# Structure and Efficiency of Computer Programs

Robert Harper
Carnegie Mellon University

July 23, 2014

## 1 Introduction

Much fundamental research in computer science is driven by two complementary notions of beauty in programming, that arising from the *structure* of a program and that arising from the *efficiency* of an algorithm. As Lewin famously said, "there's nothing so practical as good theory" [Marrow, 1984]. Decades of experience has borne out the remarkable efficacy of theory in improving the practice of programming.

Programming language theory is the study of the structural aspects of programming. The central notion is that of *compositionality*, the construction of a program by composition of separable parts. Results are formulated in terms of programming languages defined by *type systems*, which mediate composition of components, and *semantics*, which determines the execution behavior of those programs. Notable theorems include the *parametricity theorem*, which provides the mathematical foundation for the informal concept of data abstraction.

Algorithm theory is the study of the efficiency aspects of programming. The central notion is of *asymptotic analysis* of the time and space usage of a program, expressed in terms of a size measure or a probability distribution on the inputs. Results are formulated in terms of *machine models* of computation, such as TM's or RAM's, which define the basic steps of an algorithm and their exceution cost, perhaps relative to parameters such as the number of processors on a PRAM. Algorithms are expressed in "high-level assembler", a simple imperative programming language, for which the translation to the "official" machine model is readily understood. Often algorithms are expressed in terms of explicit representations of data structures using "pointers" and "words" to manage storage.

By and large the algorithms and programming languages communities operate in isolation from one another. Programming languages researchers focus on the practicalities of building large software systems, and are seldom concerned about efficiency. Algorithms researchers focus on the efficiency of core programming techniques, and are seldom concerned about composition. (These are, of course, caricatures of reality, but it seems to me that they contain an element of truth.)

The separation between these two areas is not merely an accident of history, or a product of social structures, but rather signals fundamental challenges that might profitably be addressed by a joint research effort seeking to consolidate and integrate advances on both aspects of programming. The prospects for a successful effort hinge on a shared appreciation for the importance of mathematically rigorous theories of programming and a shared sense of beauty that drives much work in both areas.

# 2 Some Challenges

## 2.1 Higher-Order Programming

There is a large and growing gap between the languages used for software development and the machine models on which algorithm analysis is grounded. When dinosaurs roamed the earth, software was written in low-level imperative languages, such as C, that are so close to machine models that one could easily ignore the distinction. Algorithms sketched in a C-like notation are readily transcribed to actual running code, resulting in a tight interplay between theory and practice.

This happy correspondence is no longer the common case. Software is written in very high-level languages whose mapping to machine models is far from transparent, making it very difficult to anticipate the cost of execution of compiled code. The gap is simply too large to be dismissed. Languages in wide use these days emphasize automatic storage management, so that the allocation, layout, and disposal of data structures is out of the explicit control of the programmer. Doing so greatly facilitates programming, and is vital to ensuring that modules are composable to form programs.

A closely-related trend is the increasing use of higher-order concepts such as functions (functionals, operators, iterators); infinite data structures, such as infinite streams of data items or events; object-oriented methods that represent data as active objects, rather than passive structures; concurrent composition of programs from components. Languages exhibiting these characteristics include Java, JavaScript, Python, Ruby, Scala, OCaml, and Haskell. Such abstractions are not easily described or manipulated in low-level terms, and in any case should not be if modularity is to be preserved.

These trends in programming emphasize structural considerations, often at the expense of a proper understanding of efficiency. Although merely anecdotal, I have often heard developers say that "everyone knows that algorithms as we learned them at school are irrelevant to practice." Setting aside the evident naivete of such a remark, there is a germ of truth to it in that a machine-level understanding of algorithms and analysis is very difficult to apply in modern programming situations that involve abstract data structures whose representations in storage are emphatically hidden from the programmer.

These considerations suggest opportunities for research on "higher-order algorithms" [Okasaki, 1999] that are expressed in terms of the high-level abstrac-

tions that are increasingly commonly used. Useful data structures include *infinite*, as well as *finite*, data, and are typically *persistent*, rather than *ephemeral*, for reasons I shall make clearer shortly.

Correspondingly, there is a strong need for research on "cost semantics" [Blelloch and Greiner, 1995] for assessing the (parallel and sequential) time and space complexity of the code that people actually write. Thinking in terms of how such programs might be compiled is both fallacious (compilers are not very predictable, and there often are many compilers for the same language) and utterly impractical. A cost semantics provides a fulcrum balancing implementation considerations (restrictions on the compiler writer) against application considerations (assumptions the developer wishes to make).

