

The Programming Languages Enthusiast

BY SWARAT CHAUDHURI | SEPTEMBER 25, 2014 · 8:04 AM

Bridging Algorithms and Programming Languages

In the computing stack, PL sits between algorithms and systems. Without algorithms to implement or computer systems to run them on, there would be no need for programming languages. However, the research communities that study algorithms, PL, and systems don't really have much of an overlap. This is perhaps unavoidable: computer science is now a mature field, and researchers in mature fields tend to pursue specialized and technical research questions.

At the same time, it seems important that the approaches — assumptions and methods — of different subfields of computing be compatible to some extent. At the end of the day, computer science aims to produce solutions for the whole computing stack. An “impedence mismatch” between its subfields compromises our ability to come up with such end-to-end solutions.

This suggests that the comparative study of assumptions, techniques and cultures of different CS fields (a sort of “comp lit” for computer science) is potentially valuable.

Personally, I have always been intrigued by the relationship between the fields of programming languages and algorithms. In this post, I discuss similarities and differences between these two areas, and argue that their synthesis could be interesting for both research and teaching.

What do PL researchers study?

To compare PL with other fields, we need to first decide what PL is. I can think of a few themes that run through our field:

- **Language design.** Unsurprisingly, the design of programming languages is a fundamental goal of PL research. A designed language may be a real, usable language, or a “calculus” that captures the mathematical core of a bigger language. In either case, a key challenge is *semantics*: the rigorous definition of how programs are evaluated. Such definitions are not arbitrary. Typically, researchers aim to guarantee that programs in their languages satisfy some desirable mathematical properties, for example [memory safety](#) or data race-free concurrency. Commonly, a language designer also wants to exploit practical features of hardware on which programs execute and the application domain where the language is used.
- **Reasoning about programs (“verification”).** Rigorous reasoning about program properties — for example, verifying termination or deadlock-freedom — is a major theme in PL. Typically, PL researchers want to accomplish this task in an algorithmic or at least semi-automated way. As [automated analysis of this sort is undecidable for Turing-complete languages](#), PL researchers focus either on weaker classes of languages where proving properties is *decidable* ([model checking](#)), or on *sound but incomplete* methods ([type systems](#) and [static analysis](#)), or on semi-algorithms guided by human programmers ([interactive theorem proving](#)).
- **Compiler design.** A programming system must generate executable code, and designing algorithms that can do so is another core PL problem. Traditional challenges here include parsing and program optimization. In recent times, research on this theme has expanded to designing [verifying compilers](#) that can reason about correctness, or [program synthesizers](#) that use sophisticated search algorithms to generate executable code from nondeterministic specifications.

What do algorithmists study?

Now let's consider the field of algorithms. Any algorithms research starts with a *model of computation* on which algorithms are to execute. Typically, such a model distills the essence of a real-world computational setting, and constrains what an algorithm is allowed or not allowed to do. For instance, in the so-called [streaming model of computation](#), an algorithm can only process a few items in an input stream at one time. In [sublinear algorithms](#), the algorithm cannot read its input data fully (capturing real-world intuitions about large datasets), but is permitted to probe or sample from this data.

Any given model of computation comes with a rigorously defined notion of *efficiency*. The algorithms researcher's goal is now to come up with algorithms that are provably efficient. Over the years, algorithmists have studied many different ways to make algorithms efficient. For example, a common approach to achieving efficiency is to weaken the guarantees that the algorithm offers. In particular, [randomized algorithms](#) are allowed to have one-way or two-way probabilistic error in their outputs. [Approximation algorithms](#) must solve hard global optimization problems in polynomial time, but are allowed to produce outputs that are only within an approximation factor of real optima.

PL and algorithms: technical differences

There are several commonalities between PL and algorithms and, to my mind, one key difference.

One obvious commonality is the object of study: an algorithm and a program are basically the same thing (algorithms are commonly defined to be programs that terminate on all inputs, but this is a technical detail). Second, the models of computations that algorithms researchers assume are, fundamentally, nothing but programming languages: a demarcation of the set of permissible programs/algorithms and a specification of their dynamics. One could think of streaming algorithms or sublinear algorithms as the classes of programs expressible in domain-specific languages with restrictions on how a program accesses data.

The fundamental difference between algorithms and PL, it seems to me, lies in a quantifier:

An algorithms researcher's goal is to produce an algorithm/program with desirable properties. In contrast, whether a PL researcher is working on language design, verification or compilers, he or she aims to come up with proofs and procedures that apply to *all algorithms* that can be written in a language.

