Grad PL vs. The World

Grad PL Conclusions

- You are now equipped to read the most influential papers in PL.
- You can also recognize PL concepts and will know what to do when they come up in your research.

Questions

- Model Checking, Abstraction Refinement, SLAM, Large-Step Opsem, Contextual Opsem, Structural Induction, Theorem Proving, Simplex, Proof Checking, Axiomatic Semantics, VCGen, Symbolic Execution, Invariant Detection, Abstract Interpretation, Lambda Calculus, Monomorphic and Polymorphic Type Systems, Recursive and Dependent Types, Pi Calculus, Machine Learning, Fault Localization, Program Repair, Instructor.

ACM SIGPLAN Most Influential Paper Awards

- SIGPLAN presents these awards to the author(s) of a paper presented at ICFP, OOPSLA, PLDI, and POPL held 10 years prior to the award year. The award includes a prize of $1,000 to be split among the authors of the winning paper. The papers are judged by their influence over the past decade. Each award is presented at the respective conference.

- 2020 (POPL 2010): **From program verification to program synthesis.** Saurabh Srivastava, Sumit Gulwani, Jeffrey Foster.
- The paper greatly advanced our ability to synthesize programs from logical specifications. It was based on the insight that much of the work carried out by a program verifier could be repurposed not just for checking that code matches a specification, but also to synthesize code that does. The user specifies the input-output behavior as a logical formula, and also provides structural and resource constraints, which describe a template language for the space of possible programs. … The authors were able to synthesize a range of clever algorithms, which served as inspiration for some of the massive effort on verification-based program synthesis over the past decade.

From program verification to program synthesis

	Benchmark	Verif.	Synthesis	Ratio
Arith. (VS^3_{QA})	Swap two	0.11	0.12	1.09
	Strassen's	0.11	4.98	45.27
	Sqrt (linear search)	0.84	9.96	11.86
	Sqrt (binary search)	0.63	1.83	2.90
	Bresenham's	166.54	9658.52	58.00
Sorting (VS^3_{PA})	Bubble Sort	1.27	3.19	2.51
	Insertion Sort	2.49	5.41	2.17
	Selection Sort	23.77	164.57	6.92
	Merge Sort	18.86	50.00	2.65
	Quick Sort	1.74	160.57	92.28
Dynamic Prog. (VS^3_{AX})	Fibonacci	0.37	5.90	15.95
	Checkerboard	0.39	0.96	2.46
	Longest Common Subseq.	0.53	14.23	26.85
	Matrix Chain Multiply	6.85	88.35	12.90
	Single-Src Shortest Path	46.58	124.01	2.66
	All-pairs Shortest Path[1]	112.28	(i) 226.71 (ii) 750.11	(i) 2.02 (ii) 6.68

Table 3. (a) Arithmetic (b) Sorting (c) Dynamic Programming. For each category, we indicate the tool used both for verification and

From program verification to program synthesis

When called with inputs $\vec{v_{in}}$ that satisfy $F_{pre}(\vec{v_{in}})$ the program terminates, and the resulting outputs $\vec{v_{out}}$ satisfy $F_{post}(\vec{v_{in}}, \vec{v_{out}})$.

Axiomatic Semantics

$$\{\phi_{pre}\}x := e_x; y := e_y\{\phi_{post}\}$$

Then we can use Hoare's axiom for assignment to generate the verification condition $\phi_{pre} \Rightarrow (\phi_{post}[y \mapsto e_y])[x \mapsto e_x]$. However,

Model Checking

Instead we will represent the computation as a transition system which provides a much cleaner mechanism for reasoning when program statements are unknown. A *transition* in a transition system

37]. Synthesis from LTL specification has also been consid-

Abstract Interpretation

wards (computing greatest-fixed point) direction starting with the approximation $\perp$ or $\top$, respectively, and iteratively refining it.

- 2018 (POPL 2008): **Multiparty asynchronous session types.** Kohei Honda, Nobuko Yoshida, Marco Carbone.
- … Session types are a type-based framework for codifying communication structures and verifying protocols in concurrent, message-passing programs. Previously, session types could only model binary (two-party) protocols. This paper generalizes the theory to the multiparty case with asynchronous communications, preventing deadlock and communication errors in more sophisticated communication protocols involving any number (two or more) of participants. The central idea was to introduce global types, which describe multiparty conversations from a global perspective and provide a means to check protocol compliance. This work has inspired numerous authors to build on its pioneering foundations …

Multiparty asynchronous session types

Pi Calculus

2. Multiparty Asynchronous Sessions

2.1 Syntax for Multiparty Sessions

Several versions of the π-calculi with session types are proposed in the literature; the paper (Yoshida and Vasconcelos 2007) offers detailed discussions and analysis of their typing systems. We use a simple extension of the original language in (Honda et al. 1998; Takeuchi et al.

