1
Introduction

1.1 From clarity to efficiency: systematic program design

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 generally a conflict between clarity and efficiency, because clear specifications usually correspond to straightforward implementations, not at all efficient, whereas efficient implementations are usually sophisticated, not at all clear. What is needed is a general and systematic method to go from clear specifications to efficient implementations.

We give example problems from various application domains and discuss the challenges that lead to the need for a general and systematic method. The example problems are for database queries, hardware design, image processing, string processing, graph analysis, security policy frameworks, program analysis and verification, and mining semi-structured data. The challenges are to ensure correctness and efficiency of developed programs and to reduce costs of development and maintenance.

Example problems and application domains

Database queries. Database queries matter to our everyday life, because databases are used in many important day-to-day applications. Consider an example where data about professors, courses, books, and students are stored, and we want to find all professor-course pairs where the professor uses any of his own books as the textbook for the course and any of his own students as the teaching assistant for the course. It is not hard to see that similar queries can be used to detect fraud in financial databases, find matches between providers and suppliers, and identify

rare correlations in data in general. If you care to know, the example query can be expressed in the dominant database query language, SQL, as follows, where * denotes everything about the matched data:

```
select * from professor, course
where professor.id = course.instructor
      and exists (select * from book
                  where book.author = professor.name and
                        book.name = course.textbook)
      and exists (select * from student
                  where student.advisor = professor.name and
                        student.teaching = course.id)
```

A straightforward computation would iterate through all professors and, for each of them, check each course for whether the professor is the course instructor; further, for each pair of professor and course found, it would check each book for whether the author is the professor and the book is the course textbook, and similarly check each student. This can take time proportional to the number of professors times the number of courses times the sum of the numbers of books and students. An efficient computation can use sophisticated techniques and take only time proportional to the size of the data plus the number of answers. For example, if there are 1,000 each of professors, courses, books, and students, then a straightforward computation can take time on the order of $1,000 \times 1,000 \times (1,000 + 1,000)$, which is 2,000,000,000, whereas an efficient computation takes time on the order of 4,000 plus the number of answers. How to design such an efficient computation?

Hardware design. Hardware design requires efficiently implementing complex operations in computer hardware using operations that already have efficient support in hardware. A good example is the square-root operation. A brute-force way to compute the square root of a given number is to iterate through a range of possible numbers and find the one whose square equals the given number, where the square operation uses multiplication, which has standard support in hardware. An efficient implementation will not use squares or multiplications, but rather a sophisticated combination of additions and *shifts*, that is, doublings and halvings, because the latter have much more efficient support in hardware. How to design such efficient implementations?

Image processing. Image processing has a central problem, which is to process the local neighborhood of every pixel in an image. A simple example is image blurring. It computes the average of the m-by-m neighborhood of every pixel in an n-by-n image. A straightforward way to compute the blurred image is to iterate over each of the n^2 pixels, sum the values of the m^2 pixels in the neighborhood of the pixel, and divide the sum by m^2. This takes time proportional to $n^2 \times m^2$. A well-known efficient algorithm computes the blurred image in time proportional to n^2, by smartly doing only four additions or subtractions in place of summing

over m^2 pixels in the neighborhood of each pixel, regardless of the size m^2 of the neighborhood. How to derive such an efficient algorithm?

String processing. String processing is needed in many applications, from text comparison to biological sequence analysis. A well-known problem is to compute a longest common subsequence of two strings, where a subsequence of a string is just the given string possibly with some elements left out. A straightforward way to compute the solution can be written as a simple recursive function, but takes time proportional to an exponential of the lengths of the two strings in the worst case. An efficient algorithm for this problem tabulates solutions to subproblems appropriately and takes time proportional to the product of the lengths of the two strings in the worst case. How to design such efficient algorithms given recursive functions for straightforward computations?

