Incrementalization: From Clarity to Efficiency

Yanhong A. Liu
Computer Science Department, Stony Brook University, Stony Brook, NY 11794
liu@cs.stonybrook.edu

ABSTRACT

Incrementalization is at the core of a systematic program design method, especially for the design of algorithms and data structures. It takes a function and an input change operation and yields an incremental function that computes each new output by using the old output and additional values. It is the analogue of differentiation in continuous domains.

This article gives an overview of a systematic program design method centered on incrementalization. The method starts with a clear specification of a computation and derives an efficient implementation by semantics-preserving program transformations. The method applies to problems specified in imperative, database, functional, logic, and object-oriented programming languages with different data, control, and module abstractions. We illustrate the method through examples from optimizing compilers, graph algorithms, string processing, and program components. The last section discusses directions for future studies.

1. INTRODUCTION

At the center of computer science, there are two major concerns of study: *what* to compute, and *how* to compute efficiently. Problem solving involves going from clear specifications for "what" to efficient implementations for "how". Unfortunately, there is a conflict between clarity and efficiency: clear specifications usually correspond to straightforward implementations, not at all efficient, while efficient implementations are usually sophisticated, not at all clear. What is needed is a systematic method to go from clear specifications to efficient implementations.

Incrementalization is at the core of a systematic method for transforming a straightforward program that clearly meets its specification into an efficient program that still meets its specification but is usually less clear. This paper gives an overview of this method. The method has three steps:

1. Iterate—determine a minimum increment to take repeatedly, iteratively, to arrive at the desired output.
2. Incrementalize—make expensive operations incremental in each iteration by using and maintaining useful additional values.
3. Implement—design appropriate data structures for efficiently storing and accessing the values maintained.

The method is driven by Step Incrementalize: because efficient computations on nontrivial input must proceed repeatedly on incremented input, Step Incrementalize aims to make the computation on each incremented input efficient by storing and reusing values computed on the previous input. Steps Iterate and Implement are enabling mechanisms: to maximize reuse by Step Incrementalize, Step Iterate determines a minimum input increment to take repeatedly; to support efficient access of the stored values by Step Incrementalize, Step Implement designs efficient data structures to hold the values.

The method applies to a wide range of programming paradigms with different data, control, and module abstractions:

- Loops with primitives and arrays in imperative programming—no high-level abstraction, but it is essential to incrementalize expensive primitive and array computations in loops.
- Sets in database programming—high-level data abstraction, where besides incrementalization, it is also essential to design efficient data structures to implement sets.
- Recursion in functional programming—high-level control abstraction, where besides incrementalization, it is also essential to transform recursion into iterative computation.
- Rules in logic programming—high-level data and control abstraction, where besides incrementalization, both transformation of recursion and implementation of sets are essential.
- Objects in object-oriented programming—high-level module abstraction, where incrementalization is essential and must be done across modules.

For programs where low-level iterative control structures or low-level data structures are already used, there is either trivial work in the respective steps or crucial work to first determine what they are trying to achieve at a high level before applying the respective steps. The higher-level the abstractions used in specifying the problems are, the better the method works. For example, for problems specified using rules in Datalog, the method can provide precise time and space complexity guarantees for the generated implementations.

Besides being applicable to widely different programming paradigms, the method is general and systematic for at least two other reasons:

- It is based on language semantics and cost models: it consists of step-wise program analysis and transformations to replace repeated expensive computations by efficient incremental maintenance while preserving program semantics.

- It is the discrete counterpart of differential and integral calculus in continuous domain: incrementalization corresponds to differentiation, iteration corresponds to integration, and iterative incremental maintenance corresponds to integration by differentiation.

Steps Iterate and Incrementalize are essentially algorithm design, and Step Implement is essentially data representation design.

The method as developed has succeeded in solving large classes of problems of many different kinds; it does not yet do the magic of generating efficient implementations from clear specifications for *all* computation problems, if such a magic method will ever exist. For example, the method can derive dynamic programming algorithms from recursive functions, produce appropriate indexing for efficient evaluation of relational database queries, and generate efficient algorithms and implementations from Datalog rules; however, it cannot yet derive a linear-time algorithm for computing strongly connected components of graphs. It is, of course, not the only method for program design. The last section discusses directions for future studies.

The method is the result of over twenty years of research by the author and collaborators, developed based on significant prior work by many others, especially work pioneered by Paige and collaborators [18, 19]. This article provides only an overview of the method, illustrating the most important ideas through examples. The last section includes bibliographical notes for some of the earliest important works and some of the closest works to the parts discussed in this article. More detailed discussion of the method, with detailed transformations, additional examples, and extensive bibliographical notes, appear in a new book [12].