Figure 2 Structural congruence.

Read me in English

Scope Extrusion

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin \mathsf{fn}(Q) \qquad (\nu n n')P \equiv (\nu n' n)P$$

$$(\nu n)\mathbf{0} \equiv \mathbf{0} \qquad \mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \qquad (\nu s_1 .. s_n)\Pi_i s_i : \emptyset \equiv \mathbf{0}$$

$$\mathsf{def}\ D\ \mathsf{in}\ (\nu n)P \equiv (\nu n)\mathsf{def}\ D\ \mathsf{in}\ P \qquad \text{if } n \notin \mathsf{fn}(D)$$

$$(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q) \qquad \text{if } \mathsf{dpv}(D) \cap \mathsf{fpv}(Q) = \emptyset$$

$$\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv \mathsf{def}\ D\ \mathsf{and}\ D'\ \mathsf{in}\ P \quad \text{if } \mathsf{dpv}(D) \cap \mathsf{dpv}(D') = \emptyset$$

Multiparty asynchronous session types

- They use a slightly different syntax. Look at this **[Recv]** rule. How do they write send? How do they write receive?

Look here ...

$$s?(\tilde{x});P \mid s:\tilde{v}\cdot\tilde{h} \;\rightarrow\; P[\tilde{v}/\tilde{x}] \mid s:\tilde{h} \qquad \text{[Recv]}$$

$$s?(\!(\tilde{t})\!);P \mid s:\tilde{t}\cdot\tilde{h} \;\rightarrow\; P \mid s:\tilde{h} \qquad \text{[SRec]}$$

$$s \triangleright \{l_i : P_i\}_{i\in I} \mid s:l_j\cdot\tilde{h} \;\rightarrow\; P_j \mid s:\tilde{h} \quad (j \in I) \qquad \text{[Branch]}$$

$$\text{if } e \text{ then } P \text{ else } Q \;\rightarrow\; P \quad (e \downarrow \text{true}) \qquad \text{[IfT]}$$

$$\text{if } e \text{ then } P \text{ else } Q \;\rightarrow\; Q \quad (e \downarrow \text{false}) \qquad \text{[IfF]}$$

Multiparty asynchronous session types

5. Safety and Progress

This section establishes the fundamental behavioural properties of typed processes. We follow three technical steps:

1. We extend the typing rules to include those for runtime processes which involve message queues.
2. We define reduction over session typings which eliminates a pair of minimal complementary actions from local types.
3. We then relate the reduction of processes and that of typings: showing the latter follows the former gives us *subject reduction* (Theorem 5.4), *safety* (Theorem 5.5) and *session fidelity* (Corollary 5.6), while showing the former follows the latter under a certain condition gives us *progress* (Theorem 5.12).

You know three of these four. Explain them to me.

- 2008 (POPL 1998): **From System F to Typed Assembly Language**, Greg Morrisett, David Walker, Karl Crary, and Neal Glew.
- ... began a major development in the application of type system ideas to low level programming. The paper shows how to compile a high-level, statically typed language into TAL, a typed assembly language defined by the authors. The type system for the assembly language ensures that source-level abstractions like closures and polymorphic functions are enforced at the machine-code level while permitting aggressive, low-level optimizations such as register allocation and instruction scheduling. This infrastructure provides the basis for ensuring the safety of untrusted low-level code artifacts, regardless of their source. A large body of subsequent work has drawn on the ideas in this paper, including work on proof-carrying code and certifying compilers.

From System F to Typed Assembly Language

Polymorphic Function Type

Tuple Type

terms. The syntax for λ^F appears below:

$$
\begin{array}{llll}
\textit{types} & \tau & ::= & \alpha \mid \textit{int} \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau \mid \langle \tau_1, \cdots, \tau_n \rangle \\
\textit{terms} & e & ::= & x \mid i \mid \textit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid \\
& & & e[\tau] \mid \langle e_1, \cdots, e_n \rangle \mid \pi_i(e) \mid \\
& & & e_1\, p\, e_2 \mid \textit{if0}(e_1, e_2, e_3) \\
\textit{prims} & p & ::= & + \mid - \mid \times
\end{array}
$$

Polymorphic Function Type Application

Tuple Field Selection

Polymorphic Function Creation

From System F to Typed Assembly Language

... but you know it.

