What is Programming Languages Research?

Michael Hicks
University of Maryland
We need to hire in PL this year!

... I have a nagging concern: Isn’t PL a solved problem?

Um, no, there’s lots to do.

Really? What is it that you PL people are working on?
We work on Programming Languages!
OK, but …

Don’t modern languages work pretty well? And aren’t they often developed by non-academics?

Yes, but there are still big research contributions still to make.

Doing what?

I should start a blog …
What is PL Research?

PL research views the programming language as having a central place in solving computing problems.

A PL researcher:

❖ develops **general abstractions, or building blocks**, for solving problems, or classes of problems,

❖ considers software behavior in a rigorous and general way, e.g., to prove that (classes of) programs enjoy properties we want, and/or eschew properties we don’t.
What is PL Research?

- The ethos of PL research is to not just find solutions to important problems, but to find the best expression of those solutions, typically in the form of a kind of language, language extension, library, program analysis, or transformation.

- The hope is for simple, understandable solutions that are also general: By being part of (or acting at the level of) a language, they apply to many (and many sorts of) programs, and possibly many sorts of problems.
Example: Improving Program Efficiency

- Quicksort in Haskell

```haskell
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
  where lesser = sort [y | y <- xs, y <  x]
       greater = sort [y | y <- xs, y >= x]
sort _ = []
```

- Parallelize it

```haskell
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = force greater `par`
  (force lesser `pseq` (lesser ++ x:greater))
  where lesser = sort [y | y <- xs, y <  x]
       greater = sort [y | y <- xs, y >= x]
sort _ = []
```
Thought Process

- Two halves of input list can be constructed in parallel
- OK because each activity is independent
- This should be a win for small \( xs \) on \( n>1 \) cores assuming `par` and `pseq` manage parallel resources efficiently
Thought Process, Generalized

- Automatically pick components of a program to parallelize
- Choose those such that the meaning of the program is preserved, and the performance is likely to improve.

PL research lifts problems to the level of the language, turning a one-off solution into a general one
Example: Authenticated Data Structure

- **Merkle tree** (1988): Complete tree, where server answers queries with evidence the answer is correct

- Since then, *separate papers* on: sets, dictionaries, range trees, graphs, skip lists, B-trees, hash trees, …
Authenticated Data Structures, Generically

Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi
University of Maryland, College Park, USA

Abstract
An authenticated data structure (ADS) is a data structure whose operations can be carried out by an untrusted prover, the results of which a verifier can efficiently check as authentic. This is done by having the prover produce a compact proof that the verifier can check along with each operation’s result. ADSs support outsourcing data maintenance and processing tasks to untrusted servers without loss of integrity. Past work on ADSs has focused on particular data structures (or limited classes of data structures) one at a time, often with support only for particular operations. This paper presents a generic method, using a simple extension to a ML-like functional programming language we call λ (lambdaault), with which one can program authenticated operations over any data structure defined by standard type constructors, including recursive types, sums, and products. The programmer writes the data structure largely as usual and it is compiled to code to be run by the prover and verifier. Using a formalization of λ we prove that all well-typed λ programs result in code that is secure under the standard cryptographic assumption of collision-resistant hash functions. We have implemented λ as an extension to the OCaml compiler, and have used it to produce authenticated versions of many interesting data structures including binary search trees, red-black trees, skip lists, and more. Performance experiments show that our approach is efficient, giving up little compared to the hand-optimized data structures developed previously.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features—Data types and structures

General Terms Security, Programming Languages, Cryptography

1. Introduction
Suppose data provider would like to allow third parties to mirror their data, providing a query interface over it to clients. The data provider wants to ensure clients that the mirrors will answer queries even on the data truthfully, even if they or no other party that compromises a mirror have an incentive to lie. As examples, the data provider might be providing stock-market data, a certificate revocation list, the for relay list, or the state of the current Bitcoin ledger [22].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PAPR’14, January 22-24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM.
http://dx.doi.org/10.1145/2535032.2535033