2. LOOPS

We start with loops and consider a simple example, repeated multiplication, to illustrate the basic ideas of Step Incrementalize. The example is to compute a×i repeatedly, where a holds some constant value, and i is initialized to 1 and is incremented by 1 as long as it is smaller than or equal to some constant value held in b, as in the following program:

```
i := 1       // initialize i to 1
while i<=b:  // iterate as long as i<=b
  ...
  ...a*i...  // compute multiplication
  ...
  i := i+1   // increment i by 1
```

We use `v := e` for assigning the value of e to variable v. The goal is to replace repeated multiplications with additions. It is an example of one of the oldest, most important compiler optimizations, called *strength reduction*, designed to optimize such multiplications that occur extensively for computing the addresses of array elements in languages like FORTRAN. The basic idea also underlies the Difference Engine—the first computing device, designed to tabulate polynomial functions in the 19th century, and ENIAC—the first electronic general-purpose digital computer, designed to calculate artillery firing tables in World War II.

Incrementalize: maintain invariants. First of all, the optimization must be based on language semantics and the cost model. Here it exploits algebraic properties of the primitives, `a*(i+1) = a*i+a`, and that multiplications are expensive whereas additions are not. The basic idea is to store the value of `a*i` and initialize and update it incrementally as i is updated, but where and how exactly?

The key idea is to maintain the *invariant* that the stored result in a fresh variable equals the value of the expensive computation, here `c = a*i`, where c is a fresh variable. The invariant is maintained by updating the stored value incrementally at every update that may affect the value of the expensive computation, here performing `c:=a` at `i:=1` and performing `c:=c+a` at `i:=i+1`. Then, every occurrence of the expensive computation is replaced with a retrieval of its value from the stored result, here replacing `a*i` with c. This yields the following optimized program:

```
i := 1
c := a       // new: c = a*i = a*1 = a
while i<=b:
  ...
  ...c...    // a*i is replaced with c
  ...
  i := i+1
  c := c+a   // new: c' = a*i' = a*(i+1) = a*i+a = c+a
```

At an update to a variable v, incremental maintenance may be performed either before or after the update if the maintenance does not use the value of v, as in the example above. In general, the maintenance may use the value of v. If it uses the value of v before the update, then it must be performed before the update. If it uses the value of v after the update, then it must be performed after the update.

Maintaining invariants eagerly at all updates allows the maintained value to be used anywhere outside the updates and the corresponding maintenance. However, if multiple updates happen before a maintained result is used, where some updates may even cancel each other out, it may be more efficient to do incremental maintenance on demand, i.e., when the result is needed, as opposed to whenever an update occurs.

When expensive computations are more complex, incrementalization may need to maintain additional values: *intermediate results*—values computed in the original expensive computation besides the final result, and *auxiliary values*—values not computed in the original computation but useful in computing the new result incrementally. This reflects a well-known trade-off between time and space. Useful intermediate results and a general class of auxiliary values can be determined by analyzing expensive computations on the changed input.

Incrementalization can also be applied repeatedly: after applying incrementalization, which replaces an expensive computation with incremental maintenance, there may still be expensive computations in the incremental maintenance; incrementalization can then be applied again to replace these expensive computations with incremental maintenance, and this process may repeat.

Applications and need for higher-level abstraction. Incrementalizing loops with primitives and arrays has important classes of applications, including:

- Strength reduction, as in optimizing compilers. It optimizes not only array element access, e.g., for matrix manipulations in scientific computing, but also repeated expensive numeric operations in general.

- Hardware design, including for the Difference Engine and ENIAC. This also benefits from additional loop optimizations enabled by incrementalization, for reducing the result, condition, and body of the loop.

- Image processing, expressed as aggregate computations over arrays. Incrementalization replaces expensive aggregate computations with incremental updates of the aggregate values, yielding efficient algorithms.

Unfortunately, many applications naturally require more complex processing of more complex data than loops and arrays. Clear specifications of these computations require high-level data abstraction using sets and high-level control abstraction using recursion, as discussed in examples in subsequent sections.

