Incremental Computation: What Is the Essence? (Invited Contribution)

Yanhong A. Liu
Stony Brook University
USA
liu@cs.stonybrook.edu

Abstract

Incremental computation aims to compute more efficiently on changed input by reusing previously computed results. We give a high-level overview of works on incremental computation, and highlight the essence underlying all of them, which we call incrementalization—the discrete counterpart of differentiation in calculus. We review the gist of a systematic method for incrementalization, and a systematic method centered around it, called Iterate-Incrementalize-Implement, for program design and optimization, as well as algorithm design and optimization. At a meta-level, with historical contexts and for future directions, we stress the power of high-level data, control, and module abstractions in developing new and better algorithms and programs as well as their precise complexities.

CCS Concepts: • **Software and its engineering** → **Language features**; **Compilers**; • **Theory of computation** → **Semantics and reasoning**; *Design and analysis of algorithms*; • **Information systems** → *Database query processing*.

Keywords: Incremental Computation, Incrementalization, Algorithm Design and Optimization, Program Design and Optimization, High-Level Abstractions

ACM Reference Format:
Yanhong A. Liu. 2024. Incremental Computation: What Is the Essence? (Invited Contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '24), January 16, 2024, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3635800.3637447

1 Introduction

As the real world changes continually, computer programs that handle input from the real world must handle continually changing input. Furthermore, because any algorithm that solves a nontrivial problem must proceed in an iterative or recursive fashion, computations must be performed on repeatedly changed state. The general area of incremental computation studies how to efficiently handle continually changing input by storing and reusing previously computed results.

There was already a large literature on incremental computation even over 30 years ago [110], and the area has grown significantly since then. [1] To grasp the depth and breadth of the area, it is important to understand the essence underlying different works and results.

This article gives a high-level overview of works on incremental computation—organizing them into incremental algorithms, incremental evaluation frameworks, and incremental-program derivation methods—and highlights the essence underlying all of them, which we call incrementalization.

> Given a program f and an input change operation $\oplus$, incrementalization aims to obtain an incremental version of f under $\oplus$, denoted f', that computes on the changed input more efficiently by reusing results computed before the change.

It is the discrete counterpart of differentiation in calculus.

We then review the gist of a systematic method for incrementalization [66], and a systematic method centered around it, called Iterate-Incrementalize-Implement (III), for program design and optimization, as well as algorithm design and optimization [67].

- Systematic incrementalization is a general transformational method for deriving incremental programs that compute more efficiently by exploiting the previous result, intermediate results, and auxiliary information. The method consists of systematic program analysis and transformations that are modular, and that are drastically easier and more powerful for programs that use high-level abstractions.
- III is a general transformational method for designing and optimizing programs, for programs written using different language features—loops and arrays, set expressions, recursive functions, logic rules, and objects

[1] A Google Scholar search of "incremental computation", including the quotes, performed on Dec. 11, 2023, returned "About 485 results" for the period up until 1992, which includes the period covered by [110], and returned "About 8,360 results" for the period up until the present.

PEPM '24, January 16, 2024, London, UK

ACM ISBN 979-8-4007-0487-1/24/01
https://doi.org/10.1145/3635800.3637447

and classes. The method is particularly powerful for high-level data abstractions using sets, as in database programming; control abstractions using recursion, as in functional programming; both sets and recursion, as in logic programming; and module abstraction using objects.

We will also see that incremental computation is closely related to and intertwined with partial evaluation, and that optimization by incrementalization corresponds to integration by differentiation.

At a meta-level, with historical contexts and for future directions, we stress the power of high-level data, control, and module abstractions in enabling systematic analysis and transformations for incrementalization, leading to new and better algorithms, programs, and precise complexities. When describing the personal experience of the author, we will write in the first person.

At the same time, systematic design and optimization using incrementalization in turn have helped raise the level of abstractions, enabling the design of a high-level language for distributed algorithms [83, 84] and subsequently a unified semantics for logic rules with unrestricted negation, quantification, and aggregation [78, 80].

For future work, significant further work is needed to put both high-level abstractions and powerful transformation methods into practice, for developing algorithms and generating programs with both correctness and performance guarantees.

2 Incremental Computation: Three Categories of Studies

Despite the vast amount and variety, we organize work on incremental computation into three main categories.

Incremental algorithms. Algorithms for computing particular functions, such as shortest paths, under particular kinds of input changes, such as adding and deleting edges.
This includes algorithms known as dynamic algorithms, online algorithms, and other variants.

Incremental program-evaluation frameworks. Frameworks for evaluating general classes of programs expressed in the framework and handling input changes. This includes frameworks known as memoization, caching, tabling, change propagation, and other variants.

Incremental-algorithm-and-program derivation methods. Methods for deriving algorithms and programs that handle input changes from given algorithms or programs and given kinds of input changes. This includes methods known as finite differencing, strengthening and maintaining loop invariants, incrementalization, and other variants.

This categorization was first developed in 1991 from my year-long study of work in incremental computation. It enabled me to formulate systematic incrementalization [62, 90] for my Ph.D. thesis proposal in May 1992. Now over 30 years later, after an extensive month-long further study of the literature, this categorization is further confirmed.

Thanks to technological advances and extensive work by many people, this new review of the literature has been blessed with tremendous new resources not available over 30 year ago: the Web, Google scholar, free access to earlier literature by major publishers like ACM and Elsevier, and to newly available literature by services like arXiv and the Computer History Museum. Given the vast literature spreading in every dimension, I have tried to include the earliest sources found, a range of examples, and latest overviews.

2.1 Incremental Algorithms

Algorithms are at the core of computation, aiming to compute desired output efficiently from given input. Given certain ways that input can change, incremental algorithms aim to compute desired output efficiently by maintaining and reusing previously computed results. For example, an algorithm for sorting computes sorted output from the given input, whereas an incremental algorithm for sorting when an element can be added to or deleted from the input computes sorted output by storing and updating the previously sorted output.

Example algorithms. Incremental algorithms have been studied since at least the 1960s [110], e.g., for maintaining the shortest distances in a graph when the length of an edge is increased or decreased [99]. The area has grown to include a vast number of algorithms for a wide variety of problems and variants, e.g., incremental parsing [41, 126], incremental attribute evaluation [111, 128], incremental circuit evaluation [4], incremental constraint solving [25, 36], as well as many incremental graph problems and more [31, 48, 97, 118], to reference a few. Although efforts in this category are directed towards particular incremental algorithms, an algorithm may apply to a broad class of problems, e.g., any attribute grammar, any graphs, etc.

Many incremental algorithms, especially those on graphs, are also known as dynamic algorithms, which are called fully dynamic, incremental, or decremental if the algorithm handles both additions and deletions of edges, only additions, or only deletions, respectively [27, 48]. Note that we use the term "incremental dynamic algorithms" to differentiate from uses of "incremental algorithms" for the general category.

A class of incremental algorithms are known as online algorithms, which process input piece-by-piece in the order that the input is fed to the algorithm [9, 118], essentially corresponding to incremental dynamic algorithms. For example, an online algorithm for sorting processes each next element in the input as it comes. Another class of incremental algorithms are known as streaming algorithms, which

process a sequential stream as input but with limited memory [97, 100], and thus can only examine the input in a few passes, typically one pass, and often produce approximate output using what is called a sketch. For example, a streaming algorithm can sort, say, the 10 smallest elements but not the entire input.

A wide range of efficiency and other measures, with tradeoffs, and complexity models, are used for characterizing incremental algorithms. General incremental algorithms and dynamic algorithms mainly aim to minimize the times to maintain needed information and return desired output. Online algorithms also aim to be competitive in performance against the case that the entire input is given at the start. Streaming algorithms aim for small space and consider also approximation ratio when output is approximated, for example, for counting the number of distinct elements when they do not fit in memory.