Such a scenario can be supported using authenticated data structures (ADS) [5, 24, 31]. ADS computations involve two roles, the prover and the verifier. The prover mirrors the role of the prover, using the data of interest and answering queries about it. The client plays the role of the verifier, passing queries to the prover and verifying that the returned results are authentic. At any point in time, the verifier holds only a short digest that can be viewed as summarizing the current contents of the data, an authentic copy of the digest is provided by the data owner. When the verifier sends a query, the prover computes the answer along with a proof that the returned result is correct; both the proof and the time to produce it are linear in the time to compute the query result. The verifier can attempt to verify the proof (in time linear in the size of the proof) using its current digest, and will accept the returned result only if the proof verifies. If the verifier is also the data provider, the verifier may also update its data verified at the prover; in this case, the result is an updated digest and the proof shows that this updated digest was computed correctly. ADS computations have two properties. Consistency implies that when both parties execute the protocol correctly, the proofs given by the verifier verify correctly and the verifier always receives the correct result. Security implies that a computationally bounded, malicious adversary can only fool the verifier into accepting an incorrect result.

Authenticated data structures can be back to Merkle [11], the well-known Merkle hash tree can be viewed as providing an authenticated version of a bounded-length array. More recently, authenticated versions of data structures as diverse as sets [23, 27], dictionaries [1, 12], range trees [15], graphs [11, 12], B-trees [21], hash trees [26], and more [15] have been proposed. In each of these cases, the design of the data structure, the supporting operations, and how they can be proved authentic have been reconsidered from scratch, involving a new, potentially tricky proof of security. Arguably, this state of affairs has hindered the advancement of new data-structures designs as previous ideas are not easily reused or repurposed. We believe that ADSs will make their way into systems more often if they become easier to build.

This paper presents λ (pronounced “lambdaaltet”), a language for programming authenticated data structures. λ captures the first generic, language-based approach to building dynamically authenticated data structures with provable guarantees. The key observation underlying λ’s design is that, whenever the data structure or operation, the computations performed by the prover and verifier can be made structurally the same: the prover constructs the proof at key points when executing queries and the verifier checks a proof by using the “Imports” the query, checking at each key point that the computation is self-consistent.

This property is sometimes called soundness but we reserve the term to avoid confusion with its standard usage in programming languages.

Simple language extension, data structure written mostly as usual. Different code generated for client and server

Expresses many prior ADSs

Proved that type correctness implies authenticity

Adversary can only fool client by inverting one-way hash

One proof for all!
Elements of PL Research

- **Design**: What feature, analysis, transformation, etc.?
- **Mathematics and proof**: What does it mean? Why is what you are doing correct?
- **Implementation**: How do you implement this language, analysis, transformation ... ?
- **Empirical evaluation**: Does the design/implementation work (most of the time)?
PL Research Toolbox

- **Language specification** (what features, syntax)
- **Semantics** (operational, denotational)
- **Static reasoning** (logics, types, static analysis)
- **Dynamic reasoning** (tests, monitors, profiles)
- **Implementation** (compilation, interpretation, services)
What’s Next: A Tour

- Disclaimer: This is my perspective
  - It is not comprehensive
  - It is probably wrong (hopefully only a little)
  - But it will give you some sense of the field
Implementation
Machines Don’t Run our Programs

Example: The same

Example: A function in hexadecimal representation of 32-bit

Through working with

Although few programs are written in machine language, programmers often become adept at reading it

By a program called a

And data in a more readable form such as

Fibonacci number

Has little

Calculator as above, but in x86 assembly language using
Other Programs Make it Possible

Three main implementation strategies:

- **Interpreter**: Runs any program in language $Q$
- **Compiler**: Converts a program in language $Q$ to one in language $L$, for which you have a machine or interpreter
- **Hybrid**: A just-in-time (JIT) **compiler** compiles the program as it interprets it
Run-time Services

- Various components support language abstractions
  - Garbage collector frees unneeded memory
  - Thread system runs different threads
  - Libraries implement key (parts of) language abstractions (e.g., strings, numbers, networking)
Research Directions

- Compiler/interpreter optimization: Register allocation, memory hierarchy optimization, use of special hardware (e.g., GPUs), partial evaluation, …
- Garbage collection algorithms: Parallel, concurrent, incremental, space-efficient, real-time, hybrid, …
- JIT implementation & optimization: fast tracing/profiling, on-stack replacement, …
- Domain-specific techniques: Probabilistic programming, neural nets, …
- The POPL’19 paper (language implementation not POPL’s main theme):
  - Efficient parameterized algorithms for data packing, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, Andreas Pavlogiannis
Semantics
Formal Semantics