3. SETS

We consider computations expressed using sets. We use the graph reachability problem as an example: given a graph with a set `e` of edges, each being a pair of a start vertex and an end vertex, and a set `s` of source vertices, the problem is to find the set `r` of vertices reachable from vertices in `s` following edges in `e`. This can be done by starting at the source vertices and adding each new reachable vertex to the result until no more new vertices can be reached, as in the following program:

```
r := s
while exists x in {v: (u,v) in e, u in r, v not in r}:
  r := r+{x}
```

Here, `{v: (u,v) in e, u in r, v not in r}` computes the set of vertices `v` newly reachable following any one edge—`v` being the end vertex of an edge from `u` to `v` in `e` where `u` is already reached and `v` is not. The goal is to design an algorithm and data structures to compute `r` efficiently, avoiding the expensive set computation in each iteration of the `while` loop.

Incrementalize: exploit maps. We maintain an invariant for the result of an expensive computation, as discussed in the previous section, here `w = {v: (u,v) in e, u in r, v not in r}`, where `w` is a fresh variable.

The basic idea is to use values bound by an update to efficiently retrieve other values needed in the expensive computation. To maintain `w`, at update `r := r+{x}`, the second clause in the computation of `w` has `u` bound to the new value `x` in `r`, so the key is to retrieve the matching `v`'s that satisfy `(x,v) in e` according to the first clause. This set of `v`'s is denoted `e{x}`, called the *image set* of `x` under `e`, i.e., `e` is used as a map from each element in the first component, the *domain*, to the corresponding set of elements in the second component, the *range*. Afterwards, the third clause is tested. Finally, the new element is added to `w` if it is already not in `w`. At the same time, update `r := r+{x}` also lets the third clause bind `v` to `x`, which was in `w` but no longer satisfies the third clause, so `x` is removed from `w`. Together, these yield the maintenance block in the body of the `while` loop for maintaining `w` at `r := r+{x}`. The same block is used in initialization as each element of `s` is added to `r` starting with `{}`.

```
w := {}               // new: w is {} when r is {}
r := {}               // r := s is done by for-loop
for x in s:
  ...                 // new: new block as in while-loop
  r := r+{x}
while exists x in w:  // {v: ...} is replaced with w
  for v in e{x}:      // new: use e as map to retrieve v
    if v not in r:    // new: test the condition on v
      if v not in w:  // new: if v is not already in w
        w := w+{v}    // new: add v to w
  if x in w:          // new: if x is in w, true here
    w := w-{x}        // new: delete x from w
  r := r+{x}
```

We will see next how to implement each set operation here in worst-case $O(1)$ time, and thus the total running time here is bounded by the number of iterations of the nested loops—the outer loop by the number of vertices, and the inner loop by the number of outgoing edges of each vertex. Therefore, the total running time is bounded by the number of edges, $O(|\texttt{e}|)$.

Here, we described incremental maintenance of `{v: (u,v) in e, u in r, v not in r}` directly. One could also specify this set computation as `{v: (u,v) in e, u in r} - r`, and maintain the left set as an intermediate result before maintaining the set difference, yielding also an $O(|\texttt{e}|)$ algorithm. Maintaining invariants following the chain of dependencies corresponds to the chain rule in calculus. Unnecessary intermediate results can be eliminated at the end, though it is nontrivial for this example.

One could also specify the expensive set computation in the original program as `s + {v: (u,v) in e, u in r} - r` and initialize `r` to `{}` instead of `s`, that is, move `s` from initialization to the `while` loop. Incrementalization will generate a similar $O(|\texttt{e}|)$ algorithm, except that the initialization code will be drastically simplified.

Implement: design linked data structures. It is well known that set operations can be implemented using hashing in $O(1)$ time, but that is average-case time, not worst-case time, and hashing has a large constant-factor overhead. A general method can design linked data structures for a large class of problems to support set operations in worst-case $O(1)$ time.

The method represents a set as a linked list, and represents a map as a linked list of linked lists, but combines everything based on *associative access*—membership test (`x in s` and `x not in s`) and image operation (`e{x}`); such an access requires the ability to locate an element (`x`) in a set (the set `s` or the domain of `e`). If associative access can be done in constant time, so can all other operations.

The basic observation is that an access of `x` in `S` in a program is not isolated—the element `x` must be retrieved from some set `W` before the access, as in

```
...  // retrieve x from W
...  // access x in S
```

The idea is to use a set B, called a *base*, to store values for both `W` and `S`, so that a retrieval of a value from `W` also locates this value in S. Base B is a set of records (this set is only conceptual), with a K field storing the key (i.e., element value).

- Set S is represented using an S field of B: records of B whose keys are in S form a linked list, with links stored in the S field; other records store a null value in the S field. If S is only tested for membership, the S field can be just a bit.
- Set W is represented as a linked list of pointers to records of B whose keys are in W.