For instance, a program verifier is a procedure that can take an *arbitrary* algorithm (within a certain model of computation) and check that it satisfies certain properties. The designer of a language with guarantees of memory safety aims to guarantee a theorem about the set of all algorithms possible under their model.

Interestingly, this makes the field of PL closer to complexity and theory of computation, which study properties of classes of algorithms. Think of two elementary statements from complexity theory: "every comparison-based sorting algorithm must have $\Omega(n \log n)$ complexity" and " $\text{PSPACE} = \text{NPSPACE}$ ". The first statement is a meta-theorem about a programming language whose semantics comes with a quantitative model of resource consumption (that comparisons of values can be done in unit time). The second is a comparison between two different programming languages.

PL and algorithms: cultural differences

Other differences between PL and algorithms are cultural rather than technical.

The first such difference is that PL research is usually designed to maintain a path to execution on real computers. Models of computation in the algorithms literature are an artifact of analysis. In contrast, even the core languages designed by programming linguists tend to have the eventual goal of implementation. It also seems fair to say that PL researchers implement their ideas more often than algorithmists do. Many if not most POPL papers come with an experimental evaluation. SODA papers, on the other hand, rarely do. One corollary of this is that algorithmists are more susceptible to

conflating models with reality. As one example, most algorithms hold polynomial time complexity as *the* notion of efficiency. However, NP-hard problems are not always hard to solve in practice — for instance, boolean satisfiability may be NP-complete, but SAT-solvers work very well on large formulas in many domains. The empirical focus of PL researchers has allowed them to leverage such practical tools.

A second cultural difference between algorithms and PL lies in their attitude towards abstraction, and their definition of what makes an algorithm/program desirable. Models of computation in the algorithms and complexity literature tend to be low-level, without the abstractions available in most modern programming languages. The main objective of design is efficiency, defined in terms of the number of low-level steps taken by executions.

As Bob Harper argues in his talk on [“two notions of beauty in programming”](#), PL researchers often have a different take on the cost of a computation. For many PL researchers (especially those who work on language design), *the true notion of a computation's cost is the effort involved in programming and reasoning about programs*. A desirable computation or proof is one that can be expressed effectively using high-level programming abstractions, and composed easily with other computations/proofs. The design of new languages or models of computation is driven by an interest in making programs more desirable in this sense.

A third, related difference is that algorithms research is traditionally much heavier on quantitative reasoning. Notions of efficiency studied by algorithmists are easy to quantify. Also, in contemporary algorithms, notions of correctness are often randomized or approximate, so that correctness analysis also requires quantitative reasoning. In contrast, PL research has historically focused on reasoning about boolean properties such as type safety, deadlock-freedom, and termination. It is true that there is a growing literature in the PL world on quantitative reasoning about programs (including reasoning about probabilistic and approximate computation), but this kind of work is still in the minority.

Finally — and I realize that I may get some flak for saying this — it seems to me that algorithms researchers are historically much more agile than PL folks in identifying new application areas. There are many thriving research topics of the form “x + algorithms”: algorithmic genomics, [algorithmic game theory](#), and [network science](#) are only three prominent examples. Algorithms researchers have been able to distill interesting algorithmic problems from these applications and come with many exciting technical results. In contrast, PL researchers have tended to focus on classic applications in system software, or chosen to be application-agnostic.

A synthesis?

The technical differences between algorithms and PL are innate — there is after all a reason why these are different areas. However, the cultural differences between PL and algorithms exist primarily for historical reasons, and it seems that a synthesis of the “positive” cultural attitudes of PL and algorithms can benefit both fields.

Specifically, it seems clear to me that there's a “PL angle” to most application domains that algorithmists study, and applying it would benefit the entirety of computer science. Take, as an example, cryptography. The vast majority of work on cryptography is algorithmic, and at a theoretical level there's nothing wrong about these algorithms. However, as we saw from the [Snowden revelations](#), because of vulnerabilities at the software and hardware level, guarantees at the level of abstractly defined algorithms may not transfer to implementations of these algorithms. PL techniques could have potentially prevented these errors from happening.