Graph analysis. Graph analysis underlies analyses of complex interrelated objects. A ubiquitous problem is graph reachability: given a set of edges, each going from one vertex to another, and a set of vertices as sources, compute all vertices reachable from the sources following the edges. Straightforwardly and declaratively, one can state two rules: if a vertex is a source, then it is reachable; if a vertex is reachable, and there is an edge from it to another vertex, then this other vertex is reachable also. An efficient algorithm requires programming a strategy for traversing the graph and a mechanism for recording the visits, so that each edge is visited only once, even if many edges can lead to a same edge and edges can form cycles. How to arrive at such an efficient program from the rules?

Querying complex relationships. Querying about complex relationships, formulated as database queries or graph queries, is essential not only for database and Web applications but also for security policy analysis and enforcement, program analysis and verification, data mining of semi-structured data, and many other applications. In security policy frameworks, complex relationships need to be captured for access control, trust management, and information flow analysis. In program analysis and verification, flow and dependency relations among program segments and values, and transitions among system states, are formulated using many kinds of trees and graphs. For mining semi-structured data, which form trees, segments of trees need to be related along the paths connecting them.

Challenges

The challenges are that, for real-world applications, computer programs need to run correctly and efficiently, be developed quickly, and be easy to maintain, all at low costs. Correctness requires that the developed programs satisfy the problem specifications. Efficiency requires guarantees on fast running times and acceptable space usages for the developed programs. Costs of development and maintenance need to be minimized while achieving desired correctness and efficiency.

Unfortunately, there are trade-offs and thus conflicts among correctness, efficiency, and costs of development and maintenance. The central conflict, as indicated through the example problems just described, is between the clarity and efficiency of computer programs. A straightforward specification of a computation is clear, and thus is not only easier to be sure of correctness but also easier to develop and maintain, but it tends to be extremely inefficient to execute. In contrast, an efficient implementation tends to be sophisticated and not at all clear, and thus is much more difficult to verify for correctness and to develop and maintain.

Of course, there are other challenges besides the trade-offs. In particular, clear specifications, capturing complete requirements of the problems, must be developed, either informally or formally, and efficient implementations, with full details needed for program execution, must be produced at the end, either programmed manually based on informal specifications or generated automatically from formal specifications. We argue here that the ideal way to address all the challenges is through development of clear high-level specifications and automatic generation of efficient low-level implementations from the specifications.

Developing clear specifications. Formal specifications are much harder to develop than informal specifications, but are substantially easier to develop, maintain, and verify than efficient implementations. It would be a significant gain if efficient implementations can be generated automatically by correctness-preserving transformations from formal specifications. How to develop precise and formal specifications? Ideally, we would like to easily and clearly capture informal specifications stated in a natural language in some suitable formal specification language. Practically, we will allow straightforward ways of computations to be specified easily and clearly in high-level programming languages.

Generating efficient implementations. Efficient implementations are much harder to develop than specifications of straightforward computations, but efficient implementations for individual problems are drastically easier to develop than general methods for systematically deriving efficient implementations from specifications. One could perceive many commonalities in solving very different individual problems. What could be a general and systematic method? Such a method should use correctness-preserving transformations starting from specifications. Despite that it must be general, that is, apply to large classes of problems, and be systematic, that is, allow automated support, it must be able to introduce low-level processing strategies and storage mechanisms that are specialized for individual problems.

Overall, we can see that a systematic design method for transforming clear specifications into efficient implementations is central for addressing all the challenges. Clear specifications of straightforward computations, together with correctness-

preserving transformations, make correctness verification substantially easier compared with ad hoc implementations written by hand. Exact understanding of the resulting algorithms and implementations derived using a systematic method is key to providing time and space guarantees. Clear specifications of straightforward computations, plus automatic generation of efficient implementations based on a systematic method, minimize development and maintenance costs.

The question is, then: does such a method exist? If yes, how general and systematic is it? In particular, can it solve all the example problems we have discussed, and what other problems can it solve? If it is not yet general and systematic in the absolute sense, that is, solving all problems, how will it grow? It is not hard to see that, for such a method to exist and to grow, to solve increasingly more problems from different application domains, it must be rooted in rigorous scientific principles.