## 2.2 Parallel Programming

Technological developments at the chip level, requirements for low-power operation, and the ever-increasing demand for compute power all argue for the long-term importance of embracing parallelism. The algorithms community has recognized this for decades, and has developed an impressive body of results providing both upper and lower bounds on parallelizability of algorithms. Much of current practice seems to rely on relatively low level methods for harnessing parallelism, including dedicated hardware (GPU's, co-processors), concurrent programming techniques based on message-passing or locking primitives that require detailed scheduling of resources and controlled placement of data, and often bake in platform considerations that impede scalability and evolution of programs.

There is some, relatively limited, research in the programming languages community addressing these problems at a much higher level of abstraction, one that is scalable across platforms and over time. But this work is in its infancy. Most practical work takes place at low levels of abstraction in order to exploit parallelism to maximum effect, or at high levels of abstraction, such as concurrency, that impede precise analysis of their complexity properties. It seems that much could be done to improve both the structural aspects of parallel programming and the analytic aspects of understanding their complexity.

A promising start is to use functional, rather than imperative, programming languages [Blelloch and Greiner, 1995] for parallelism. Functional languages are naturally parallelizable, because the value of an expression is fully determined by the expression itself, and cannot be influenced by any other computation occurring in parallel. Data structures in functional languages are naturally persistent, because they are handled as atomic values on a par with numbers or strings in other languages, and are, by definition, immutable (no in-place update is permitted). This makes them ideal for parallelism, since a given data structure may be acted upon by mainly parallel computations, often simultaneously, without the possibility of interference.

## 2.3 Modularity

The single most important method for controlling the complexity of building large software systems is modularity, the decomposition of programs into composable components. The vast bulk of research in programming languages may be seen as an attempt to address this basic principle. The dominance of type systems as a tool for language design stems directly from Reynolds's dictum that a type system is a syntactic discipline for enforcing levels of abstraction. Roughly speaking, one component of a system should depend only on the *type*, or the *specification*, of another, and not on its *implementation*.

This general approach works well for *behavioral*, or *correctness*, properties, but breaks down completely for *complexity*, or *efficiency*, properties. In technical terms this is a tension between *extensionality* and *intensionality*. Extensionally, a function is a pure I/O behavior, and has no complexity properties—it is a mathematical function that happens to be computable. Intensionally, a function is a program to which we may ascribe complexity—it is a piece of code to which we may associate an extension.

Effective modularity depends crucially on the extensional viewpoint: one programmer may work against an interface whose code does not even exist, or may change radically over time. To minimize integration problems the assumptions about the other component should be concerned only with its behavior, and not the details of its code. But understanding the efficiency of a program depends crucially on the intensional viewpoint: one must understand the *code* of another component in order to assess its complexity and how it may interact with the *code* in a component of interest.

Put another way, algorithms tend not to compose well. A good example is provided by dynamic algorithms [Acar et al., 2006] whose behavior is defined in terms of "small changes" to the input, with the goal to achieve an efficient computation of the correspondingly altered output. But unfortunately dynamic algorithms do not compose! For example, if "small" changes to the input to the first induce "large" changes to its output (in the sense of "small" defined for the second), then the composition can be very inefficient, even when the components are very efficient.

A related example is how to specify the complexity of higher-order programs. Consider the familiar `filter` function that takes a predicate (function from the element type to the boolean type) as argument and selects from a given list all the elements of that satisfy the predicate. It is very difficult to make statements about the complexity of `filter` in general, because the behavior of the predicate can vary wildly on each argument. We can restrict attention to only "well-behaved" predicates, which may work well for this example, but in general when using higher-order programming it is difficult to make general statements about the complexity of higher-order functions.

More generally, algorithms are usually evaluated as if the algorithm were the entire program, rather than a subroutine in a larger application. The analysis is in terms of a measure of the input (say, the number of nodes or edges of a finite graph), and the complexity of the operations on the data structure are given in

terms of that measure. From a systems perspective, this amounts to evaluating the efficiency of the API, which can have little bearing on the efficiency of a program that uses it. An example that arose in the PSciCo Project [Blelloch and Harper, 2004] involved the construction of simplicial complexes. In the interest of parallelism the operations on a complex were typically slower by a factor of $\lg n$ when compared to an imperative implementation, with the payoff being that the representation is persistent and would therefore be more amenable to parallelization. But even that overhead turned out not to matter in practice. For example, the QuickHull implementation of the 3d-complex hull requires only $\mathcal{O}(n)$ operations on the hull itself, which translated to $\mathcal{O}(n \lg n)$ time in the functional case. Yet this was no slow-down, because it nevertheless matches the well-known lower bound.