This representation is called *based representation.* It allows an arbitrary number of sets like W, called *weakly based*, but only a constant number of sets like S, called *strongly based.* Essentially, base B provides a kind of indexing to elements of S starting from elements of W.

For the graph reachability example, the base is a collection of records, one for each vertex in the graph. Set s is represented as a linked list of pointers to the records for vertices in the set, and so is each image set e{x}. Domain of e, set r, and set w are represented using three fields in the records, storing a pointer to the image set linked list, a bit, and a link for the next element, respectively.

Applications and need for control abstraction. Optimizing set computations has a vast number of important applications, including:

- Graph algorithms, for processing any kind of connections among objects. Graph reachability and similar problems are particularly important, e.g., in program analysis, model checking, and simply computing the prerequisites of courses.
- Query optimization, especially in relational database, where relations are just sets of tuples. Systematic incrementalization has led to advances over the best prior results.
- Access control, with complex policy analysis and enforcement. Incrementalization allowed the ANSI standard for Role-Based Access Control (RBAC) to be specified more clearly and efficient implementations to be generated automatically.

Despite these, some more sophisticated applications require the use of recursion to be expressed more clearly, as discussed next.

4. RECURSION

We consider computations expressed using recursive functions. We use the longest common subsequence problem as an example: given two sequences x[1]..x[i] and y[1]..y[j], the problem is to compute the length of a longest common subsequence (LCS) of the two sequences, where a subsequence of a given sequence is just the given sequence possibly with some elements left out. This can be computed using the following recursive function—if either sequence is empty, i.e., i = 0 or j = 0, then the result is 0; if both are not empty and x[i] = y[j], then every LCS must end with this common element, and the result is 1 plus the length of an LCS of x[1]..x[i-1] and y[1]..y[j-1]; otherwise, both sequences are not empty and x[i] $\neq$ y[j], so an LCS either does not end with y[j] or does not end with x[i], and the result is the maximum of these two possibilities:

```
def lcs(i,j) where i>=0, j>=0:
  if i=0 or j=0 then 0
  else if x[i]=y[j] then 1 + lcs(i-1,j-1)
  else max(lcs(i,j-1), lcs(i-1,j))
```

Straightforward computation of the recursive function takes exponential time, because of overlapping recursive calls for subcomputations. The goal is to design an efficient algorithm, known as a *dynamic programming* algorithm, that appropriately computes and stores the results of subcomputations and reuses them for enclosing computations.

Iterate: determine minimum increments. What is an appropriate order of subcomputations for recursively defined functions? In general, how should computations proceed on repeatedly incremented input? This is theoretically hard, and there is no general solution—appropriate input increments correspond to well-founded orderings in domain theory and steps for induction in proof theory. However, we have found a simple but powerful method based on the underlying principle of incrementalization; albeit a heuristic, it has succeeded on all problems we have encountered.

The idea is to let the increment be a minimum change opposite the arguments of recursive calls. The rationale is that (1) taking the opposite of arguments of recursive calls gives an increment, the direction that results on subproblems are used in computing results on bigger problems, and (2) minimizing change allows maximizing reuse in incremental computation, the essence of the overall method.

1. Determine arguments of all recursive calls. For function lcs, they are (i-1,j-1), (i,j-1), and (i-1,j).
2. Take any one that changes minimally from the parameters of the function. For lcs, this is either one of the last two; say we take the last one, (i-1,j).
3. Take the opposite of the change, yielding a minimum increment operation, denoted next. For lcs, this yields next(i,j) = (i+1,j).

Then one can compute the original function by starting at the base case and repeatedly computing on an incremented input, as discussed next.

Incrementalize: derive incremental functions. How to store and use appropriate values for incremental computation? The basic idea is to use the stored result of a function f for computing f on the incremented input by next, similarly as discussed in the previous two sections, yielding an incremental function f'(x,r) that takes an additional argument r and satisfies: if r = f(x), then f'(x,r) = f(next(x)). For lcs, the goal is to obtain an incremental function lcs'(i,j,r) to compute lcs(i+1,j) using r = lcs(i,j). Expanding lcs(i+1,j) by definition and simplifying i+1-1 to i yield:

```
lcs(i+1,j) = if i+1=0 or j=0 then 0
             else if x[i+1]=y[j] then lcs(i,j-1)+1
             else max(lcs(i+1,j-1), lcs(i,j))
```

Consider the three function calls in computing lcs(i+1,j): lcs(i,j-1) also appears in lcs(i,j); lcs(i,j) has value r; and lcs(i+1,j-1) equals a recursive call lcs'(i,j-1,lcs(i,j-1)) by definition of lcs', where the third argument is the same as the first of the three function calls. Therefore, lcs(i,j) is transformed to lcsExt to cache also the value of `lcs(i,j-1)` recursively in an additional component of the result, and the corresponding incremental function `lcsExt'` uses that value as considered above and maintains it, leaving only a single recursive call to `lcsExt'` itself.