2.2 Incremental Program-Evaluation Frameworks

Rather than manually developing incremental algorithms for each particular problem, an incremental execution framework allows non-incremental programs that are expressed in the framework to run directly and automatically handles changes supported by the framework to achieve incremental computation. For example, a framework can support caching the return values of certain functions and automatically reuse the cached results when such functions are called again on the same arguments.

Example frameworks. Incremental computation appeared in the 1960s as a framework using LISP for computing function applications as partial information about the input is provided [92, 93]. "Memo" functions and machine learning appeared in Nature April 1968 as a framework for caching and reusing the results of functions [98]. Numerous frameworks have since been proposed, e.g., incremental attribute evaluation frameworks performing change propagation [111], function caching with improved cache replacement [109], the INC language with change detailing network [129], incremental reduction in lambda calculus [1, 34], logic rule engines with tabling [20, 121] and incremental tabling [114], combining change propagation and memoization [3, 50], combining them also with demand [47, 50], and a recent more extensive framework [49], to mention a few.

Caching—also known as memoization and tabling—is the key idea in all these frameworks to enable reuse of previously computed results. On top of it, with previously computed results saved, change propagation aims to compute only results that depend on the changes in input. Additional improvements to these techniques and combinations of them, e.g., [3, 47, 49, 50], can enable more refined reuse in more specialized cases, all by expressing the given problem using the mechanisms supported by the framework, without manually writing a particular incremental program for each particular problem.

When such frameworks are used, no explicit incremental version of an application program is derived and run by a standard evaluator. Also, any input change to an application program is captured as a form that the framework can handle, which is limited for each framework. As a result, these solutions to the incremental computation problem for particular application functions and changes are not readily comparable with explicitly developed incremental algorithms and programs for those functions and changes.

2.3 Incremental-Algorithm-and-Program Derivation Methods

Instead of developing incremental algorithms for each problem in an ad hoc way or an incremental evaluation framework for problems expressed in that framework, incremental algorithm and program derivation methods aim to derive explicit incremental algorithms and programs from non-incremental programs and given input change operations using semantics-preserving program transformations. For example, a method can start with a sorting program and an input change operation that adds a new input element and derive an algorithm and program that inserts the new element into the previously sorted result.

Example methods. Early proponents of high-level languages supporting recursive functions [12, 94] and sets [29, 115] pioneered the study of efficient language implementations [22, 26, 28, 95, 117], especially transformations to avoid repeated expensive computations in recursion [11, 96, 116] and in iteration [30, 35, 101]. Significant effort has been devoted to such transformation methods for incremental computation, e.g., transformation techniques for tabulation [5, 96, 116], iterator inversion for converting from a batch to an incremental algorithm in VERS2 [30], finite differencing of set expressions in SETL [101, 102, 106], deriving incremental view maintenance in databases [2, 18, 51, 60, 61, 103], promotion and accumulation [6, 7], differentiation of functional programs in KIDS [119, 120], incrementalization for recursive functions [66], generating incremental object queries [68, 81, 113], static differentiation for lambda calculus [17, 42], and incrementalizing graph algorithms [32].

Transformations for recursive functions, e.g., [5, 6, 11, 66, 120], can be general just as recursive functions are, but are also limited by the structure of recursion. The generality can be seen from the wide variety of transformations on recursive functions, e.g., [33, 108]. The limitation is especially notable on list, the main data type used in recursive functions, because list elements must be traversed from head to tail in a linear order, which makes it exceedingly complex and inefficient to access an element in the middle of the list, not to mention more complex nested lists. Overall, it is challenging to develop systematic and automatic transformations that are general [8, 63, 119].

Transformations for set queries, e.g., [18, 30, 35, 60, 68, 91, 101, 103, 106, 113], can be systematic and automatic, but unlike recursive functions, these queries are not general, i.e., Turing-complete. Basically, fixed but powerful rules can be developed for transforming each kind of high-level operation on sets into more efficient incremental operations when the sets are updated. In particular, Paige's finite differencing of set expressions [106] led to new and faster algorithms for many challenging problems, e.g., attribute closure [105] in database design, partition refinement [107] for model checking, tree pattern matching [16] for program transformation, DFA minimization [59] and regular expression to DFA [19] conversion in automata theory, copy elimination [46] for compiler optimization, and more.

3 Essence of Incremental Computation

The three categories of incremental computation are clearly different from each other—particular algorithms vs. general program-evaluation frameworks vs. derivation methods for algorithms and programs. What is the essence of incremental computation in all of them? Additionally, how is incremental computation related to partial evaluation?

In fact, it is understanding this essence and following the transformational approach for partial evaluation that enabled the systematic method for incrementalization discussed in Section 4.

3.1 Incrementalization

We first define incremental programs before discussing incrementalization as the essence of incremental computation.

Given a program f and an operation $\oplus$, a program f' is called an incremental version of f under $\oplus$ if f' computes $f(x \oplus y)$ efficiently by using $f(x)$. Precisely,

$$f(x) = r \implies f'(x, y, r) = f(x \oplus y) \tag{1}$$

That is, if the result of $f(x)$ is r, then f' can use x, y, and r in computing the result of $f(x \oplus y)$.

Note that f and $\oplus$ are just two functions. An input x to f can have any structure, e.g., a tuple $(x_1, ..., x_k)$. So can a value of parameter y. Operation $\oplus$ can be any function that takes an old input x to f and a value of parameter y and returns a new input $x \oplus y$ to f. Note also that just as $x \oplus y$ captures how the input changes from the previous input x, $f'(x, y, r)$ captures how the output changes from the previous output r.

Given a program f and an operation $\oplus$, incrementalization is the problem of finding an incremental version f' of f under $\oplus$.

Incrementalization is the essence of incremental computation. To see this, we examine how each of the three categories in Section 2 achieves incrementalization.

- Incremental algorithms. It is easy to see that an explicit incremental algorithm corresponds to an incremental version f_0' of a particular function f_0 under a particular kind of input change operation $\oplus_0$:

  $$f_0(x) = r \implies f_0'(x, y, r) = f_0(x \oplus_0 y)$$

  For examples, f_0 takes an input list x and computes a sorted list r, $\oplus$ takes a list x and a new element y and returns a new list with element y added to x, and f_0' inserts y into r in the right place instead of sorting the new list from scratch.
  Note that function f_0 and thus f_0' may return values that have any structure, e.g., a map mapping each pair (u, v) of vertices in a graph to the shortest distance from u to v. In general, function f_0' may be computed in two steps: (1) incrementally maintain appropriate values when the input is changed and (2) retrieve a new return value when the value is used.
  Also, operation $\oplus$ may express different kinds of input changes, e.g., adding or deleting an edge (u, v) to a set e of graph edges—as for fully dynamic graph algorithms [27, 48]—depending on the value of a parameter tag, i.e., to be precise, $\oplus$ takes e and (u, v, tag) and returns $e \cup \{(u, v)\}$ if $tag =$ `'add'` and $e - \{(u, v)\}$ if $tag =$ `'del'`.
- Incremental program-evaluation frameworks. An incremental evaluation framework corresponds to an incremental version *eval'* of an evaluator *eval* under an input change operation $\oplus$, where the input x to *eval* is a pair: a function f and an input $data$ to f. Precisely,

  $$eval(f, data) = r \implies$$
  $$eval'((f, data), y, r) = eval((f, data) \oplus y)$$

  That is, *eval'* is in itself an incremental algorithm and program, handling a certain kind of change to its input. For example, a function caching framework can incrementally execute any program written with caching directives, by capturing all changes as changed functions and arguments in function calls, caching results of certain calls, and reusing cached results for calls with same, or unchanged, functions and arguments.
  For another example, an incremental attribute evaluation framework [111] can solve any tree analysis problems specified using supported attribute equations, by capturing all changes to the input tree as subtree replacements that it handles and running a specific incremental change-propagation attribute evaluation algorithm.
  Note that an input change operation for *eval* can in general change both f and $data$, e.g., in [1, 34, 55], but most incremental evaluation frameworks handle only changes to $data$, e.g., [3, 47, 49, 50, 111], especially frameworks for compiled languages.
- Incremental-algorithm-and-program derivation methods. An incremental algorithm-and-program derivation method is simply a method *inc* for deriving

an incremental version f' of f under $\oplus$, given f and $\oplus$:

$$inc(f, \oplus) = f'$$

where f, $\oplus$, and f' together satisfy (1). For example, finite differencing of set expressions [101, 106] can automatically derive incremental maintenance of complex set expressions under various set update operations by using a collection of finite differencing rules—each for a kind of set expression and a few update operations—and using the chain rule to handle nested expressions. It has been used for optimizing bodies of loops in set languages [101, 106] and also efficient database view maintenance [61, 103].
Note that f and $\oplus$ can be any functions or operations written in the language that the method applies to. Higher-level or more limited languages allow more systematic and automatic derivations, while lower-level and more general languages make the transformations more challenging.