❖ To work with programs, we must know what they mean
❖ *Semantics* comes from the Greek *semaino*, “to mean”
❖ Most language semantics are informal. But we can do better by making them *formal*. Two main styles:
  ❖ **Operational semantics** (like an interpreter)
  ❖ **Denotational semantics** (like a compiler)

Formal semantics is a key PL tool
Operational Semantics

- Evaluation is described as transitions (aka reductions) in some abstract machine; a **definitional interpreter**
  - The *meaning* of a program is its fully reduced form
  - `let x=2 in x+3 \rightarrow 2+3 \rightarrow 5`
- Can model *many* programming language features
  - Concurrency, non-determinism, run-time cost, higher-order functions, probabilistic choice, ...
- This is the most popular style of semantics
The meaning of a program is defined as a **mathematical object**, e.g., a function or number.

Typically define an **interpretation function** $⟦ ⟧$

- Meaning of program fragment given by meaning of its components; $⟦e_1+e_2⟧ = ⟦e_1⟧ + ⟦e_2⟧$

- Gets interesting when we try to find denotations of loops or recursive functions, cyclic heap state

- Particularly useful for **equational reasoning**
Research Directions

❖ PL researchers frequently use operational semantics to model new languages and language features
❖ Or model features in a new way, to facilitate some other advance (e.g., proof of a property)
❖ Also, new techniques for semantics modeling, particularly for domain-specific computations
Some POPL’19 Papers

- **Skeletal semantics and their interpretations**, Martin Bodin, Philippa Gardner, Thomas Jensen, Alan Schmitt
- **A calculus for Esterel: if can, can. if no can, no can.** Spencer P. Florence, ShuHung You, Jesse A. Tov, Robert Bruce Findler
- **Familial monads and structural operational semantics**, Tom Hirschowitz
- **Game semantics for quantum programming**, Pierre Clairambault, Marc De Visme, Glynn Winskel
- **ISA semantics for ARMv8a, RISCv, and CHERIMIPS**, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, Peter Sewell
Static Reasoning
Static Analysis

- Goal: establish property $R$ of (all of) a program $P$’s executions. Examples:
  - $R = \text{no run-time failures}$, or $R = \text{always terminates}$
- But: Reasoning via the semantics directly — testing — is infeasible (i.e., infinite number of runs)
- Many static analysis algorithms/techniques: type systems, dataflow analysis, abstract interpretation, symbolic execution, constraint-based analysis, …
Soundness and Completeness

❖ Suppose a static analysis $S$ attempts to prove property $R$ of program $P$. For example,

❖ $R = \text{program has no run-time errors (e.g., div-by-zero)}$

❖ $S(P) = \text{true}$ implies $P$ has no run-time errors
Soundness and Completeness

- **Analysis** $S$ is **sound** iff for all $P$,
  - $S(P) = \text{true} \implies P$ exhibits $R$

- **Analysis** $S$ is **complete** iff for all $P$,
  - $P$ exhibits $R \implies S(P) = \text{true}$
Abstract Interpretation

- Rice’s Theorem: Any non-trivial property \( R \) is **undecidable** (not both sound and complete)

\begin{itemize}
  \item Abstract the behavior of the program, so that
  \item Proof about abstraction \( \implies \) proof about real thing
  \item Seminal papers: Cousot and Cousot, 1977, 1979
\end{itemize}

Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints: Most cited POPL paper ever!
Abstract domain = \{?,0,+,\}

Need for ? arises because of the abstraction
Abstract Domains, and Semantics

- Many abstract domains developed
  - Signs (previous slide), intervals \([l, u]\) where \(l \leq n \leq u\), convex polyhedra, octagons, pentagons, …
- Abstract semantics for language constructs
  - Basic constructs (sequence, assignment, etc.) easy
  - Key challenge is loops: Need to ensure their analysis terminates (idea: “widening”)
All static analyses can be viewed as abstract interpretation. Easy to relate to data flow analysis, symbolic execution, typing (later), etc.