We omit the details of the remaining transformations, but from the discussions above, we can already see that the optimized `lcs` calls `lcsExt'` repeatedly $O(\texttt{i})$ times on incremented input, and `lcsExt'` calls itself recursively $O(\texttt{j})$ times, each call taking $O(1)$ time. Thus the total time complexity is $O(\texttt{i} \times \texttt{j})$. Caching the value of `lcs(i,j-1)` in `lcsExt(i,j)` recursively requires $O(\texttt{j})$ auxiliary space, not $O(\texttt{i} \times \texttt{j})$ space as used by memoization or a two-dimensional array.

Applications and need for data abstraction. Many computation problems are best expressed using recursive functions and benefit from incrementalization of expensive functions. Three kinds of example problems are:

- Sequence processing, or string processing. Sources of these problems range from text document comparison to biological sequence analysis.
- Combinatorial optimization. Solving these problems requires considering overlapping subproblems and depends crucially on dynamic programming for efficiency.
- Math and puzzles. Although these are well-studied, systematic incrementalization has led to interesting insights and discoveries, even for problems as simple as the factorial function.

Despite these, recursive functions are not suitable for describing computations on arbitrary collections of data, whereas set expressions are, as discussed in the previous section.

5. RULES

We look at computations expressed using logic rules. We consider the transitive closure problem as an example: given a graph with a set of edges, the problem is to compute the transitive closure of the graph—the set of all pairs of vertices `u` and `v` such that there is a path from `u` to `v` following a sequence of edges. Let `edge(u,v)` denote that there is an edge from vertex `u` to vertex `v`, and let `path(u,v)` denote that there is a path from vertex `u` to vertex `v` following a sequence of edges. The problem can then be expressed as computing the set of all `path` facts using two logic rules: if there is an edge from vertex `u` to vertex `v`, then there is a path from `u` to `v`; if there is an edge from `u` to `w`, and there is a path from `w` to `v`, then there is a path from `u` to `v`.

```
edge(u,v) -> path(u,v)
edge(u,w), path(w,v) -> path(u,v)
```

The goal is to design an efficient algorithm. Given that high-level abstractions are used for both control and data, we need to determine how the computation should proceed iteratively and incrementally, and how to represent and access all the facts. Because such logic rules are very high-level, we can compile the given rules into an efficient implementation with time and space complexity guarantees.

Iterate: transform to fixed points. The meaning of a set of rules and a set of facts is the least set of facts that contains all the given facts and all the facts that can be inferred using the rules. To compute this meaning, the set of rules is first transformed into a fixed-point computation. Using our method for determining minimum increments, we choose the addition of a single new fact as the increment in each iteration of the fixed-point computation. This generates the following `while` loop:

```
R := {}          // initialize the result set
while exists x in e0 + e1(R) + e2(R) - R:
  R := R + {x}  // add one fact to the result set
```

where `e0` is the set of given `edge` facts, and `e1` and `e2` are sets of new `path` facts inferred using the two respective rules based on the facts in `R`:

```
e0    = {[edge,u,v]: edge(u,v) in givenFacts}
e1(R) = {[path,u,v]: [edge,u,v] in R}
e2(R) = {[path,u,v]: [edge,u,w] in R, [path,w,v] in R}
```

Here, facts are represented as tuples containing the relation name and arguments.

Incrementalize: exploit auxiliary maps. We incrementally maintain the result of expensive computation `e0 + e1(R) + e2(R) - R` in a fresh variable `W`, when an `edge` or `path` fact is added to `R`. Use of rule 1 as in `e1(R)` is easy: when an `edge` fact is added to `R`, a corresponding `path` fact is added to `W` if not already present. Use of rule 2 as in `e2(R)` requires the use of maps for efficient retrieval: when an `edge(u,w)` is added to `R`, the matching `v`'s are found by using `path` as a map, and when a `path(w,v)` is added to `R`, the matching `u`'s are found by using as a map an auxiliary relation `edgewu` that is the inverse of `edge`:

```
edgewu = {[w,u]: [edge,u,w] in R}
```

Initialization of `W` and addition of a set to `W` will be done by a `for` loop, and addition of an element to `W` will be guarded by a test that the element is not in `W` already. Finally, tuple operations will be done one component at a time, e.g., test `[x,y] in S` will be done as `x in domain(S) and y in S{x}`.