It seems to me that we can amend our list of activities of PL researchers as follows:

- **Language design.** Come up with high-level programming abstractions for application domains where there are interesting algorithms to be designed. Consider any property of the model of computation that the algorithm designer cares about. For example, this property could be “A program can only read k items of a data stream at a time” or “A program can sample from certain probability distributions” or even “A program must terminate in polynomial time”. Make sure that every program in the language actually satisfies this property.

- **Reasoning about programs.** Consider any of those rich properties of algorithms that algorithms researchers like to prove. For each such property, there's a program analysis/verification problem: show that a given program/algorithm satisfies the property. Properties of interest could include bounds on the algorithm's complexity, optimality of the algorithm's output, approximate optimality, and bounds on the algorithm's probabilistic errors.
- **Compiler design.** Design compiler transformations and optimizations that do not compromise guarantees developed at the algorithmic level, and instead preserve them all the way to executable code.

There are quite a few existing PL research efforts that can be said to fit this agenda. Examples include recent work by PL researchers on [reasoning about statistical privacy](#), [verification of cryptographic protocols](#), and [analysis of probabilistic programs](#). In each of these cases, our community targeted an algorithmic application and extracted PL problems from them. I would like to see many more efforts like this, in areas ranging from biology to graphics to robotics to economics (disclosure: PL approaches to robotics and computational economics are two of my current research interests). The technical problems of interest in these areas are likely to be quantitative, which opens up possibilities of generalizing the qualitative techniques that PL researchers have traditionally used.

There are also pedagogical benefits to a synthesis of PL and algorithms. For one, it would tie the undergraduate sequence of programming courses to undergraduate theory classes. An effort of this sort is CMU's two-course introduction to functional programming and algorithms ([15-150](#) and [15-210](#)), which emphasizes the benefits of functional program abstractions while teaching algorithms and data structures. For another example, PL ideas are a natural fit for a theory of computation class. Today, the standard model of programs in a class on automata and computability is the Turing machine, which is far removed from the abstraction-rich programming languages in which students implement their algorithms. Complementing the Turing machine model with recursive functions and the lambda-calculus would partly bridge this gap. Finally, finite and pushdown automata are presented in undergrad automata theory using old-fashioned applications in parsing. These applications can and should be complemented with the use of automata in modern program verification, as models of software and hardware systems.

All in all, it seems clear to me that bridging some of the gaps between PL and algorithms is both possible and desirable. Some encouraging steps in this direction have already been taken, but much more remains to be done. Perhaps the best way to develop synergy between the fields is to create funding opportunities for research at their interface. Bob Harper's [recent NSF whitepaper](#), which builds on the [talk](#) of his that I mentioned earlier, is an example of an effort to mobilize such funding. A less ambitious goal is to create fora with representation from both communities. I personally can think of worse ways to spend a few days than hanging out with some algorithms researchers, hearing about recent exciting results in their field and thinking about their language-level analogs.

[Note: This article has been edited since it was originally posted.]

Share this:



14 Responses to *Bridging Algorithms and Programming Languages*

Robert Harper

September 25, 2014 at 2:10 pm



Last spring I wrote a position paper [Structure and Efficiency of Computer Programs](#) outlining a program of research integrating semantics and combinatorial theory.

[Reply](#)**Swarat Chaudhuri**

September 26, 2014 at 10:44 am




I've updated the post, and now it links to the position paper.

[Reply](#)**Adam**

September 25, 2014 at 8:51 pm



"As one example, most algorithms hold polynomial time complexity as the notion of efficiency."

Maybe you've been hanging around with the wrong algorithms researchers 

There are gazillions of researchers who work very hard to design algorithms that satisfy the constraints of a particular application on particular hardware, but they publish their work in more specific venues (NIPS/ICML, KDD, SIGMOD, SPAA, high-performance computing,...). Of course, since they value their funding, they don't *call* themselves algorithms researchers. But that is simply a case of sheep dressing up as well-funded wolves.

That said, you are right that the theoretical algorithms community is largely disjoint from these more applied communities. [The gap is not large everywhere: much theoretical work on topics like optimization, statistical learning and numerical analysis, for example, is relatively close to practice. Theoretical crypto, on the other hand, ranges from the practical to the proud-to-be-useless, with many researchers publishing only the latter category of papers.] Why is the attitude so different among the STOC/FOCS crowd from that of PL folks?

[Reply](#)

Robert Harper

September 25, 2014 at 11:03 pm



You should cite our 15-150 and 15-210 as a pair; they are two parts of the same course on parallel functional programming and verification of correctness and complexity.