Thus, works in all three categories of incremental computation aim to obtain incremental algorithms or programs for computing a function incrementally under an input change operation. This is exactly the essence of incremental computation, and is what we call incrementalization.

Each work differs in the particular f and $\oplus$ considered, including different evaluators as f, and the language in which f and $\oplus$ are written.

The challenge in all cases is how to achieve better incrementalization more systematically for more general problems and languages, and ideally achieve even the best incrementalization possible fully automatically.

3.2 Incremental Computation vs. Partial Evaluation

An area closely related to and intertwined with incremental computation is partial evaluation [57].

Partial evaluation, also called specialization, considers a program f whose input has two parts, say (x_1, x_2), and aims to evaluate f as much as possible on a given value x_1 of the first part, yielding a partially evaluated program f_{x_1}:

$$PE(f, x_1) = f_{x_1}$$

so that when given a value x_2 for the second part, $f_{x_1}(x_2)$ can compute the result of $f(x_1, x_2)$ more efficiently.

Partial evaluation is especially important and interesting in connecting compilers with interpreters. Consider an interpreter $interp$ that takes program $prog$ and data $data$ as input. We can see the following:

1. $PE(interp, prog)$ yields $interp_{prog}$, which is like a compiled program for $prog$, because $interp_{prog}(data)$ can compute the result of $interp(prog, data)$ more efficiently.
2. $PE(PE, interp)$ yields PE_{interp}, which is like a compiler, because $PE_{interp}(prog)$ can compute the result $interp_{prog}$ of $PE(interp, prog)$ more efficiently..
3. $PE(PE, PE)$ yields PE_{PE}, which is like a compiler generator, because $PE_{PE}(interp)$ can compute the result PE_{interp} of $PE(PE, interp)$ more efficiently..

Interestingly, incremental computation and partial evaluation were related physically in 1993, at the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), as two tutorials with those names, even with articles in the proceedings [23, 110]. I think it was the first time that POPL had any tutorials and many years before it had any again.

It is also interesting that the earliest work I could find that proposes a general form of partial evaluation calls it "incremental computer" and "incremental computation" [92, 93], although "partial evaluation" is also mentioned for one case. On the other hand, the earliest work I could find on a form of incremental computation in database by monitoring changes and avoiding recomputation calls it "partial evaluation" [10], and no "incremental" or "propagation" was mentioned.

There are actually interesting exact relationships between incremental computation and partial evaluation, going both ways;

- On the one hand, partial evaluation can be viewed as a special case of incremental computation.
  Consider any program f whose input has two parts (x_1, x_2). Define $\oplus$ as: $(x_1, x_2) \oplus y = (x_1, y)$ Then an incremental program will aim to reuse computed values on the x_1 part of the input as much as possible and thus compute with the new parameter y as the changed part more efficiently. This is exactly what partial evaluation aims to do.
- On the other hand, incremental computation can be viewed as a special case of generalized partial evaluation [37, 38].
  Generalized partial evaluation aims to evaluate as much as possible on any given information about program f and input x, not limited to x being a given value of one part of the input. For a simple example, the information may be $x > 5$, so any computation in f in a branch with condition, say, $x < 3$ can be removed.
  With that, consider the given information $x = x_{prev} \oplus y$ and $f(x_{prev}) = r$. Using this information to compute efficiently is exactly what incremental computation does.

Note that generalized partial evaluation essentially aims to optimize any program to the best using any given information, which is an undecidable problem in general. However, powerful methods can be developed for special kinds of input information such as that for partial evaluation and

incremental computation. In fact, systematic incrementalization described in Section 4 uses specialization of $f(x)$ in a specific context, which can also be regarded as a special case of generalized partial evaluation.

4 Systematic Incrementalization---Using Previous Result, Intermediate Results, and Auxiliary Values

We give a highly distilled overview of a general and systematic method for incrementalization, first developed in my Ph.D. work [64]. The method is general in that it applies to any language for writing f and $\oplus$. It is systematic in that it consists of systematic program analysis and transformations.

Note, however, that higher-level languages allow easier and better analysis and transformations, and thus better incrementalization, enabling more drastic optimization by incrementalization discussed in Section 5.

The transformational approach was inspired by the use of systematic analysis and transformations in partial evaluation [57], and made possible by focusing on the problem of incrementalization, not general program improvement using general unfold-fold transformations that require "eureka" [11].

We conclude the section by relating incrementalization to differentiation and furthermore to integration in calculus.

4.1 A Systematic Transformational Method for Incrementalization

The overall method was developed incrementally, by solving three key and increasingly harder problems, but with each next problem better understood after the previous problems were solved, and also more easily solved by reducing to the previous problems, giving increasingly greater incrementality.

The key idea for solving all three problems is exactly to analyze and transform $f(x \oplus y)$—expanding it and separating computations that depend on x from those that depend on y, and then storing and reusing values that were computed on x.

P1. Exploiting the previous result. The problem is to use the return value r of $f(x)$ in computing $f(x \oplus y)$.
The most straightforward use is: after transforming $f(x \oplus y)$ to separate computations on x and on y, if there is a computation on x that is exactly $f(x)$, then replace it with r.
More powerful uses are by exploiting data structures and control structures in $f(x)$:
If r is a structured data, e.g., a tuple whose first component is $f_1(x)$, then a computation $f_1(x)$ in $f(x \oplus y)$ can be replaced with a retrieval from r, e.g., $1st(r)$.
If $f_1(x)$ is computed only inside a branch with a condition, e.g., $x \neq null$, in $f(x)$, then replacement of $f_1(x)$ in $f(x \oplus y)$ must be in a branch where $x \neq null$ holds.

P2. Caching intermediate results. The problem is to use helpful values computed while computing $f(x)$, not just the return value. But, which values and how to use them?
With P1 solved, the conceptually simplest solution is cache-and-prune: (1) transform f to cache all intermediate values in the return value. yielding f_{cache}, (2) use P1 to incrementalize f_{cache}, yielding f'_{cache}, and (3) prune f'_{cache} and f_{cache} to retain only values needed for obtaining the original return value.
Alternatively, selective caching can (1) use and extend solutions to P1 to identify, in $f(x \oplus y)$, computations on x that are intermediate computations in $f(x)$, (2) transform f to cache such computation results in the return value, and (3) use P1 to incrementalize the transformed program.

P3. Discovering auxiliary information. The problem is to use also values not computed in $f(x)$ at all but that can help in computing $f(x \oplus y)$. But, where to find such values, and how to use them?
With P1 and P2 solved, the simplest solution is to (A) use and extend P1 to identify, in $f(x \oplus y)$, computations on x that are not computed at all in $f(x)$ as candidate auxiliary values, and (B) use and extend P2 to extend f to also return the candidate auxiliary values.
This process may repeat, because computing the auxiliary values after $\oplus$ may need other auxiliary values. Also, because these values are not computed in the original program, cost analysis is needed to use only auxiliary values that help incremental computation.
This enables the use of a general class of auxiliary information that can be found systematically.