The total number of iterations of the `while` loop, including iterations of the `for` loops to find the matching variable values using maps, is bounded by the number of combinations of facts that make all hypotheses of a rule true at the same time, summed over all rules. This is also the worst-case time complexity, because each iteration takes worst-case $O(1)$ time using the data structure design discussed next. For the transitive closure example, let `Vertex` be number of vertices and `Indegree` be the indegree of the given graph. Then time complexity is the sum of $O(|\texttt{edge}|)$ for the first rule and $O(\min(|\texttt{edge}| \times \texttt{Vertex}, |\texttt{path}| \times \texttt{Indegree}))$ for the second rule. That is, the time complexity is bounded by the number of edges times the number of vertices, and is also bounded by the output size if `Indegree` is a constant.

Implement: design linked and indexed data structures. Similarly as discussed for the design of linked data structures for sets, based representation is used to represent facts—sets of tuples, except that linked list alone is not sufficient, because in general a non-constant number of sets may need to be strongly based. For example, test `y in S{x}` requires `S{x}` to be strongly based, but the number of such sets is the size of the domain of `S`, which is not a constant.

The idea is to use an array for each such component of the tuple, and use linked lists for other sets. This yields combined linked and indexed data structures, with nested array structures for sets of tuples with associative access to every component. The space complexity is the output size plus the space for auxiliary maps.

For the transitive closure example, elements of the base are stored in an array indexed by the vertices `1` to `Vertex`, for efficient access of the first component of `edgewu` and of `path` facts in `R` and `W`. Each element `u` of the base array is a record of six fields: two arrays for the second component of `path` facts in `R` and `W`, two linked lists for traversing elements in those two arrays, a linked list for the second component of `edgewu`, and a linked list for the second component of `edge` facts. The first component of `edge` facts is a linked list. The output space is $O(|\mathsf{edge}|^2)$, and the auxiliary space is $O(|\mathsf{edge}|)$.

Applications and need for module abstraction. The method described applies to all Datalog rules with no more than two hypotheses. Datalog rules are a general class of rules where each argument of a relation is an atomic value, not a structured value. Rules with more than two hypotheses can be decomposed into rules with two hypotheses first. Many difficult query problems can be specified easily using Datalog.

- Complex database queries, especially queries involving both sets and recursion. These include graph queries and semantics web queries that are hard or impossible to express in traditional relational databases.
- Program analysis, as well as verification and model checking. Datalog is excellent for expressing program flow and dependency analysis, including complex pointer analysis, for which the method described has led to improved algorithms and complexity analysis.
- Trust management, for access control in decentralized systems. These policies are very complex and are made significantly simpler by using Datalog with constraints; so are certain network programming tasks.

All set expressions, recursive functions, and logic rules are best only for expressing isolated computations. To build software systems with many components, module abstraction is needed and is discussed next.

6. OBJECTS

We finally consider the use of objects to provide module abstraction in building software components. We use a simple linked-list component as an example. A linked-list component provides the functionalities of a linked-list implementation, such as adding an element at the beginning of the list and returning the number of elements in the list.

Although components separate how operations are implemented inside from what these operations are to the users outside, there is a conflict between clarity and efficiency in implementing the operations. To resolve this conflict, incrementalization is needed to transform straightforward but inefficient computations into sophisticated but efficient implementations for these components.

Queries and updates: clarity versus efficiency. *What* functionalities a component provides can be classified as, or decomposed into, two kinds of operations: queries and updates. Queries compute results using data without changing data, and are sometimes called views or observations. Updates change data. For example, the `LinkedList` class in Java has a query method `size` that returns the number of elements in the list, almost two dozen update methods that add or remove elements, and over a dozen other query methods that return elements, their indices, membership test results, and so on. These numbers have in fact been increasing since `LinkedList` was first introduced in Java.

How to implement the queries and updates can vary significantly.