[Reply](#)**Swarat Chaudhuri**

September 26, 2014 at 9:37 am



Thanks, Bob. I have updated the post, and now it has links to both courses.

[Reply](#)**Jan Hoffmann**

September 25, 2014 at 11:36 pm



Thanks Swarat, that's another great post.

I am working on automatic resource bound analysis for both functional programs and C code. The project websites are

<http://raml.tcs.ifi.lmu.de>

and

<http://www.cs.yale.edu/homes/qcar/aaa/>

One of our papers about resource analysis for function programs is for instance “Multivariate Amortized Resource Analysis” (POPL’11).

In general, there is a lot of interesting work going on in automatic resource bound analysis. Two great papers are

“A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis” by Sinn, Zuleger, and Veith (CAV’14)

and

“SPEED: Precise and Efficient Static Estimation of Program Computational Complexity” by Gulwani, Mehra, and Chilimbi (POPL’09).

Nevertheless I agree with you that there are a lot of research opportunities in this area and that it would be great if more people would work on quantitative software analysis.

One thing that I would like to add to your post is that quantitative software analysis is not only intellectually interesting; it has also many important applications. One example is memory-usage guarantees for verified software (see, e.g., “End-to-End Verification of Stack-Space Bounds for C Programs” (PLDI’14)). Another example is worst-case execution time (WCET) analysis for real time systems.

[Reply](#)

Swarat Chaudhuri

September 26, 2014 at 10:51 am



Thanks for the great links!

[Reply](#)

Justin Hsu

September 26, 2014 at 2:07 am



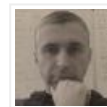
Adam:

“That said, you are right that the theoretical algorithms community is largely disjoint from these more applied communities...Why is the attitude so different among the STOC/FOCS crowd from that of PL folks?”

I think this depends on what PL folks you are talking about 🤔 It may be true the STOC/FOCS crowd is more motivated by the theoretical beauty of an idea rather than the practical applicability, but I think you can find such people in PL as well (e.g, people working on denotational semantics, the LICS crowd, etc.).

[Reply](#)**Dan Razvan Ghica**

September 26, 2014 at 2:43 am



Bounded Linear Logic as a type-theoretic characterisation of polynomial complexity is one bridge worth mentioning. We had a recent workshop on it and the web page includes a BLL reader.

<http://www.cs.bham.ac.uk/~drg/bll.html>

[Reply](#)**Justin Hsu**

September 26, 2014 at 9:21 pm



A very interesting post, thanks! As someone who works in both fields (co-advised by an “algorithmist” and a PL person), I’ve spent way too much time thinking about these fields. (More than) a few thoughts...

“It also seems fair to say that PL researchers implement their ideas more often than algorithmists do. Many if not most POPL papers come with an experimental evaluation. SODA papers, on the other hand, rarely do.”

Making algorithms efficient in practice is difficult and very valuable work. Instead of investing resources in making each algorithm implementable and efficient in practice from the start, I think theoretical algorithms researchers would rather aim for the theoretical guarantee at first (which can already be quite involved). People with applications in mind can pick algorithms that solve their problem, and implementation effort can be focused on algorithms that will actually be used.

This would be problematic if theoretically efficient algorithms often could not be made efficient, but the track record of turning (with a lot of work) theoretically efficient algorithms into practically efficient algorithms is not bad. Even things like fully homomorphic encryption and probabilistically checkable proofs, which many algorithms researchers may have believed to be unimplementable, are performing much better in practice than anyone could have hoped. Of course, there are also some theoretically efficient algorithms that can only be implemented on a computer the size of the universe, but these seem to be few and far between.

“One corollary of this is that algorithmists are more susceptible to conflating models with reality. As one example, most algorithms hold polynomial time complexity as the notion of efficiency. However, NP-hard problems are not always hard to solve in practice — for instance, boolean satisfiability may be NP-complete, but SAT-solvers work very well on large formulas in many domains. The empirical focus of PL researchers has allowed them to leverage such practical tools.”

I think most algorithmists would agree that our understanding of hard problems is still incomplete. Many NP-hard problems can be solved in practice (termination checkers are even attacking undecidable problems!), while the security of cryptographic systems are based on problems that haven’t been proved

theoretically hard (but seem very hard in practice). As long as heuristics like SAT solvers remain poorly understood, they aren't very useful for proving theoretical results.