The resulting overall method is modular. The method can be fully automated because the analysis and transformations used can be conservative and fully automatic [130].

The method is described in more detail in an overview paper [66] and in the last chapter of a book [67, Chapter 7], with references to detailed analyses and transformations for P1 [62, 90], P2 [86, 89], and P3 [85, 87]. Thanks to Neil Jones, Michel Sintzoff, and others for repeated suggestions and encouragement for me to write the book, and to Olivier Danvy for inviting me to write the overview article and giving me detailed comments and suggestions for revisions.

4.2 Incrementalization---Differentiation in Discrete Domains

It is easy to see that incrementalization corresponds to differentiation in calculus, except that incrementalization takes place in discrete domains as opposed to the continuous domain. We even use the same notation f' for both the incremental version of f from incrementalization and the derivative of f from differentiation.

Both study changes in the output of functions given changes in the input. While differentiation yields the derivative of a function where input changes are infinitesimal, incrementalization yields an incremental version of a function under an input change operation. Note that in the precise definition of incremental version, $x \oplus y$ captures exactly how the input changes from the previous input x, and $f'(x, y, r)$ captures exactly how the output changes from the previous output r.

There are many correspondences. For example, to incrementalize a function defined by composing two smaller functions, an incremental version of the inner function acts as the input change operation of the outer functions, enabled by caching all intermediate results; this corresponds to the chain rule for differentiation in calculus. For another example, repeating incrementalization for discovered auxiliary values as discussed in Section 4.1 corresponds to computing higher-order derivatives.

In general, however, handling changes in discrete domains makes incrementalization much more difficult than differentiation, because functions must be continuous for differentiation, but functions on discrete domains have mostly holes. Thus significant additional effort is needed for incrementalization. Consider the following simple example. We know that:

$$\text{if } f(x) = x^2, \text{ then its derivative is } f'(x) = 2x$$

But if x can only be integers, even for the smallest change $x \oplus y = x + 1$[2], and given $f(x) = r$, additional care is needed:

- First, we have

  $$f(x \oplus y) = f(x + 1) = (x + 1)^2 = x^2 + 2x + 1,$$
  $$\text{so } f'(x, r) = r + 2x + 1$$

  That is, the increment in output is not $2x$ but $2x + 1$ because the change to x is not infinitesimal.
- Second, maintaining r depends on whether the maintenance is done before or after the update to x:

  $$\text{if before } x \mathrel{+}= 1, \text{ we must do } r \mathrel{+}= 2x + 1$$
  $$\text{if after } x \mathrel{+}= 1, \text{ we must do } r \mathrel{+}= 2x - 1$$

Additional care is needed for other kinds of use, and for more complex computations [67, Section 2.2]. Nevertheless, a general and systematic method can handle these issues correctly and automatically [67, 106], and because of the complexities of the matter, is even more desirable.

Note that tabulating polynomials is a generalization of this example, and incrementalization yields exactly the same algorithm from finite difference [58] used by Babbage's difference engine [43].

[2] y is a dummy unused variable in this example

4.3 Optimization by Incrementalization--- Integration by Differentiation

One of the big surprises at my Ph.D. thesis proposal examination, after I presented my P1 method for systematic incrementalization, was that Anil Nerode said: "You are doing integration by differentiation."

I knew I was doing differentiation, and that was part of the reason I used f' to denote the incremental version. However, I thought integration was the opposite of differentiation. How could I be doing something by doing its opposite?

It took me three years of puzzling, until I was close to finishing my dissertation. I had a neatly derived incremental version of the Fibonacci function under the input change operation of incrementing by 1—it only adds the value of $fib(x - 1)$ to the return value of $f(x)$ and takes O(1) time to maintain both. One day when I was wondering what to do with it, I decided I could add a loop outside the incremental version to compute the original Fibonacci function in linear time instead of the original exponential time. How obvious!

There it dawned on me that this is like integration by differentiation. Most excitingly, it clearly pointed to a general method for optimization by incrementalization. Since then, systematic incrementalization has enabled a systematic method for program design and optimization, discussed in Section 5.

5 Systematic Design and Optimization: Iterate, Incrementalize, Implement

Iterate-Incrementalizer-Implement (III) is a systematic method for design and optimization. We present a vastly distilled overview of III and how it applies to different core language features. The method has three key steps, centered around incrementalization:

- **I1. Iterate:** determine a minimum increment to take repeatedly, iteratively, to arrive at the desired output.
- **I2. Incrementalize:** make expensive operations incremental in each iteration by using and maintaining useful additional values.
- **I3. Implement:** design appropriate data structures for efficiently storing and accessing the values maintained.

Thanks to Tom Rothamel for picking the name III out of a combination of choices I had in my Advanced Programming Languages course in Spring 2003.

The method was first developed for recursive functions, in the order of Steps I2 [85, 89, 90], I1 [72, 73], and I3 [74]. Since then, it has been used extensively in general settings, and has proved to be drastically more powerful when used on high-level abstractions, especially with sets and relations as high-level data abstractions.

In particular, when applied to set expressions extended with fixed-point operations [105], Steps I1, I2, and I3 correspond to what Paige et al called dominated convergence [13,

15], finite differencing [101, 102, 106], and real-time simulation [14, 44, 104], respectively. Interestingly, those were developed in the same order as Steps I2, I1, and I3.

Table 1 summarizes how the III method applies to different language features that provide different data, control, and module abstractions—loops, sets, recursion, rules, and objects—in different programming paradigms. In particular, Step I1 is essential when recursion as high-level control abstraction is used, Step I3 is essential when sets as high-level data abstraction is used, and Step I2 is essential in all cases,

Note that while loops, sets, functions, and rules can express all computations and updates, objects are essential for building large applications with modular components. One can of course program with all these key language features in one language, e.g., [88].

Details of the III method appear in [67], with a respective chapter for each of the main features [67, Chapters 2-6]. Incrementalization, as done in Step I2, is the core for all these features, where the analysis and transformations in Section 4 are based on the semantics and properties of the features used.

Loops with primitives and arrays—imperative programming. For programs written using loops with primitives and arrays, there are no high-level abstractions for either data or control.

- Loops already encode ways to iterate, and primitives and arrays already have direct mappings to hardware for implementation. So there is little to do for Steps I1 and I3 but to adopt those.
- Step I2 makes expensive computations on primitives and arrays in loop bodies incremental, using incrementalization exploiting properties of primitive operations and aggregate array computations.
  Note that variables holding the values of expensive computations in loops automatically form loop invariants.

Examples of these transformations include classical strength reduction that replaces multiplications with additions [21, 24] for compiler optimization, more general incrementalization for primitives [56, 65] for hardware design, incrementalizing aggregate array computations [71, 82] for, e.g., image processing, and incrementalizing more or different kinds of aggregate array computations, e.g., [40, 127], including for probabilistic inference [127].

Note, however, that if the given ways to iterate and implement do not lead to an efficient program, it is generally too difficult to find better ways to do those, because doing so requires understanding what the loops are computing at a higher level, which is an undecidable problem in general. This is why we advocate higher-level data abstractions in problem specifications when lower-level details are unnecessary.

Set expressions—database programming. For programs that use expressions over sets, which provide high-level data abstractions that must be mapped to low-level data structures, Steps I2 and I3 are essential. The programs may still use loops or use fixed-point operations over sets. Note that relations as in relational databases are just sets of tuples,

- Fixed-point operations, if used, are first transformed into while-loops. Here, Step I1 simply chooses to iterate at the minimum increment to a set, i.e, adding or removing a single element.
- Step I2 transforms expensive set expressions in loop bodies into incremental updates, using auxiliary maps as needed. Set expressions are so high-level, that a set of rules for transforming particular kinds of expressions suffices for excellent results [103, 106]. For example, for set union $u = s \cup t$ under change $s \cup= \{y\}$, the rule gives the maintenance $if\ y \notin t : u \cup= \{y\}$.
  A systematic method can also derive such rules automatically [91], following P1-P3 for incrementalization in Section 4.
- Step I3 designs a combination of linked lists, arrays, and/or hash tables for all sets, so that each element-wise operation on a set can be done in constant time. For example, for various graph traversal algorithms, this yields adjacency list representation.