But

precise setup can differ significantly, with different precision/performance tradeoffs.
Research Directions: Static Analysis

- Analyses for **new properties**
  - Side-channel freedom, data-race freedom, proper resource use, tainting, bias-freedom in ML-inferred algorithms, …

- **Implementation methods,**
  - often to better trade off performance and precision: new / faster abstract domains, new heuristics/search, connections to machine learning methods, …

- Analyses for **new applications**
Some POPL’19 papers

- **A true positives theorem for a static race detector**, Nikos Gorogiannis, Peter W. O’Hearn, Ilya Sergey

- **Concerto: a framework for combined concrete and abstract interpretation**, John Toman, Dan Grossman

- **A²I: abstract² interpretation**, Patrick Cousot, Roberto Giacobazzi, Francesco Ranzato

- **An abstract domain for certifying neural networks**, Gagandeep Singh, Timon Gehr, Markus Püschel, Martin Vechev

- **Context-, flow-, and field-sensitive dataflow analysis using synchronized Pushdown systems**, Johannes Späth, Karim Ali, Eric Bodden

- **Fast and exact analysis for LRU caches**, Valentin Touzeau, Claire Maïza, David Monniaux, Jan Reineke

- **Refinement of path expressions for static analysis**, John Cyphert, Jason Breck, Zachary Kincaid, Thomas Reps
Formal Verification

- This is a static analysis with a very specific property — functional correctness

- Given program P and a spec $\phi$ relating inputs to outputs. Prove that P meets the spec, ie $\forall x. \phi(x, P(x))$

- Lots of approaches to do this based on ideas like verification condition generation, weakest preconditions, dependent types, etc.

Program Synthesis

- Don’t prove the program — construct it automatically!
- Given a spec $\phi$ relating inputs to outputs. **Find a program $P$ that meets the spec**, ie $\forall x. \phi(x,P(x))$
- Many methods being explored
  - explicit search, symbolic search, hybrid search, type-directed search, derivational synthesis, domain-specific synthesis, …
Some POPL’19 papers

- **Verification**
  - **Decoupling lock-free data structures from memory reclamation for static analysis**, Roland Meyer, Sebastian Wolff
  - **Pretend synchrony: synchronous verification of asynchronous distributed programs**, Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, Ranjit Jhala

- **Synthesis**
  - **Bayesian synthesis of probabilistic programs for automatic data modeling**, Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, Vikash K. Mansinghka
  - **Structuring the synthesis of heap-manipulating programs**, Nadia Polikarpova, Ilya Sergey
  - **Hamsaz: replication coordination analysis and synthesis**, Farzin Houshmand, Mohsen Lesani
Type Systems

❖ A **type system** is
  ❖ a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. —Pierce

❖ They are good for
  ❖ **Detecting errors** (don’t add an integer and a string)
  ❖ **Abstraction** (hiding representation details)
  ❖ **Documentation** (tersely summarize an API)
  ❖ Designs trade off precision, efficiency, readability
Example Type System

\[
e ::= n \mid e + e \\
\mid \text{true} \mid \text{false} \mid e = e \\
\mid \text{if } e \text{ then } e \text{ else } e
\]

\[
\tau ::= \text{int} \mid \text{bool}
\]

Judgment \( \vdash e : \tau \) means “expression \( e \) has type \( \tau \)”

Examples

\( \vdash \text{true} : \text{bool} \) “true has type bool”

\( \vdash 1 + 2 : \text{int} \)

\( \vdash \text{if } 1 = 1 \text{ then } \text{true} \text{ else } \text{false} : \text{bool} \)

\( \vdash \text{if } \ldots \text{ then } 1 \text{ else } \text{false} : ? \) error

\( \vdash \text{if } \text{true} \text{ then } 1 \text{ else } \text{false} : ? \) doesn't check
Rules of Inference

⊢ $e : \tau$ means “expression $e$ has type $\tau$”

<table>
<thead>
<tr>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ $e1 : \text{int}$</td>
<td>⊢ $e1 + e2 : \text{int}$</td>
</tr>
<tr>
<td>⊢ $e2 : \text{int}$</td>
<td></td>
</tr>
<tr>
<td>⊢ $e1 : \tau$</td>
<td>⊢ $e1 = e2 : \text{bool}$</td>
</tr>
<tr>
<td>⊢ $e2 : \tau$</td>
<td></td>
</tr>
</tbody>
</table>

Axioms

⊢ $n : \text{int}$ ⊢ $\text{true} : \text{bool}$ ⊢ $\text{false} : \text{bool}$

NB: Operational semantics often also expressed using rules of inference

$e1$ and $e2$ must have same type $\tau$
Soundness

- If $\vdash e : \tau$ then either
  - $e$ reduces to a value $v$ of type $\tau$, or $e$ diverges*
  - (for our example, values $v$ are $n$, true, false)
  - Reduction often defined as operational semantics

