equivalent data structures. For example, the first figure shows the time to create a sequence from scratch using Rust’s vector or the Giraz. The test that generated this data pushes elements into an empty data structure until it reaches the target size. During this process the vector needs to occasionally relocate when it reaches its carrying capacity. The test also switches the mode of the Giraz from “edit” to “compute” when the target size is reached, which takes  $O(\log n)$  time, where  $n$  is the number of vectors(elements/gauge). This test uses gauge 1000 and elements with a primitive copy operation.

**Tuning** Making a dependency graph node for every aspect of a program requires so much memory that traversing it to adjust after a change may take longer than recomputation. While grouping data and functions can reduce this effect, it does not guarantee better performance. Incrementally updating a computation with everything grouped into one dependency node is equivalent to from-scratch computation but with the overhead of the incremental techniques. The user of the Giraz needs to chose parameters that optimize the performance gains of incremental computation. The design of the Giraz allows for this tuning in newly added data.

The second figure shows the affects of different gauges on a non-trivial toy computation. The task is to parse a string of numbers and addition symbols, interpreting the sequence like a reverse-polish calculation. The computation begins with a sequence of

characters and ends with a shorter sequence of integers. The native rust implementation of this requires under 10ms to parse a sequence of a million characters. The incremental test folds over the Giraz, memoizing at the internal markers, in this case located at vector boundaries. The best initial incremental computation shown takes at least 65ms. However, as the test proceeds, the time to recompute after an insertion to the character sequence is accumulated. There is a crossover point where it is faster to use incremental computation than to recompute after each edit. The test clearly demonstrates the trade off described above: low gauges update fast with slow initial computation, while high gauges update slow with fast initial computation.

**Related Work** Prior work on SAC introduced abstract data types [1] to group data and functions to be memoized. That work concentrates on a generic interface for hand-crafted data types that can not be further tuned. Later work on SAC introduces “blocked lists” [2] similar to using vectors in the Giraz. The block boundaries are based on the data, allowing the possibility of degenerate blocks. Late in the paper is a “reduce” benchmark that is similar to one written for the Giraz. At low gauges (block sizes), our results are similar, with the Giraz taking slightly more memory, and slightly more recomputation time. At high gauges, however, this situation is reversed, and additionally, the Giraz initial computation time is an order of magnitude faster.

## References

- [1] Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoglu. Traceable data types for self-adjusting computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 483–496, 2010.
- [2] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 227–240, 2014.
- [3] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA*, 2015.
- [4] Kyle Headley and Matthew A. Hammer. The random access zipper: Simple, purely-functional sequences. *CoRR*, abs/1608.06009, 2016.