Exercise 1.1 (Problem description) Describe a computation problem that is interesting to you in any way. Can you describe what it is that should be computed without stating how to compute it? That is, describe what is given as input and what is asked as output, including any restrictions on the input and how the output is related to the input, but not how to go from the input to the output.

1.2 Iterate, incrementalize, and implement

This book describes a general and systematic design and optimization method for transforming clear specifications of straightforward computations into efficient implementations of sophisticated algorithms. The method has three steps: Iterate, Incrementalize, and Implement, called *III* for short.

1. *Step Iterate* determines a minimum input increment operation to take repeatedly, iteratively, to arrive at the desired program output.
2. *Step Incrementalize* makes expensive computations incremental in each iteration by using and maintaining appropriate values from the previous iteration.
3. *Step Implement* designs appropriate data structures for efficiently storing and accessing the values maintained for incremental computation.

We describe the essence of each step separately in what follows, especially how they matter in the wide range of different programming paradigms with different programming abstractions. We first introduce these paradigms and abstractions. We then show that the III method applies uniformly, regardless of the programming paradigms used; this starts with Step Incrementalize, the core of the method. We finally discuss why the three steps together form a general and systematic method for design and optimization.

Programming paradigms and abstractions

We consider five main paradigms of programming: imperative programming, database programming, functional programming, logic programming, and object-oriented programming.

1. *Imperative programming* describes computation as commands that update data storage; at the core are *loops and arrays*—commands for linearly repeated operations and consecutive slots for storing data.
2. *Database programming* expresses computation as queries on collections of records in a database; at the core are *set expressions*—expressions for querying sets of data.
3. *Functional programming* treats computation as evaluation of mathematical functions; at the core are *recursive functions*—functions defined recursively using themselves.
4. *Logic programming* specifies computation as inference of new facts from given rules and facts using deductive reasoning; at the core are *logic rules*—rules for logical inference.
5. *Object-oriented programming* describes computation as objects interacting with each other; at the core are *objects and classes*—instances and their categories for encapsulating combinations of data and operations.

Languages for logic programming, database programming, and functional programming are sometimes called *declarative languages*, which are languages that specify more declaratively what to compute, in contrast to how to compute it.

Regardless of the paradigm, programming requires specifying data and control, that is, what computations manipulate and how computations proceed, and organizing the specifications. This is done at different abstraction levels in different paradigms.

1. Loops and arrays explicitly specify how data is represented and how control flows during computations; they are not high-level abstractions for data or control.
2. Set expressions support computations over sets of records used as *high-level data abstraction*. This eliminates the need to explicitly specify data representations.
3. Recursive functions allow computations to follow recursive function definitions used as *high-level control abstraction*. This eliminates the need to explicitly specify control flows.
4. Logic rules let sets of records be represented as predicates, and let predicates be defined using recursive rules; they provide high-level abstractions for both data and control.

5. Objects and classes provide *high-level module abstraction*, which allows modules or components that encapsulate data and control to be composed to form larger modules.

Uses of these language features are not exclusive of each other and could in fact be supported in a single language; in current practice, however, there is not a well-accepted language that supports them all, but many good languages support subsets of them.

Incrementalize

We discuss Step Incrementalize first because it is the core of the III method. Efficient computations on nontrivial input must proceed repeatedly on input increment. Step Incrementalize makes the computation on each incremented input efficient by storing and reusing values computed on the previous input. Whether problems are specified using loops and arrays, set expressions, recursive functions, logic rules, or objects and classes, it is essential to make repeated expensive computations incremental after the values that they depend on are updated.

More precisely, expensive computations include expensive array computations, set query evaluations, recursive function calls, and logical fact deductions. Variables whose values are defined outside a computation and used in the computation are called *parameters* of the computation, and any operation that sets the value of a parameter is called an *update* to the value of the parameter. The values of parameters of expensive computations may be updated slightly in each iteration of the enclosing computation. The goal of *incrementalization* is to incrementally maintain the results of expensive computations as the values of their parameters are updated in each iteration, by storing and using the results from the previous iteration. This often incurs the need to store and use appropriate additional values and maintain them incrementally as well in each iteration; this reflects a trade-off between running time and space usage.