- A straightforward implementation lets each operation do only its respective query or update and is clear and modular. For example, the `size` method can iterate over elements in the list to count them, and each update method can do just the specified addition or removal of elements. However, this can have poor performance, because queries may be repeated, and many are expensive. For example, `size` takes time linear in the number of elements in the list, and if it occurs in a loop, the overall running time blows up.
- A sophisticated implementation can have good performance, by storing the results of expensive queries appropriately and maintaining them incrementally when the data are updated. For example, the `LinkedList` class may maintain the result of `size` in a field. However, this is not modular, less clear, and error-prone, because each update that may affect a query result must update that result appropriately. In the `LinkedList` class, each of the almost two dozen update methods must also update the field for `size` appropriately.

Clearly, there is a conflict between clarity and efficiency, even for the simple `LinkedList` example. The situation becomes much worse for complex systems that have many queries and updates, where queries may cross multiple classes, and updates may be spread in many places. A query can be affected by many updates, and an update can affect many queries. It poses a serious challenge to consider complex dependencies and trade-offs and decide where and how to maintain what results, and the resulting code can become significantly more difficult to understand. This conflict between clarity and modularity, and thus software productivity and cost, on one side, and program efficiency and system performance on the other side, manifests itself widely in complex systems and components.

Incrementalize: develop and apply incrementalization rules. Incrementalization first analyzes expensive queries and updates to values that queries depend on, taking aliasing into account as appropriate. Then it must determine where to store the query results, and where and how to update them. Module abstraction makes this more difficult. In general, there may be many expensive queries and many kinds of updates that are interdependent and spread across different components. How to incrementally maintain all the invariants involved under all updates in a coordinated way, using a systematic and even automatic method?

The idea is to maintain invariants one at a time, following the chain rule as discussed in Section 3, except that the maintenance of an invariant at a set of updates may be much more sophisticated, so a declarative language is desired for specifying the transformations. One may use *incrementalization rules* to specify coordinated maintenance of a query result at a set of updates to the values that the query depends on, maintaining the invariant that the value of a fresh variable equals the result of the query.

For example, the following incrementalization rule expresses that, to maintain the invariant $r = s$`.size()` when all updates that may affect the size of s are s `:= new set()`, s`.add(`x`)`, and s`.del(`x`)`, the respective maintenance is setting r to `0`, incrementing r by `1` if x is not in s before the addition, and decrementing r by `1` if x is in s before the removal; the cost of the original query is linear in the size of s, and the cost of each update and maintenance is constant.

```
inv r = s.size()              O(|s|)

at  s := new set()            O(1)
do  r := 0                    O(1)

at  s.add(x)                  O(1)
do before                     O(1)
   if not s.contains(x):
     r := r + 1

at  s.del(x)                  O(1)
do before                     O(1)
   if s.contains(x):
     r := r - 1
```

In general, one may also use **if** clauses under the **inv** and **at** clauses to specify conditions on the query and updates, including the classes, methods, etc. in which the queries and updates may occur, and **de** clauses to define new classes, methods, etc. Indeed, for the costs listed in the rule above to be correct, the last two updates need a condition that x is a variable, not an arbitrary expression.

A library of incrementalization rules can be built and reused from application to application, as opposed to being rediscovered and manually embedded in scattered places in each application program. Rules should be developed systematically, following general methods for maintaining invariants, whenever possible; indeed, a large class of rules can be derived automatically. Other rules that we do not yet know how to systematically derive can be added manually when they are discovered. The correctness of rules needs to be verified rigorously.

Applications and need for additional abstraction. Among widespread applications of incrementalization across components, we mention two general classes:

- Information systems, such as management information systems (MIS) and electronic health records (EHR). Objects can model components at different granularities, and object queries can help express functionalities at a very high level.
- Simulations and games, such as network simulations and large role-playing games. In these applications, many objects constantly interact with each other in a dynamic environment, querying and updating many attributes of the objects.

Incrementalization rules can be interpreted in general as invariant rules, used for not only optimization, but also reverse engineering and verification: whereas optimization yields incremental maintenance given the invariant and updates, reverse engineering discovers the invariant given the updates and maintenance, and verification checks consistency when all three of the invariant, updates, and maintenance are given.

Finally, for querying complex objects and relationships, one can use not only set expressions, recursive functions, and logic rules as discussed in previous sections, but also powerful path expressions, as additional abstraction, to query graphs that capture the objects and relationships.

7. CONCLUSION

We discuss a few important remaining issues and provide bibliographical notes.