I think the main reason PL researchers are comfortable using SAT solvers isn't so much the empirical focus, but rather that PL researchers don't place importance on proving efficiency/termination of algorithms. PL researchers would find a system that provides an "empirical" case for soundness by trying some examples to be baffling, just like some algorithms researchers would find empirically evaluating running time (or termination) of an algorithm by feeding some examples to a SAT solver to be strange.

"A second cultural difference between algorithms and PL lies in their attitude towards abstraction, and their definition of what makes an algorithm/program desirable. Models of computation in the algorithms and complexity literature tend to be low-level, without the abstractions available in most modern programming languages. The main objective of design is efficiency, defined in terms of the number of low-level steps taken by executions."

I would say that the main objective is not the number of steps (which is kind of an arbitrary number, sensitive to the underlying machine model), but rather the scaling behavior as inputs get larger. It's true that Turing machines are incredibly low level and not programmable, but it matters little for analyzing how the space consumption/running time scale with input size.


"For many PL researchers (especially those who work on language design), the true notion of a computation's cost is the effort involved in programming and reasoning about programs."

Perhaps we could say that while algorithmists are concerned about big inputs, PL researchers are more concerned about big programs.

"A third, related difference is that algorithms research is traditionally much heavier on quantitative reasoning."

Of all the cultural differences you mentioned, I think this is the biggest one. It seems to reflect a similar divide in mathematics [1]: algorithmists think much more like analysts and statisticians (working quantitatively, comparing quantities, approximating), while PL researchers think more like algebraists (working with layers of abstraction, working with equality, working exactly). Anecdotally, most algorithms researchers I've talked to preferred analysis, while most PL researchers preferred algebra in school.

"There are many thriving research topics of the form "x + algorithms": algorithmic genomics, algorithmic game theory, and network science are only three prominent examples. Algorithms researchers have been able to distill interesting algorithmic problems from these applications and come with many exciting technical results."

I guess this is a matter of taste  I do believe that algorithmists have a much wider range of tools available for proving theorems. Tools from all areas of mathematics and even physics are freely borrowed, and many results use ad hoc proof techniques. By making compositionality and modularity so central, the main

proof technique that makes sense for most PL results is induction. There is much less flexibility in applying other techniques.

“Specifically, it seems clear to me that there’s a “PL angle” to most application domains that algorithmists study, and applying it would benefit of the entirety of computer science.”

Along with the domains you listed, I think what would interest algorithms researchers the most is applying PL techniques to solve a problem that is algorithmic in nature. Unfortunately I don’t have any ideas for this, but it would be a strong point: “You can use this technique from PL to prove that theorem about randomized algorithms.”

In the other direction, I wonder about the proliferation of SAT solvers in PL. They are a great hammer to solve algorithmic problems without thinking about algorithms, but there is a whole field of algorithmic research! It’s true that many PL problems are considered hard by the algorithms community, but maybe not all. Solving algorithmic problems with a magic black box is great, but if the black box doesn’t work or you need more control about what comes out of the black box, there isn’t much to be done.

[1] <http://bentilly.blogspot.com/2010/08/analysis-vs-algebra-predicts-eating.html>

[Reply](#)

GASARCH

June 3, 2015 at 10:11 am



(Late to the party with my comment)

One other difference that you didn’t mention:

People in algorithms (and also complexity theory) ask short well defined questions that have answers, though they may be hard to find.

PL asks longer questions that need not be as well defined.

This is not an insult to PL.

bill g.

[Reply](#)

Michael Hicks

June 3, 2015 at 10:29 am



I think PL also asks well-defined questions (is your language type-safe?), but sometimes these questions are not particularly short (you need the whole language semantics to say something about type safety) or interesting (type safety is at the basic level — see Derek Dreyer’s comment to my type safety post).

The less well-defined questions about programmer productivity, software reusability, security, etc. are very important for modern practice, and getting to making these well-defined would be a great step forward.

[Reply](#)

arielbyd

July 1, 2015 at 9:20 am



Algorithmists research how to perform a computation efficiently, while PL-people focus on how to efficiently describe a computation.

[Reply](#)**Robert Harper**

July 1, 2015 at 11:19 am



Algorithms ARE programs. The supposed distinction only arose because of ultra-crappy programming languages. PL's need cost semantics to justify algorithm analysis. See my "Structure and Efficiency of Computer Programs" on my web site.

[Reply](#)