When objects and classes are used to provide module abstraction for large applications, expensive computations and updates to parameter values may be scattered across classes, and thus we must also incrementalize across objects and classes. This allows incrementalization to be used for scaled-up applications.

Iterate

Step Iterate is the first step of the III method, and determines how computations should proceed. Even though it must be decided before incrementalization, it is actually driven by incrementalization: the goal of incrementalization is to maximize reuse, and therefore a critical decision we make is to minimize the increment in each iteration.

When straightforward computations are specified using loops over array computations or over set expressions, the ways of iterating are already specified by

the loops, and thus Step Iterate is not necessary. The ways of iterating specified by the given loops often lead to desired efficient computations. However, they do not always do so, and determining appropriate ways of iterating that are different from the specified ways can be very difficult because it requires understanding at a higher level what the given loops compute.

When straightforward computations are specified using general recursive functions or logic rules, which provide high-level control abstraction, the ways of iterating are not specified, and thus Step Iterate is essential. In general, there can be many ways of iterating given a recursive specification. Even with the goal of minimizing the increment, there can be multiple ways that are incomparable with each other. Different ways of iterating may impact both the running time of the resulting computation and the space needed for storing values over the iterations.

Implement

Step Implement is the last step of the III method. It designs appropriate data structures. It first analyzes all data accesses needed by incremental computations and then designs appropriate combinations of indexed and linked structures to make the accesses efficient.

When straightforward computations are specified to process data in arrays and recursive data types, it is easy to map these data representations directly on the underlying machine, as indexed consecutive slots and tree-shaped linked structures, respectively, and thus Step Implement is straightforward. These data representations are sufficient for efficient computations for many applications. However, they are not always sufficient, and determining appropriate data representations that are different from the specified ones can be very difficult because it requires understanding at a higher level what the data representations represent.

When straightforward computations are specified using set expressions or logic rules, which use sets and relations as high-level data abstractions, it is essential to determine how sets and relations can be stored in the underlying hardware machines for efficient access. In general, this can be a sophisticated combination of indexed and linked structures. There are also trade-offs between the times needed for different accesses.

A general and systematic method

The III method is general and systematic for at least three reasons: (1) it is based on languages, (2) it applies to a wide range of programming paradigms, and (3) it is the discrete counterpart of differentiation and integration in calculus for continuous domains.

The method is based on languages, meaning that the method consists of analysis and transformations for problems that are specified using the constructs of

languages. This allows the method to apply to large classes of problems specified using the languages, not just some individual problems. It also allows the method to be systematic by formulating the analysis and transformation procedure precisely and completely. We will see that the III method can solve all the example problems discussed earlier and many more that can be specified using the languages we discuss. 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 generate optimal implementations with time and space guarantees.

The method applies to the wide range of programming paradigms discussed earlier in this section, as summarized in Figure 1.1. The boxes indicate programming paradigms by their essential language features in boldface; the steps in boldface below the line indicate the essential steps for each paradigm. Arrows indicate essential abstractions added to go from one box to another; they do not exclude, for example, loops with sets in the "sets" box and recursion with arrays in the "recursion" box. The gist of this diagram is the following:

- The core step, Step Incrementalize, is essential for all programming paradigms.
- Step Iterate is essential when high-level control abstraction is used.
- Step Implement is essential when high-level data abstraction is used.
- Doing Step Incrementalize across modules is essential when high-level module abstraction is used.

We will see that the driving principles underlying the III method are captured in step-by-step analysis and transformations for problems specified in all of the paradigms. Indeed, the method can be fully automated given simple heuristics for using algebraic laws to help determine minimum increments and reason about equalities involving primitive operations; the method can also be used semiautomatically or manually.

The method is the discrete counterpart of differential and integral calculus for design and optimization in continuous domains for engineering system design, rooted rigorously in mathematics and used critically for sciences like physics. In particular, incrementalization corresponds to differentiation of functions, iteration corresponds to integration, and iterative incremental maintenance corresponds to integration by differentiation. Minimizing iteration increments and maintaining auxiliary values for incrementalization yields the kind of continuity that is needed for differentiation in calculus. The extra concept of implementation is needed because we have to map the resulting computations in the discrete domains onto computer hardware. Indeed, Step Iterate and Step Incrementalize are essentially algorithm design, whereas Step Implement is essentially data structure design.