These transformations have enabled new and better algorithms to be developed, including those referenced in Section 2 and more, e.g., solving regular tree grammar based constraints [69], parametric regular path queries [70], and alias analysis [45], as well as efficient implementation of tuple-pattern based retrievals [112] that translate pseudocode algorithms into efficient C++ implementations.

Note, however, that writing appropriate fixed-point expressions, even though they higher-level than writing while-loops, is still non-trivial, compared with writing logic rules with recursion, which is even higher level.

Recursive functions—functional programming. For programs that use recursive functions, which provide high-level control abstractions that must be transformed to iterations, Steps I1 and I2 are essential.

- Step I1 determines a minimum increment for iteration, by selecting arguments of a recursive call that change minimally from the given function parameters, and taking the opposite of the change. For example, for the Fibonacci function $fib(n)$ whose definition contains calls $fib(n-1)$ and $fib(n-2)$, the increment for iteration is $n \mathrel{+}= 1$.
- Step I2 transforms recursive functions into incremental versions, which are possibly also recursive, exactly by following P1-P3 for incrementalization in Section 4.
- Step I3 selects recursive or indexed data structures, i.e., trees or arrays, to store results of function calls:

Table 1. III method applied to different language features and abstractions

Language features	High-level abstractions	Applying III steps
loops with primitives and arrays	none	I2
set expressions	data	I2, I3
recursive functions	control	I1, I2
logic rules	data, control	I1, I2, I3
objects with fields and methods	module	I2

the latter if the arguments of function calls can take arbitrary values, and the former otherwise.

These transformations have enabled the systematic derivation of dynamic programming [72, 75] and transformation of recursion to iteration [73], with interesting results even on the well-known smallest functions: factorial, Fibonacci, and Ackermann [67, Chapter 4].

Note, however, that lists and trees traversed in fixed order by recursive functions are often unnecessarily lower-level and thus limiting, compared with logic rules with recursion over sets.

Logic rules—logic programming. Logic rules provide both high-level data abstractions and high-level control abstractions, because predicates are simply relations and thus sets of tuples, and predicates can be defined recursively. All three Steps I1–I3 are essential for efficient implementations.

- Step I1 first transforms rules into fixed-point operations over sets following the semantics of the rules, and then into while-loops that add one inferred fact at a time.
- Step I2 transforms expensive set expressions in loops, essentially as for set expressions.
- Step I3 designs data structures for implementing set operations, essentially as for set expressions.

These transformations have been developed for Datalog [76, 77] supporting also on-demand queries [122, 123] and various extensions, e.g., [125], especially including precise time and space complexity guarantees. They have led to new and improved algorithms or improved complexities, e.g., for model checking [52], secure information flow [53], trust management [54], and pointer analysis [124].

Objects with fields and methods—object-oriented programming. Finally objects provide module abstraction, hiding both data and control, in representations of fields and implementations of methods, respectively. Performing Step I2 across modular components is essential, because expensive expressions may depend on data hidden in different objects.

- Objects encapsulate both control structures, whether low-level loops or high-level recursion, and data representations, whether low-level arrays or high-level sets. Steps I1 and I3 are generally the same as already discussed, having little to do for loops and arrays and transforming within objects for recursion and sets that are encapsulated.
- Step I2 transforms expensive computations that use data within the same object as already discussed, and transforms across objects for expensive computations using data encapsulated and updated in different objects [81]. The latter follows the principle that each object hides its own data and provides methods for others to observe needed information, and get notified when changes happen so as to perform incremental maintenance.

Objects help organize complex applications, and incrementalization allows high-level queries to be used and be optimized, as discussed for a large electronic health record system [67, Chapter 6.4] and a robot game [67, Chapter 6.5]. Incrementalization across object abstraction [81] actually yields the well-known widely used observer pattern [39]. For complex queries over nested objects and sets, a neat translation into queries over sets of pairs allows incremental queries to be generated fully automatically [68, 113].

Interestingly, studying concurrent and distributed objects has led to a high-level languages for distributed algorithms and a unified semantics for logic rules as discussed in Section 6.

6 Conclusion---Raising the Level of Abstractions

We have discussed that incrementalization is the essence of incremental computation and, even more importantly, the core of a systematic method for program design and optimization, including algorithm design and optimization.

High-level abstractions. The key meta-level observation is that high-level abstractions enable drastically easier and more powerful optimization. In particular:

- High-level data abstractions using sets, including relations and predicates, enable high-level declarative queries, leading to not only efficient incremental maintenance of sophisticated database views but also new and better algorithms for challenging problems in complex applications.
  They also obviate unnecessary uses of error-prone loops and tedious recursion over low-level data.

- High-level control abstractions using recursion, especially when used with high-level data abstraction, allow sophisticated analysis and queries over complex transitive relations to be expressed clearly, and be implemented efficiently using incrementalization for fixed-point computations.
  They furthermore enable automatic calculation of precise complexity guarantees that is impossible otherwise.
- High-level module abstractions using objects, by encapsulating both data and control so that all queries and updates on the data are in the same object, help make the analysis and transformations more localized. They also force objects that interact with each other to follow established patterns for general extensibility.

Most interestingly, because objects can be concurrent and distributed, studying distributed algorithms has led to the creation of a powerful language, DistAlgo, for specifying and programming distributed algorithms at a high level [83, 84], extending the Python language and compiler. In particular, expressing distributed algorithms using high-level queries of message histories reveals substantial need of logic quantifications; and systematic incrementalization and optimization for quantifications [83, 84] helped enable the support of high-level queries in DistAlgo.

Most unexpectedly, efficient implementations of quantifications further led to the discovery and development of a unified semantics for logic rules with unrestricted negation, quantification, and aggregation [78, 80], including knowledge units for using the semantics at scale [79]. The new semantics not only unifies disagreeing previous semantics but also is much simpler and exponentially more expressive. Experiments with examples implemented with optimization by incrementalization also show superior performance over the best well-known systems when they can compute correct answers, and that on some examples, none of those systems can compute correct answers [80].

Limitation and future work. Of course systematic incrementalization will not be able to derive all best algorithms, because it is in general an undecidable problem. However, by making the design systematic and automated for everything that can be automated, designers and developers can focus on truly creative things.

Overall, tremendous effort is needed to support high-level abstractions in widely-used languages, and implement powerful analysis and transformations in real-world compilers. The goal is to support rapidly developing algorithms and generating programs with both correctness and efficiency guarantees.

Additional technical questions include: Can we raise the level of abstraction even higher and generate even better algorithms and programs in better ways? In particular, can we program with higher-level constraints and derive efficient programs that find desired solutions with complexity guarantees? Also, can distributed algorithms and programs be derived systematically from desired global properties?

Practical implementation questions include: How can we maintain compiler extensions and optimizations when program representations in the compiler keep changing? Can we make practical compiler construction for rich languages much easier, supporting logic rules for analysis, and transformation rules for optimizations, in an overall powerful language?