We interpret λ^{F} with a conventional call-by-value operational semantics (not presented here). The static semantics is specified as a set of inference rules that allow us to conclude judgments of the form $\Delta; \Gamma \vdash_{F} e : \tau$ where Δ is a set containing the free type variables of Γ, e, and τ; Γ assigns types to the free variables of e; and τ is the type of e.

Typing Judgment

As a running example, we will be considering compilation and evaluation of 6 factorial:

What is "fix" like?

$$(\mathit{fix}\ f(n{:}\mathit{int}){:}\mathit{int}.\ \mathit{if0}(n, 1, n \times f(n-1)))\ 6.$$

The operational semantics of TAL is presented in Figure 7 as a deterministic rewriting system $P \longmapsto P'$ that maps programs to programs. Althoug[h] ...

Operational Semantics, Forward Symex

$(H, R, S) \longmapsto P$ where	
if $S =$	then $P =$
add $r_d, r_s, v; S'$	$(H, R\{r_d \mapsto R(r_s) + \hat{R}(v)\}, S')$ and similarly for mul and sub
bnz $r, v; S'$ when $R(r) = 0$	(H, R, S')
bnz $r, v; S'$ when $R(r) = i$ and $i \neq 0$	$(H, R, S''[\vec{\tau}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\tau}]$ and $H(\ell) = \mathsf{code}[\vec{\alpha}]\Gamma.S''$
jmp v	$(H, R, S'[\vec{\tau}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\tau}]$ and $H(\ell) = \mathsf{code}[\vec{\alpha}]\Gamma.S'$
ld $r_d, r_s[i]; S'$	$(H, R\{r_d \mapsto w_i\}, S')$ where $R(r_s) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle$ with $0 \leq i < n$
malloc $r_d[\tau_1, \ldots, \tau_n]; S'$	$(H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n\rangle\}, R\{r_d \mapsto \ell\}, S')$ where $\ell \notin H$

" if false ... ; S' "

"L" is fresh. Small-step opsem for allocation.

From System F to Typed Assembly Language

Lemma 5.1 (Subject Reduction) *If* $\vdash_{\text{TAL}} P$ *and* $P \longmapsto P'$ *then* $\vdash_{\text{TAL}} P'$.

Type Preservation

Lemma 5.2 (Progress) *If* $\vdash_{\text{TAL}} P$ *then either:*

cf. Decomposition

1. *there exists* P' *such that* $P \longmapsto P'$, *or*

2. P *is of the form* $(H, R\{\mathtt{r1} \mapsto w\}, \mathtt{halt}[\tau])$ *where there exists* Ψ *such that* $\vdash_{\text{TAL}} H : \Psi$ *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$.

Corollary 5.3 (Type Soundness) *If* $\vdash_{\text{TAL}} P$, *then there is no stuck* P' *such that* $P \longmapsto^{*} P'$.

- 2011 (POPL 2001): **Anytime, Anywhere: Modal Logics for Mobile Ambients**, Luca Cardelli and Andrew D. Gordon.
- ... helped spur a flowering of work in the area of process calculi that continues today. The paper focused on modal logics for reasoning about both temporal and spacial modalities for ambient behaviours, demonstrating techniques that also apply to other process calculi (even those without an explicit notion of location), so contributing to excitement in an area that was growing at that time and continues. The work has led to application of concurrency theory in fields as diverse as security, safety critical applications, query languages for semistructured data, and systems biology.

Anytime, Anywhere: Modal Logics for Mobile Ambients

Processes

$P, Q, R ::=$	processes	
$\mathbf{0}$	void	spatial
$P \mid Q$	composition	spatial
$!P$	replication	spatial
$M[P]$	ambient	spatial
$M.P$	capability action	temporal
$(n).P$	input action	temporal
$\langle M \rangle$	output action	temporal
$M ::=$	messages	
n	name	names
$in\ M$	can enter into M	capabilities
$out\ M$	can exit out of M	capabilities
$open\ M$	can open M	capabilities
ε	null	paths
$M.M'$	composite	paths

