Space and Time Optimizations In Adapton

Kyle Headley[1]
University of Maryland[2]
kylenheadley@gmail.com

Incremental Computation has great potential for improving computer performance in the future. A computation is incremental if repeating it with a changed input is faster than from-scratch recomputation. This can be done by storing partial results. Individual processor speed hasn't improved much lately, whereas the memory available to a processor continues to grow. Techniques like IC that use less compute cycles and more memory to generate results seem to be a perfect fit. I've been working on ways to balance this trade-off.

Not all IC needs to store partial results. For example, a list of a million integers does not need a full summation recalculated after a single number has been removed, we can just subtract it from the previous result. In general, though, there is no easy way to do this. Language-based methods have been developed that allow programmers to specify dependencies between subroutines. Partial results are automatically stored, or memoized, to be used next time the dependency is required. Adapton (Hammer et al. 2014) is being developed in this way, with a goal of providing incremental computation techniques to people without requiring much special training.

One drawback of language-based incremental computation is excessive memory usage. Maintaining tables of prior results and dependency information can easily take hundreds of times the memory of the data being manipulated. Working with this memory adds overhead to the computation time. In order to get the best performance out of a system, we need to find some way to balance the trade-off between incremental efficiency and memory usage.

Adapton allows the programmer to choose whether a particular function call will be memoized or have its computation accounted for in the calling function's memo-table entry. Data is accessed in a lazy manner, so memoized accessor functions can appear within data structures. The choice of when to memoize a function provides the mechanism for reducing memory usage. Unfortunately, it is not a trivial choice.

Functions need to be memoized somewhat uniformly throughout the computation to maintain balance regardless of what may change. This must be done in such a way as to be stable after a change, or the benefit of memoization is lost when a different function is memoized in response to a change. Memoized functions within data must maintain these properties even if the data has been transformed. I was able to come up with and implement some techniques to achieve these goals.

I proposed a memoization strategy for lists based on a hash function of each input element. At creation, an accessing function is memoized when the current element's hash satisfies a filter. Inserting and deleting elements has no effect on which accessors are memoized, other than those directly affected by the change. The main strategy is similar to the probabilistic chunking scheme of (Chen et al. 2014), but can be used with Adapton. It is

---

[1] ACM Student Member 1004593
[2] Work done as an undergraduate, Spring ‘15

implemented for lazy data structures. Functions over the list are memoized when taking memoized accessors as parameters. Modifying a value after the initial definition does not change which functions are memoized.

For efficiency in incremental processing, Adapton's lists are often first transformed into trees, and the results transformed back into lists. This can result in significant shuffling of the location of memoized functions. In our previous implementation, the ones in the branches were all mapped to locations at the beginning of the output list. My solution was to pass a flag to the function traversing the tree, such that memoization would occur further towards the leaves at each iteration. This procedure is deterministic, and when the final transformation back to a list occurs, the memoized functions are on the same level of the tree and handled uniformly.

An improvement to the PLDI14 Adapton (Hammer et al. 2014) paper was recently submitted to OOSLA15 as Nominal Adapton (Hammer et al. 2015), adding more customizable matching patterns for memoized functions. The data in the paper is produced by memoizing every function call. By using techniques described above, both the time and memory requirements have been reduced significantly.

Sorting is one of the computations that can be incremental without the need for language-based methods. However, mergesort is well-studied, has a complex dependency graph, and is structured in a way that's suited to the incremental computation style we're using. It is a good example of what can be done with our system. When memoizing each function call for mergesort, Nominal Adapton takes 0.94 seconds (on our test machine) to regenerate a sorted list of 100k items after an insertion. The memory required for the computation is around 1600 MB of space. With the above methods, the incremental change takes 0.05 seconds and 120 MB of space. We get similar results for subsequently calculating the median of that list, which is only possible if the sorting is able to compose with later computation.

References

M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. PLDI 2014.

M. A. Hammer, K. Headley, N. Labich, J. Dunfield, M. Hicks, J. S. Foster, D. Van Horn, Incremental Computation with Names. http://arxiv.org/pdf/1503.07792v1.pdf, 2015

Y. Chen, U. A. Acar, and K. Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *ICFP*, 2014.

W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.