## 2.4 Verification

Proving properties of programs has long been a central goal in programming research. These days type systems are being developed that blur or erase the distinction between type checking and verification—types are sufficiently rich as to be able to state and enforce detailed correctness properties of programs, often with sufficient automation as to make it routinely possible to ensure compliance with useful specifications [Appel et al., 2014]. It seems clear that the trend is now irreversible, for both practical and theoretical reasons.

Very little has been done on the mechanical verification of complexity properties of programs. There are very good reasons for this. One is that language research has long been deeply intertwined with verification. Indeed, the very idea of type systems arises from constructive mathematics, which is based on computation as a starting point. The "throw the code over the wall" model of program verification has had substantial success, but it seems that the process is much easier if verification is integrated with development at the outset. At the behavioral level much has been achieved to making verification practical day-to-day, but relatively little effort has been expended on how to express and verify complexity properties.

Another major obstacle to mechanical verification of complexity is the sheer mathematical sophistication of the methods used in algorithm analysis. It is routine to deploy deep results in analysis, combinatorics, and probability theory, for example. Very little has been done to mechanize these important bodies of mathematics. This means that there are huge opportunities in this area, but it is clear that there are also huge challenges to be overcome.

Despite these concerns substantial progress is already being made. For example, one may isolate crucial properties of a data structure that drive its complexity, and verify these properties mechanically. A good example is provided by Appel's recent work [Appel, 2014] on verifying the complexity of balanced tree algorithms. The main point in a complexity analysis is that the height of a balanced tree is logarithmic in the size of its frontier. Such a property can be readily stated and verified, even at the level of C code, by proving that the difference in height of the children of a node is bounded by a constant. Dunfield

[2007] shows that similar results can be achieved by nearly automatic methods using type refinements for functional programs. Similarly, the Easycrypt Project [Barthe, 2014] and Morrisett [Morrisett, 2014] are mechanically verifying security protocols using probabilistic methods, which one might fear would require a large body of mathematics, such as measure theory. But it turns out that many useful properties can be proved using relatively elementary methods, lowering the barrier to entry for machine-checked proof of such an important class of programs.

# 3 Some Opportunities

## 3.1 Certifying Algorithms

A very interesting trend in algorithm design is the idea of *certifying algorithms*, those that produce a machine-checkable certificate of the validity of their output. For example, a certifying planarity tester would not take a finite graph and return a boolean, but rather would either produce an embedding of the graph into the plane, or produce an embedding of a Kuratowski subgraph into the given graph as proof that it is non-planar. This point of view is not only tremendously practical as a means of ensuring code correctness, but it is also fundamentally coherent with the emphasis on constructive mathematics in language research—a certifying algorithm is a *constructive* proof that for every finite graph $G$ either there is an embedding of $G$ into $\mathbf{R}^2$ or there is an embedding of $K_5$ or $K_{3,3}$ into $G$. Constructively, one is required to produce a proof of one of the disjuncts (not merely that both cannot be false), and in each case to explicitly exhibit the embeddings (not merely that an embedding cannot fail to exist).

Certification provides a point of contact between existing work in languages and algorithms that could well provide the basis for further coordination and collaboration between the two areas. One possibility is to develop a *constructive* formulation of the properties of algorithms that is, from a computational viewpoint, sharper than the classical view. According to classical logic there is no distinction between a graph being planar and the impossibility of a graph being non-planar. But constructively there is all the difference between the "mere existence" of an embedding in the plane (given by evidence that cannot be used in a further computation) and the "existence" of such an embedding (which must be presented by an assignment of coordinates to the nodes, say, in $\mathcal{R}^n$). Mehlhorn's work shows that there is an important practical application to drawing such a distinction, and demonstrates that in algorithms, as in other settings [The Univalent Foundations Program, 2013], constructivity may be a useful tool for obtaining practical results.

## 3.2 Language-Based Models

It would be useful to bridge the gap between the low-level machine models used in algorithms research and the high-level languages used in software de-

velopment. It seems clear that the performance of practical programs could be greatly improved by greater use of sophisticated algorithmic techniques. But, as mentioned earlier, the use of higher-order programming impedes the expression and analysis of algorithms written in these languages.

One important ingredient in addressing this problem is the development of *cost semantics* for realistic languages. The importance of a machine model in algorithms is simply that it provides a means of counting space and time usage of a program: one instruction, one unit of time; one word, one unit of space. Rather than defining complexity at the level of the machine, it is possible to assign cost measures to the constructs of a high-level language directly, so that the analysis can be made in terms of the high-level code, rather than in terms of how it is compiled to a low-level machine. The high-level costs are separately validated by proving that they can be realized on various machine models, with bounds influenced by platform parameters. This separation improves scalability as technology changes, and admits more abstract notions of cost than the individual steps of a machine. Moreover, by working at the language level one may consider more abstract notions of cost than are easily expressible at the machine level.