Acknowledgment. I am deeply grateful for all the advice and comments by many colleagues on our work on incrementalization, but especially Anil Nerode, for amazing insight and guidance for over 30 years; Cordell Green, Neil Jones, Jack Schwartz, and Michel Sintzoff, for great encouragement for my book; Jon Barwise, Olivier Danvy, Robert Dewar, Fritz Henglein, and Moshe Vardi, for special encouragement for our work; Bob Paige, Tom Reps, Doug Smith, and Tim Teitelbaum, for excellent related work and advice; and Scott Stoller, for all the help and collaboration for over 30 years. Many thanks to new brilliant work by all my Ph.D. students, especially Tom Rothamel, Tuncay Tekle, and Bo Lin, for amazingly neat incremental object and set queries, demand-driven Datalog queries, and DistAlgo compiler and optimization, respectively. Special thanks to Paul McJones for creating critical historical software archives at the Computer History Museum. Additional thanks to Fritz Henglein, Tom Rothamel, Scott Stoller, Tim Teitelbaum, and Yi Tong for helpful and detailed comments on drafts of this article.

This work was supported in part by NSF under grant CCF-1954837. This work was done in part while the author was visiting the Simons Institute for the Theory of Computing. Thanks to the Simons Institute for the Theory of Computing and organizers of the program on Logic and Algorithms in Database Theory and AI for bringing wonderful diverse communities together.

References

[1] Martin Abadi, Butler Lampson, and Jean-Jacques Lévy. 1996. Analysis and caching of dependencies. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*. 83–91.

[2] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries using Relative Partial Aggregate Indexes (RPAI). In *Proceedings of the 2022 International Conference on Management of Data*. 136–149.

[3] Umut A. Acar. 2009. Self-adjusting Computation: (an Overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM Press, 1–6.

[4] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. 1990. Incremental Evaluation of Computational Circuits. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM Press, 32–42.

[5] Richard S. Bird. 1980. Tabulation Techniques for Recursive Programs. *Comput. Surveys* 12, 4 (1980), 403–417.

[6] Richard S. Bird. 1984. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems* 6, 4 (1984), 487–504.

[7] Richard S. Bird. 1985. Addendum: The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems* 7, 3 (1985), 490–492.

[8] Lee Blaine and Allen Goldberg. 1991. DTRE—A Semi-Automatic Transformation System. In *Constructing Programs from Specifications*, B. Möller (Ed.). North-Holland, 165–203.

[9] Allan Borodin and Ran El-Yaniv. 2005. *Online computation and competitive analysis* (2 ed.). Cambridge University Press.

[10] O Peter Buneman and Eric K Clemons. 1979. Efficiently monitoring relational databases. *ACM Transactions on Database Systems (TODS)* 4, 3 (1979), 368–382.

[11] R. M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (1977), 44–67.

[12] Rod M Burstall and Robin J Popplestone. 1968. POP-2 reference manual. *Machine Intelligence* 2, 205-246 (1968), 1. https://www.cs.utexas.edu/~moore/best-ideas/pltp/POP-2-Reference-Manual.pdf.

[13] Jiazhen Cai. 1987. *Fixed Point Computation and Transformational Programming*. Ph. D. Dissertation. Department of Computer Science, Rutgers, The State University of New Jersey.

[14] Jiazhen Cai, Philippe Facon, Fritz Henglein, Robert Paige, and Edmond Schonberg. 1991. Type Analysis and Data Structure Selection. In *Constructing Programs from Specifications*, Bernhard Möller (Ed.). North-Holland, 126–164.

[15] Jiazhen Cai and Robert Paige. 1988. Program Derivation by Fixed Point Computation. *Science of Computer Programming* 11 (1988), 197–261.

[16] Jiazhen Cai, Robert Paige, and Robert Tarjan. 1992. More Efficient Bottom-Up Multi-Pattern Matching in Trees. *Theoretical Computer Science* 106, 1 (1992), 21–60. https://core.ac.uk/download/pdf/82558972.pdf

[17] Yufei Cai, Paolo G Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 145–155.

[18] Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*. Morgan Kaufmann, 577–589.

[19] Chia-Hsiang Chang and Robert Paige. 1997. From Regular Expressions to DFA's Using Compressed NFA's. *Theoretical Computer Science* 178, 1–2 (1997), 1–36.

[20] Weidong Chen and David S. Warren. 1996. Tabled Evaluation with Delaying for General Logic Programs. *J. ACM* 43, 1 (1996), 20–74. https://doi.org/10.1145/227595.227597

[21] John Cocke and Ken Kennedy. 1977. An Algorithm for Reduction of Operator Strength. *Commun. ACM* 20, 11 (1977), 850–856.

[22] John Cocke and J. T. Schwartz. 1968-1969; second revised version April 1970. *Programming Languages and Their Compilers: Preliminary Notes* (2 ed.). Courant Institute of Mathematical Sciences, New York University. https://www.softwarepreservation.org/projects/SETL/precursors/Cocke_Schwartz-Programming_Languages_and_Their_Compilers-1970.pdf.

[23] Charles Consel and Olivier Danvy. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 493–501.

[24] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction. *ACM Transactions on Programming Languages and Systems* 23, 5 (2001), 603–625. https://doi.org/10.1145/504709.504710

[25] Neil T Dantam, Zachary K Kingston, Swarat Chaudhuri, and Lydia E Kavraki. 2018. An incremental constraint-based framework for task and motion planning. *The International Journal of Robotics Research* 37, 10 (2018), 1134–1151.

[26] John Darlington and R. M. Burstall. 1973. A System Which Automatically Improves Programs. In *Proceedings of the 3nd International Joint Conference on Artificial Intelligence*. 479–485.

[27] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. 2009. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook* (2 ed.). Chapman and Hall/CRC, Chapter 9, 9–1–9–28.

[28] Jay Earley. 1971. Toward an understanding of data structures. *Commun. ACM* 14, 10 (1971), 617–627.

[29] Jay Earley. 1974. High Level Operations in Automatic Programming. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. 34–42. https://doi.org/10.1145/800233.807043

[30] Jay Earley. 1976. High Level Iterators and a Method for Automatically Designing Data Structure Representation. *Journal of Computer Languages* 1 (1976), 321–342.

[31] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[32] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2021. Incrementalizing graph algorithms. In *Proceedings of the 2021 International Conference on Management of Data*. 459–471.

[33] Martin S. Feather. 1987. A Survey and Classification of Some Program Transformation Approaches and Techniques. In *Program Specification and Transformation*, L. G. L. T. Meertens (Ed.). North-Holland, 165–195.

[34] John Field and Tim Teitelbaum. 1990. Incremental Reduction in the Lambda Calculus. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 307–322.

[35] Amelia C. Fong and Jeffrey D. Ullman. 1976. Inductive Variables in Very High Level Languages. In *Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages*. 104–112.

[36] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. 1990. An Incremental Constraint Solver. *Commun. ACM* 33, 1 (1990), 54–63.

[37] Y. Futamura and K. Nogi. 1988. Generalized Partial Evaluation. In *Partial Evaluation and Mixed Computation*, Bines Bjørner, Andrei P. Ershov, and Neil D. Jones (Eds.). North-Holland, 133–151.

[38] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. 1991. Essence of generalized partial computation. *Theoretical Computer Science* 90, 1 (1991), 61–79.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[40] Gautam and S. Rajopadhye. 2006. Simplifying Reductions. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 30–41.

[41] C. Ghezzi and D. Mandrioli. 1979. Incremental Parsing. *ACM Transactions on Programming Languages and Systems* 1, 1 (1979), 58–70.

[42] Paolo G Giarrusso, Yann Régis-Gianas, and Philipp Schuster. 2019. Incremental-Calculus in Cache-Transfer Style: Static Memoization by Program Transformation. In *European Symposium on Programming*. Springer, 553–580.

[43] Herman H. Goldstine. 1972. Charles Babbage and His Analytical Engine. In *The Computer from Pascal to von Neumann*. Princeton University Press, 10–26. New printing with new preface in 1993.

[44] Deepak Goyal. 2000. *A Language Theoretic Approach to Algorithms*. Ph. D. Dissertation. Department of Computer Science, New York University.