- Corollary: $e$ will never get “stuck”
  - i.e., never fails to reduce a non-value
  - which constitutes a “run-time error”

- Proof by induction on typing derivation

*Divergence not possible in this simple language, but is for real ones!
Types and Static Analysis

- Relating to AI, a type is an abstract domain
- Proving $P$ is well-typed is a static analysis of $P$
- By type safety, analysis is **sound** (but not **complete**)
- Viewed as a static analysis, types need not be present in (or even defined by) the language
- Analysis becomes a kind of **type inference**
Properties by Typing

- Idea: Formulate an operational semantics for which violation of a property $R$ results in a stuck program. E.g.,
  - The program *divides by zero*, dereferences a *null pointer*, or accesses an array *out of bounds*
  - A thread attempts to *dereference a pointer without holding a lock*
  - The program *uses tainted data* (i.e., from an adversary) where untainted data expected
  - A program *dereferences a dangling pointer*
- Formulate a type system to enforce $R$; prove type safety
Example: ADS

- Define **language extension** for using recursive hashes, the key ADS feature
- Define **operational semantics** that models one-way nature of hashes, with variants for verifier (client), prover (server), and "ideal"
- Define **type system** that ensures proper use of hashed values
- Prove security (in the style of **type safety**): Only by finding a hash collision can the server fool the client
Research Directions: Types

- Type systems have a strong connection to formal logic: Ideas go back and forth
- Type system development is a central PL activity
  - add precision (e.g., dependent and refinement types),
  - prove new properties (e.g., abstraction, free theorems),
  - support new language constructs and/or domain-specific properties
Some POPL’19 Papers

- **Intersection types and runtime errors in the pi-calculus**, Ugo Dal Lago, Marc de Visme, Damiano Mazza, Akira Yoshimizu
- **Sound and complete bidirectional type-checking for higher-rank polymorphism with existentials and indexed types**, Joshua Dunfield, Neelakantan R. Krishnaswami
- **Exceptional asynchronous session types: session types without tiers**, Simon Fowler, Sam Lindley, J. Garrett Morris, Sára Decova
- **Polymorphic symmetric multiple dispatch with variance**, Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., Sukyoung Ryu
- **Abstracting extensible data types: or, rows by any other name**, J. Garrett Morris, James McKinna
Dynamic Reasoning
Dealing with False Alarms

- Recall Rice’s theorem: no sound and complete static analysis (for interesting properties/languages)
- Type systems reject safe programs; static analyses emit false alarms
- Idea: Alarm if property $R$ is violated during execution
- Do so during testing, and/or during deployment (e.g., dynamic typing)
Hybrid Reasoning

- Most type systems do not try to prove that an index to an array is always within its bounds
  - Compiler adds a **dynamic check** (monitor) if unsure
  - Trades added precision with run-time overhead (and chance of failure)
- Such type systems **combine static and dynamic reasoning**
- **Gradual typing**: Hybrid support for static types (proved once and for all) and dynamic types (checked at run-time)
Gradual Typing is Popular

Dart
flow-typed
TS
Reticulated
Some POPL’19 Papers

- **Adventures in monitorability: from branching to linear time and back again**, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, Karoliina Lehtinen
- **Modular quantitative monitoring**, Rajeev Alur, Konstantinos Mamouras, Caleb Stanford
- **Gradual type theory**, Max S. New, Daniel R. Licata, Amal Ahmed
- **LWeb: information flow security for multitier web applications**, James Parker, Niki Vazou, Michael Hicks
- **From fine- to coarse-grained dynamic information flow control and back**, Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, Deian Stefan
So as you can see …

PL has a substantial toolbox of mathematics and implementation techniques that are widely applicable.

With these: We can make it more general, more elegant, more direct, more efficient, more reliable, more secure …

Wow! Thanks for getting me up to date…
Recap: What is PL Research?

PL research views the programming language as having a central place in solving computing problems.

A PL researcher:

- develops *general abstractions*, or *building blocks*, for solving problems, or classes of problems,
- considers software *behavior in a rigorous and general way*, e.g., to prove that (classes of) programs enjoy properties we want, and/or eschew properties we don’t.