## 3.3 Functional Parallel Algorithms

A growing trend in program development is the use of functional programming, which emphasizes a mathematical formulation of a problem in terms of functions and data structures as abstract notions, rather than in terms of their concrete realization in memory.

One important application of functional programming is to parallelism. The advantage is the "declarative" nature of functional programs, which avoid overspecifying the details of how data is represented or manipulated, in favor of a more equational description of what the computation is to achieve. Functional languages provide a useful example of an abstract cost measure, the *cost graph*, which associates to a computation a representation of the dependencies among subcomputations, exposing the implicit parallelism available for that program. The cost graph determines both the sequential and parallel time and space complexity of the program, and is a useful abstraction with which to analyze the asymptotic complexity of a program without regard to platform-specific parameters, which can be factored in later.

By stressing the tight connection with mathematics, functional languages encourage a "theoretical" mindset that is consonant with the aims of both the algorithms and language communities. Behavioral verification of functional programs is far easier than for their imperative counterparts; it is interesting to ask whether the same might be said for verification of their efficiency. Certainly the correctness arguments, on which any complexity analysis depends, are far simpler, particularly in the parallel case, so this can only help.

The functional perspective has also proved effective in a distributed setting in which there is no useful concept of shared state. The well-known Hadoop and Map-Reduce systems exploit this aspect of parallelism to perform parallel

computations over large-scale distributed data sets. Such algorithms are highly sensitive to issues of data locality, because of the enormous difference in time to access "remote" from "local" data. This kind of problem was studied recently by Blelloch and Harper [2013, 2014] from the perspective of a cost semantics, showing that one can reason about the I/O complexity of an algorithm without having to drop down to a low-level machine model.

### 3.4 Performance Analysis and Verification

Performance debugging is an enormous time sink in program development. Since the efficiency of high-level abstract languages can be hard to assess by analytic means, it is necessary to resort to profiling tools that treat the efficiency problem "empirically", as if the program were an object found in nature, rather than an artifact of known origin with known properties. Improving this situation would be greatly beneficial in practice, and is likely to be a source of many interesting research ideas. One may view mechanical complexity verification as the ultimate profiling tool—one that does not even require the execution of the program to obtain useful information! More likely, however, a combination of static verification and dynamic measurement techniques are likely to be helpful. Developing these ideas seems to be a prime opportunity for interaction between the algorithms and languages communities.

The concept of a cost semantics presents opportunities for mechanized proof that build on recent developments in compiler correctness. Appel and Leroy [Appel et al., 2014] have amassed an impressive body of work on proving that the code output by a compiler (for a simple C-like language) is behaviorally equivalent to the source code. Given a cost semantics, the natural next step is to show not only that the emitted code is functionally correct, but moreover that it meets the expected complexity bounds as stated in the cost semantics. Doing so would rule out extremely subtle bugs, such as space leaks, that arise from compiler errors that, in the case of space leaks, retain live data that truly ought to be considered dead. (See, for example, Spoonhower [2009] for a real-world scenario of this kind.)

## 4 A Proposal

This document is written to stimulate discussion and further the possibilities for fruitful interaction between the algorithms and languages communities. Such suggestions demand discussion and debate with the goal of generating new ideas and new research programs that would stimulate both communities. To this end I propose a workshop on algorithms and languages that would be devoted to this task.

# References


U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

A. W. Appel. Efficient verified red-black trees. Unpublished manuscript, 2014. URL `http://www.cs.princeton.edu/~appel/papers/redblack.pdf`.

A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Programs Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN 9781107048010.

G. Barthe. Easycrypt. Project Web Site, 2014. URL `https://www.easycrypt.info/trac`.

G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.

G. E. Blelloch and R. Harper. The PSciCo Project. Project Web Site, 2004. URL `http://www.cs.cmu.edu/~pscico`.

G. E. Blelloch and R. Harper. Cache and I/O efficent functional algorithms. In *POPL*, pages 39–50, 2013.

G. E. Blelloch and R. Harper. Cache efficient functional algorithms. *Communications of the ACM*, 2014. URL `http://www.cs.cmu.edu/~rwh/papers/iolambda-cacm/cacm.pdf`. (To appear as an ACM Research Highlight).

J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. URL `http://www.cs.cmu.edu/~rwh/theses/dunfield.pdf`.

A. Marrow. *The practical theorist : the life and work of Kurt Lewin*. BDR Learning Products, Annapolis, MD, 1984. ISBN 0934698228.

G. Morrisett. Verifying security protocols in coq. (Private communication), June 2014.

C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2.

D. J. Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009. URL `http://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf`.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Institute for Advanced Study, 2013. URL `http://homotopytypetheory.org/book`.