Overall, the III method unifies many ad hoc optimizations used in the implementations of languages and supports systematic design of algorithms and data

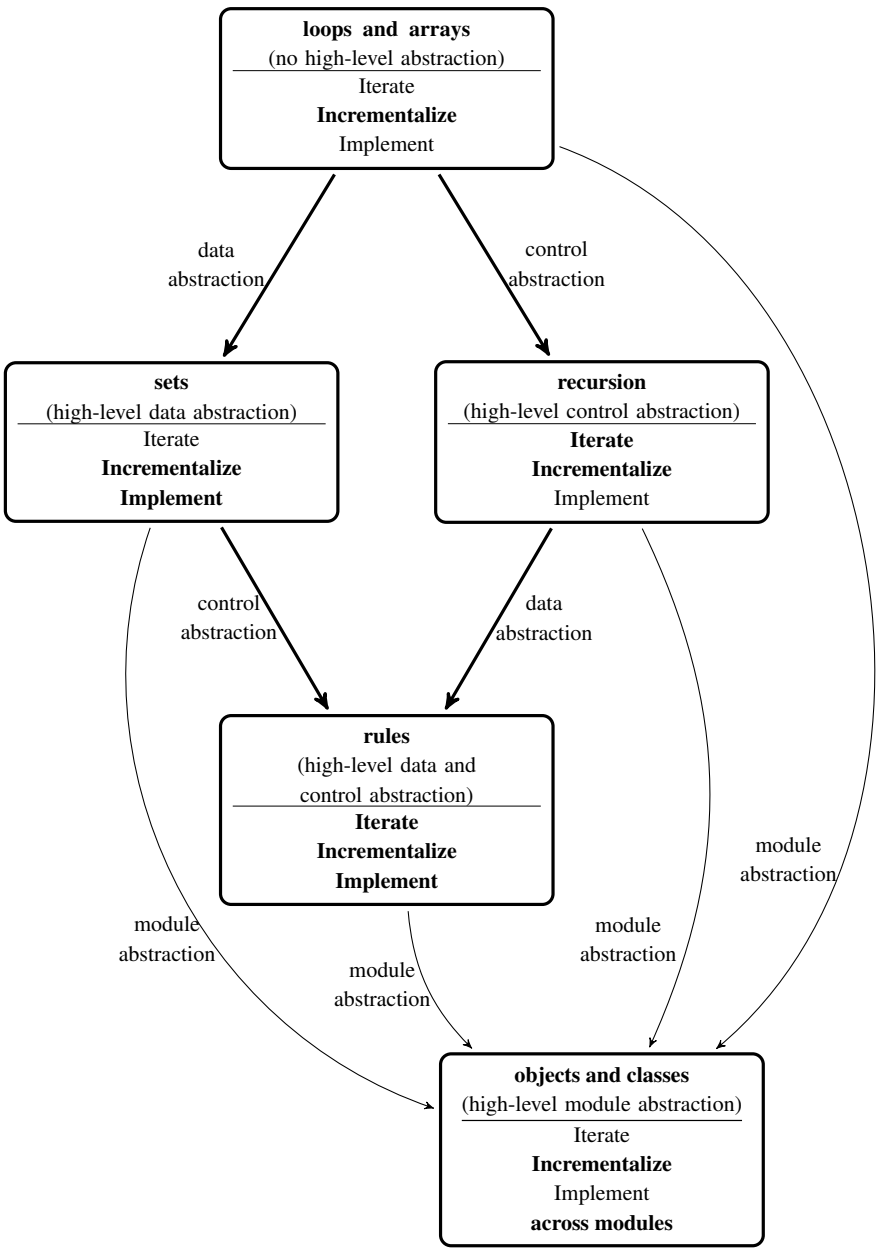

Figure 1.1 III method for different language abstractions.