Building up and breaking through abstractions. Incrementalization allows the transformation of a straightforward computation to proceed on repeatedly incremented input, use and maintain previously computed values in each iteration, and store and access these values efficiently, arriving at an efficient implementation. This in turn supports the computation to be specified clearly at a high level without sophisticated implementation details. Whereas developing clear high-level specifications require building up data, control, and module abstractions, transformation into efficient implementations requires breaking through the abstractions—to determine iterative structures from recursive definitions, design efficient data structures from set operations, make expensive computations incremental, and do so across components. Methods for both building up and breaking through abstractions need further studies together.

Implementations and experiments. Although only sketched in this overview, the transformational method for incrementalization is based on analyzing each program construct, for structures of control and operations on data, to determine increments for iterative computation, additional values for incremental computation, etc. On the one hand, the method can be used manually for algorithm design. On the other hand, with the heuristic for finding increments and conservative approximations for analyzing dependencies and equalities, the transformations can be fully automated, as prototype implementations have shown. Experiments on a wide variety of applications have resulted in new or improved algorithms and implementations, improved complexity analysis and understanding of existing algorithms, as well as improved problem specifications. However, much more work is needed to improve the implementations.

Limitations and future work. There are many limitations of the method and many research problems that need further study. First, better characterizations of the method are needed to help put it into perspective. At the same time, extensions of the method are needed in many areas: space-driven optimization, concurrent and distributed programming, languages supporting all high-level abstractions, and more. Study is also needed on relationships with design and optimization in continuous domains for the design of engineering systems, as studied in applied mathematics and in sciences like physics. Finally, substantial effort is needed to validate the method and future developments through experiments with large-scale applications.

Bibliographical notes. The ideas of incrementalization underlie a large body of research [20, 12]. The following are some of the earliest important works and some of the closest works to the parts discussed in this article. Interested readers may find detailed and additional discussions in [12]. The most basic idea can be traced back to the difference method of Henry Briggs in the 16th century, used by the Difference Engine of Charles Babbage in the 19th century [7].

- Maintaining and strengthening loop invariants by Dijkstra [5], Gries [9], and others, including techniques for strength reduction [8, 3, 4].
- Finite differencing of set expressions by Earley [6], Paige et al [18, 19], and others, and incremental view maintenance in database [2, 10]. The graph reachability example in Section 3 is simplified from [12] based on ideas from Liu et al [15].
- Transforming recursive functions by Burstall and Darlington [1] and others, and various techniques for memoization [17]. The longest common subsequence example in Section 4 is discussed in detail in [12].
- Tabling for logic programs by Tamaki and Sato [22] and others, and methods for efficient evaluation with complexity analysis [24, 16, 23]. The transitive closure example in Section 5 is discussed in detail in [13, 12].
- Incrementalization of computations on objects [11, 21]. The linked-list class example in Section 6 is from [14, 12].

Acknowledgment

I would like to thank all my students and other collaborators who contributed to this work. I would like to thank Moshe Vardi for his encouragement and invitation for this article. This work was supported in part by ONR grants N00014-92-J-1973, N00014-99-1-0132, N00014-01-1-0109, N00014-04-1-0722, and N00014-09-1-0651, and NSF grants CCR-9711253, CCR-0204280, CCR-0306399, CCR-0311512, CNS-0509230, CCF-0613913, CCF-0964196, and CCF-1248184.

8. REFERENCES

[1] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[2] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589. Morgan Kaufman, 1991.

[3] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, 1977.

[4] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, 2001.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[7] H. H. Goldstine. Charles Babbage and his Analytical Engine. In *The Computer from Pascal to von Neumann*, pages 10–26. Princeton University Press, 1972.

[8] D. Gries. *Compiler Construction for Digital Computers.* Wiley, 1971.

[9] D. Gries. *The Science of Programming.* Springer, 1981.

[10] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *Materialized Views: Techniques, Implementations, and Applications*, pages 145–157. MIT Press, 1999.

[11] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 258–267, 1991.

[12] Y. A. Liu. *Systematic Program Design: From Clarity To Efficiency.* Cambridge University Press, 2013.

[13] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):1–38, 2009.

[14] Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

[15] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*, pages 112–120, 2006.

[16] D. A. McAllester. On the complexity analysis of static analyses. In *Proceedings of the 6th International Static Analysis Symposium*, pages 312–329. Springer, 1999.

[17] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[18] B. Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 58–71, 1977.

[19] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[20] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 502–510, 1993.

[21] T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 55–66. ACM Press, 2008.

[22] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 84–98. Springer, 1986.

[23] K. T. Tekle and Y. A. Liu. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 661–672, 2011.

[24] M. Y. Vardi. The complexity of relational query languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 137–146, 1982.