[45] Deepak Goyal. 2005. Transformational Derivation of an Improved Alias Analysis Algorithm. *Higher-Order and Symbolic Computation* 18, 1–2 (2005), 15–49. https://doi.org/10.1007/s10990-005-7005-6

[46] Deepak Goyal and Robert Paige. 1998. A New Solution to the Hidden Copy Problem. In *Proceedings of the 5th International Symposium on Static Analysis*. Springer, 327–348.

[47] Matthew A Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S Foster. 2014. Adapton: Composable, demand-driven incremental computation. *ACM SIGPLAN Notices* 49, 6 (2014), 156–166.

[48] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2022. Recent advances in fully dynamic graph algorithms–A quick reference guide. *ACM Journal of Experimental Algorithmics* 27 (2022), 1–45.

[49] Georg Hinkel. 2021. *Implicit incremental model analyses and transformations*. The Karlsruhe Series on Software Design and Quality, Vol. 26. KIT Scientific Publishing.

[50] Roger Hoover. 1992. Alphonse: Incremental Computation as a Programming Abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*. 261–272.

[51] Susan Horwitz and Tim Teitelbaum. 1986. Generating Editing Environments Based on Relations and Attributes. *ACM Transactions on Programming Languages and Systems* 8, 4 (1986), 577–608.

[52] Katia Hristova and Yanhong A. Liu. 2006. Improved Algorithm Complexities for Linear Temporal Logic Model Checking of Push Down Systems. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation (LNCS, Vol. 3855)*. Springer, 190–206.

[53] Katia Hristova, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. 2006. Efficient Type Inference for Secure Information Flow. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. ACM Press, 85–94.

[54] Katia Hristova, K. Tuncay Tekle, and Yanhong A. Liu. 2007. Efficient Trust Management Policy Analysis from Rules. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 211–220.

[55] Zezhou Huang and Eugene Wu. 2024. Lightweight Materialization for Fast Dashboards Over Joins. In *Proceedings of the 2024 International Conference on Management of Data*.

[56] Steve D. Johnson, Yanhong A. Liu, and Yuchen Zhang. 2003. A Systematic Incrementalization Technique and Its Application to Hardware Design. *International Journal on Software Tools for Technology Transfer* 4, 2 (2003), 211–223.

[57] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

[58] Charles Jordan. 1965. *Calculus of Finite Differences* (3 ed.). Vol. 33. American Mathematical Society Chelsea Publishing. https://bookstore.ams.org/chel-33

[59] J. P. Keller and Robert Paige. 1995. Program Derivation With Verified Transformations—A Case Study. *Communications on Pure and Applied Mathematics* 48, 9–10 (1995), 1053–1113.

[60] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 87–98.

[61] S. Koenig and R. Paige. 1981. A Transformational Framework for the Automatic Control of Derived Data. In *Proceedings of the 7th International Conference on Very Large Data Bases*. 306–318.

[62] Yanhong Liu and Tim Teitelbaum. 1993. *Deriving Incremental Programs*. Technical Report TR 93-1384. Department of Computer Science, Cornell University.

[63] Yanhong A. Liu. 1995. CACHET: An Interactive, Incremental-Attribution-Based Program Transformation System for Deriving Incremental Programs. In *Proceedings of the 10th IEEE Knowledge-Based Software Engineering Conference*. IEEE CS Press, 19–26.

[64] Yanhong Annie Liu. 1996. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. Ph. D. Dissertation. Department of Computer Science, Cornell University. A slightly revised version appears as Cornell University Computer Science Department Technical Report TR 95-1551, October, 1995. https://hdl.handle.net/1813/7208.

[65] Yanhong A. Liu. 1997. Principled Strength Reduction. In *Algorithmic Languages and Calculi*, Richard Bird and Lambert Meertens (Eds.). Chapman & Hall, 357–381. http://www.cs.stonybrook.edu/~liu/papers/Psr-IFIP97.pdf

[66] Yanhong A. Liu. 2000. Efficiency by Incrementalization: An Introduction. *Higher-Order and Symbolic Computation* 13, 4 (2000), 289–313. http://www.cs.stonybrook.edu/~liu/papers/IncEff-HOSC00.pdf

[67] Yanhong Annie Liu. 2013. *Systematic Program Design: From Clarity To Efficiency*. Cambridge University Press.

[68] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. 2016. Demand-Driven Incremental Object Queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. ACM Press, 228–241. https://doi.org/10.1145/2967973.2968610.

[69] Yanhong A. Liu, Ning Li, and Scott D. Stoller. 2001. Solving Regular Tree Grammar Based Constraints. In *Proceedings of the 8th International Static Analysis Symposium*. Springer, 213–233.

[70] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott Stoller, and Nanjun Hu. 2004. Parametric Regular Path Queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press, 219–230.

[71] Yanhong A. Liu and Scott D. Stoller. 1998. Loop Optimization for Aggregate Array Computations. In *Proceedings of the IEEE 1998 International Conference on Computer Languages*. IEEE CS Press, 262–271. http://www.cs.stonybrook.edu/~liu/papers/Array-ICCL98.pdf

[72] Yanhong A. Liu and Scott D. Stoller. 1999. Dynamic Programming via Static Incrementalization. In *Proceedings of the 8th European Symposium on Programming*. Springer, 288–305.

[73] Yanhong A. Liu and Scott D. Stoller. 2000. From Recursion to Iteration: What Are the Optimizations?. In *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Program Manipulation*. 73–82.

[74] Yanhong A. Liu and Scott D. Stoller. 2002. Program Optimization Using Indexed and Recursive Data Structures. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Program Manipulation*. 108–118.

[75] Yanhong A. Liu and Scott D. Stoller. 2003. Dynamic Programming via Static Incrementalization. *Higher-Order and Symbolic Computation* 16, 1–2 (2003), 37–62. Special issue dedicated to Bob Paige.

[76] Yanhong A. Liu and Scott D. Stoller. 2003. From Datalog Rules to Efficient Programs with Time and Space Guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 172–183.

[77] Yanhong A. Liu and Scott D. Stoller. 2009. From Datalog Rules to Efficient Programs with Time and Space Guarantees. *ACM Transactions on Programming Languages and Systems* 31, 6 (2009), 1–38. https://doi.org/10.1145/1552309.1552311

[78] Yanhong A. Liu and Scott D. Stoller. 2020. Founded Semantics and Constraint Semantics of Logic Rules. *Journal of Logic and Computation* 30, 8 (Dec. 2020), 1609–1638. Also http://arxiv.org/abs/1606.06269.

[79] Yanhong A. Liu and Scott D. Stoller. 2021. Knowledge of Uncertain Worlds: Programming with Logical Constraints. *Journal of Logic and Computation* 31, 1 (Jan. 2021), 193–212. Also https://arxiv.org/abs/1910.10346.

[80] Yanhong A. Liu and Scott D. Stoller. 2022. Recursive Rules with Aggregation: A Simple Unified Semantics. *Journal of Logic and Computation* 32, 8 (Dec. 2022), 1659–1693. https://doi.org/10.1093/logcom/exac072 Also http://arxiv.org/abs/2007.13053.

[81] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni E. Liu. 2005. Incrementalization Across Object Abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 473–486.

[82] Yanhong A. Liu, Scott D. Stoller, Ning Li, and Tom Rothamel. 2005. Optimizing Aggregate Array Computations in Loops. *ACM Transactions on Programming Languages and Systems* 27, 1 (Jan. 2005), 91–125.

[83] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Transactions on Programming Languages and Systems* 39, 3 (May 2017), 12:1–12:41. https://doi.org/10.1145/2994595 Also http://arxiv.org/abs/1412.8461.

[84] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. 395–410.

[85] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. 1996. Discovering Auxiliary Information for Incremental Computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*. 157–170.

