Iterate, incrementalize, and implement: A systematic approach to efficiency improvement and guarantees

Yanhong Annie Liu
Computer Science Department
SUNY Stony Brook
liu@cs.sunysb.edu

People may more easily specify what they would like to compute if not concerned with how to compute efficiently. Given descriptions that may lead to inefficient computations, can we systematically obtain efficient algorithms and implementations? Can we further have efficiency guarantees for them?

We give an overview of a general and systematic method for achieving efficiency improvement and providing efficiency guarantees. The method is centered around incrementalization: making computation proceed repeatedly on small incremented inputs and effectively maintaining and using previously computed results. The method has three steps: iterate, incrementalize, and implement. Step 1 determines a minimum step to be taken repeatedly. Step 2 makes each step compute incrementally, particularly by using and maintaining and appropriate additional values. Step 3 designs appropriate data structures for storing and accessing the values maintained.

The central step of incrementalization [4] is: given a program f and an increment operation +, derive an incremental program that computes f(x+y) efficiently by using the result of f(x) [13], the intermediate results of f(x) [11], and auxiliary information of f(x) that can be inexpensively maintained [12]. This unifies existing approaches to incremental computation and is generally applicable; it exploits many existing program analysis and transformation techniques and can be systematically applied.

The method has been used successfully in optimizing expensive recursive functions and recursive data structures [11, 12, 8, 7, 10], optimizing loops and aggregate array computations [3, 5], transforming recursion into iteration [6], implementing sets and fixed points by Paige et al. [14, 2, 1], and currently implementing rules [9] and optimizing objects. Example applications include problems in list processing, graph algorithms, VLSI design, image processing, program analysis, and database queries.

These optimizations yield drastic algorithmic improvements. For example, incrementalizing recursive equations allows dynamic programming programs to be automatically derived [8, 7]. Not only are the resulting programs drastically faster, but the time and space complexities of the resulting programs can also be much more easily calculated. In particular, in the most recent work on implementing Datalog rules [9], the time and space complexities of the resulting algorithms can be calculated from the rules. A prototype system, CACHET, for optimizations based on incrementalization has been implemented and gradually extended.
References