[86] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. 1998. Static Caching for Incremental Computation. *ACM Transactions on Programming Languages and Systems* 20, 3 (1998), 546–585. http://www.cs.stonybrook.edu/~liu/papers/Scic-TOPLAS98.pdf

[87] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. 2001. Strengthening Invariants for Efficient Computation. *Science of Computer Programming* 41, 2 (2001), 139–172.

[88] Yanhong A. Liu, Scott D. Stoller, Yi Tong, and Bo Lin. 2023. Integrating logic rules with everything else, seamlessly. *Theory and Practice of Logic Programming* 23, 4 (2023), 678–695–. https://doi.org/10.1017/S1471068423000108 Special issue for selected papers from ICLP 2023. Also http://arXiv.org/abs/2305.19202.

[89] Yanhong A. Liu and Tim Teitelbaum. 1995. Caching Intermediate Results for Program Improvement. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*. 190–201. http://www.cs.stonybrook.edu/~liu/papers/Cir-PEPM95.pdf

[90] Yanhong A. Liu and Tim Teitelbaum. 1995. Systematic Derivation of Incremental Programs. *Science of Computer Programming* 24, 1 (1995), 1–39.

[91] Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. 2006. Core Role-Based Access Control: Efficient Implementations by Transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation*. 112–120.

[92] A. A. Lombardi. 1967. Incremental Computations: The Preliminary Design of a Programming System Which Allows for Incremental Data Assimilation in Open-Ended Man-Computer Information Systems. In *Advances in Computers, Volume 8*, F. L. Alt and M. Rubinoff (Eds.). Academic Press, New York, 247–333.

[93] L. A. Lombardi and B. Raphael. 1964. LISP as the Language for an Incremental Computer. In *The Programming Language LISP: Its Operation and Applications*, Eemund C. Berkeley and Daniel G. Bobrow (Eds.). MIT Press, 204–219. https://dspace.mit.edu/bitstream/handle/1721.1/48305/lispaslanguagefo00lomb.pdf;sequence=1.

[94] John McCarthy. 1959. LISP: A programming system for symbolic manipulations. In *Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*. 1–4. https://dl.acm.org/doi/pdf/10.1145/612201.612243.

[95] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195.

[96] John McCarthy. 1962. *On efficient ways of evaluating certain recursive functions*. Artificial Intelligence Memo No. 32. Massachusetts Institute of Technology. https://www.bitsavers.org/pdf/mit/ai/aim/AIM-032.pdf.

[97] Andrew McGregor. 2014. Graph stream algorithms: A survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[98] Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* 218 (1968), 19–22.

[99] J Murchland. 1967. *The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph*. Technical Report. Technical report, LBS-TNT-26, London Business School, Transport Network ....

[100] S Muthukrishnan. 2005. Data streams: algorithms and applications. *Foundations and Trends® in Theoretical Computer Science* 1, 2 (2005), 117–236.

[101] Bob Paige and J. T. Schwartz. 1977. Expression Continuity and the Formal Differentiation of Algorithms. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*. 58–71.

[102] Robert Paige. 1981. *Formal Differentiation: A Program Synthesis Technique*. UMI Research Press. Revision of PhD dissertation, New York University, 1979.

[103] Robert Paige. 1984. Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization. In *Advances in Database Theory, Volume 2*. Vol. 2. Plenum Press, 171–210.

[104] Robert Paige. 1989. Real-Time Simulation of a Set Machine on a RAM. In *Proceedings of the International Conference on Computing and Information*. Canadian Scholars Press, 69–73.

[105] Robert Paige and Fritz Henglein. 1987. Mechanical Translation of Set Theoretic Problem Specifications into Efficient RAM Code—A Case Study. *Journal of Symbolic Computation* 4, 2 (1987), 207–232.

[106] Robert Paige and Shaye Koenig. 1982. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 402–454.

[107] Robert Paige and Robert Tarjan. 1987. Three Partition Refinement Algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989.

[108] Alberto Pettorossi and M. Proietti. 1996. Rules and Strategies for Transforming Functional and Logic Programs. *Comput. Surveys* 28, 2 (1996), 360–414.

[109] William Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. 315–328.

[110] G. Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*. 502–510.

[111] Thomas Reps, Tim Teitelbaum, and Alan Demers. 1983. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems* 5, 3 (1983), 449–477.

[112] Tom Rothamel and Yanhong A. Liu. 2007. Efficient Implementation of Tuple Pattern Based Retrieval. In *Proceedings of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*. 81–90. https://doi.org/10.1145/1244381.1244394.

[113] Tom Rothamel and Yanhong A. Liu. 2008. Generating Incremental Implementations of Object-Set Queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, 55–66. https://doi.org/10.1145/1449913.1449923.

[114] Diptikalyan Saha and C. R. Ramakrishnan. 2003. Incremental Evaluation of Tabled Logic Programs. In *Proceedings of the 19th International Conference on Logic Programming*. Springer, 392–406. https://doi.org/10.1007/978-3-540-24599-5_27.

[115] J. T. Schwartz. 1970-1971. *Abstract Algorithms And A set-Theoretic Language for Their Expression, Preliminary draft, first part*. Technical Report. Computer Science Department, Courant Institute of Mathematical Sciences, New York University. https://www.softwarepreservation.org/projects/SETL/setl/doc/Schwartz-Abstract_Algorithms-1971.pdf.

[116] J. T. Schwartz. 1975. *Intermediate Result Recording and Other Techniques for Optimizing Recursions and Backtrack Programs*. SETL Newsletter 155. Courant Institute of Mathematical Sciences, New York University.

[117] J. T. Schwartz. 1975. *On the 'Base Form' of Algorithms.* SETL Newsletter 159. Courant Institute of Mathematical Sciences, New York University.

[118] Alexa Megan Sharp. 2007. *Incremental algorithms: Solving problems in a changing world.* Ph. D. Dissertation. Department of Computer Science, Cornell University.

[119] Douglas R. Smith. 1990. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering* 16, 9 (1990), 1024–1043.

[120] Douglas R. Smith. 1991. KIDS—A Knowledge-Based Software Development System. In *Automating Software Design*, Michael R. Lowry and Robert D. McCartney (Eds.). AAAI Press and MIT Press, 483–514.

[121] Hisao Tamaki and Taisuke Sato. 1986. OLD Resolution with Tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*. Springer, 84–98.

[122] K. Tuncay Tekle and Yanhong A. Liu. 2010. Precise Complexity Analysis for Efficient Datalog Queries. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. 35–44. https://doi.org/10.1145/1836089.1836094

[123] K. Tuncay Tekle and Yanhong A. Liu. 2011. More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 661–672. https://doi.org/10.1145/1989323.1989393

[124] K. Tuncay Tekle and Yanhong A. Liu. 2016. Precise Complexity Guarantees for Pointer Analysis via Datalog with Extensions. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 916–932.

[125] K. Tuncay Tekle and Yanhong A. Liu. 2019. Extended Magic for Negation: Efficient Demand-Driven Evaluation of Stratified Datalog with Precise Complexity Guarantees. In *Proceedings of the 35th International Conference on Logic Programming (Technical Communications)*. Open Publishing Association, 241–254.

[126] Tim A Wagner and Susan L Graham. 1998. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 5 (1998), 980–1013.

[127] Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying dependent reductions in the polyhedral model. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–33. https://dl.acm.org/doi/pdf/10.1145/3434301 ref toplas 95, more general popl06,static compilation implemented in python, own native vs optimized code.

[128] D. Yeh and U. Kastens. 1988. Improvements on an Incremental Evaluation Algorithm for Ordered Attribute Grammars. *SIGPLAN Notices* 23, 12 (1988), 45–50.

[129] Daniel M. Yellin and Robert E. Strom. 1991. INC: A Language for Incremental Computations. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 211–236.

[130] Yuchen Zhang and Yanhong A. Liu. 1998. Automating Derivation of Incremental Programs. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*. 350. http://www.cs.stonybrook.edu/~liu/papers/AutoInc-ICFP98.pdf