"Pi Calculus"-esque

cf. pi send and receive

Anytime, Anywhere: Modal Logics for Mobile Ambients

Structural Congruence

$P \equiv P$	(Struct Refl)
$P \equiv Q \;\Rightarrow\; Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \;\Rightarrow\; P \equiv R$	(Struct Trans)
$P \equiv Q \;\Rightarrow\; P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \;\Rightarrow\; !P \equiv !Q$	(Struct Repl)
$P \equiv Q \;\Rightarrow\; M[P] \equiv M[Q]$	(Struct Amb)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	
$P \mid \mathbf{0} \equiv P$	
$!(P \mid Q) \equiv !P \mid !Q$	
$!\mathbf{0} \equiv \mathbf{0}$	
$!P \equiv P \mid !P$	
$!P \equiv !!P$	

Pi Calculus

Synchronous Rendezvous

Reduction

$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$	(Red Open)
$(n).P \mid \langle M \rangle \longrightarrow P\{n \leftarrow M\}$	(Red Comm)
$P \longrightarrow Q \;\Rightarrow\; n[P] \longrightarrow n[Q]$	(Red Amb)
$P \longrightarrow Q \;\Rightarrow\; P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \;\Rightarrow\; P' \longrightarrow Q'$	(Red $\equiv$)

$\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$

- 2011 (PLDI 2001): **Automatic predicate abstraction of C programs**, Thomas Ball, Rupak Majumdar, Todd Millstein, Sriram K. Rajamani.
- ... presented the underlying predicate abstraction technology of the SLAM project for checking that software satisfies critical behavioral properties of the interfaces it uses and to aid software engineers in designing interfaces and software that ensure reliable and correct execution. The technology is now part of Microsoft's Static Driver Verifier in the Windows Driver Development Kit. This is one of the earliest examples of automation of software verification on a large scale and the basis for numerous efforts to expand the domains that can be verified.

Automatic predicate abstraction of C programs

Axiomatic Semantics

4.1 Weakest Preconditions and

For a statement s and a predicate φ, let $WP(s, \varphi)$ denote the *weakest liberal precondition* [16, 20] of φ with respect to statement s. $WP(s, \varphi)$ is defined as the weakest predicate whose truth before s entails the truth of φ after s terminates (if it terminates). Let "**x = e**" be an assignment, where x is a scalar variable and e is an expression of the appropriate type. Let φ be a predicate. By definition $WP(\mathtt{x = e}, \varphi)$ is φ with all occurrences of x replaced with e, denoted $\varphi[e/x]$. For example:

$$WP(\mathtt{x=x+1}, x < 5) = (x + 1) < 5 = (x < 4)$$

- 2009 (PLDI 1999): **A Fast Fourier Transform Compiler**, Matteo Frigo
- ... describes the implementation of genfft, a special-purpose compiler that produces the performance critical code for a library, called FFTW (the "Fastest Fourier Transform in the West"), that computes the discrete Fourier transform. FFTW is the predominant open fast Fourier transform package available today, as it has been since its introduction a decade ago. genfft demonstrated the power of domain-specific compilation—FFTW achieves the best or close to best performance on most machines, which is remarkable for a single package. By encapsulating expert knowledge from the FFT algorithm domain and the compiler domain, genfft and FFTW provide a tremendous service to the scientific and technical community by making highly efficient FFTs available to everyone on any machine. As well as being the fastest FFT in the West, FFTW may be the last FFT in the West as the quality of this package and the maturity of the field may mean that it will never be superseded, at least for computer architectures similar to past and current ones.

A Fast Fourier Transform Compiler

```
type node =
  | Num of Number.number
  | Load of Variable.variable
  | Store of Variable.variable * node
  | Plus of node list
  | Times of node * node
  | Uminus of node
```

Figure 3: Objective Caml code that defines the `node` data type, which encodes an expression dag.

No joke. cf. Homeworks!

- 2014 (POPL 2004): **Abstractions from proofs**, Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth McMillan.
- ... the authors demonstrated a fundamental generalization of Craig interpolation to program analysis by predicate abstraction, opening the door for interpolation to be applied to abstraction refinement of infinite-state systems. This work showed how interpolation offers a fundamental way to explain abstraction refinement in a logical framework and has led to many extensions to increase the power of abstraction in program analysis.

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

Side Condition

$$\textsc{Comb}\ \frac{\Gamma \vdash 0 \leq x \quad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

Proof Checking, Abstract Interpretation

$$\textsc{Contra}\ \frac{\{\phi_1, \ldots, \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, \ldots, \neg\phi_n}\ c < 0$$

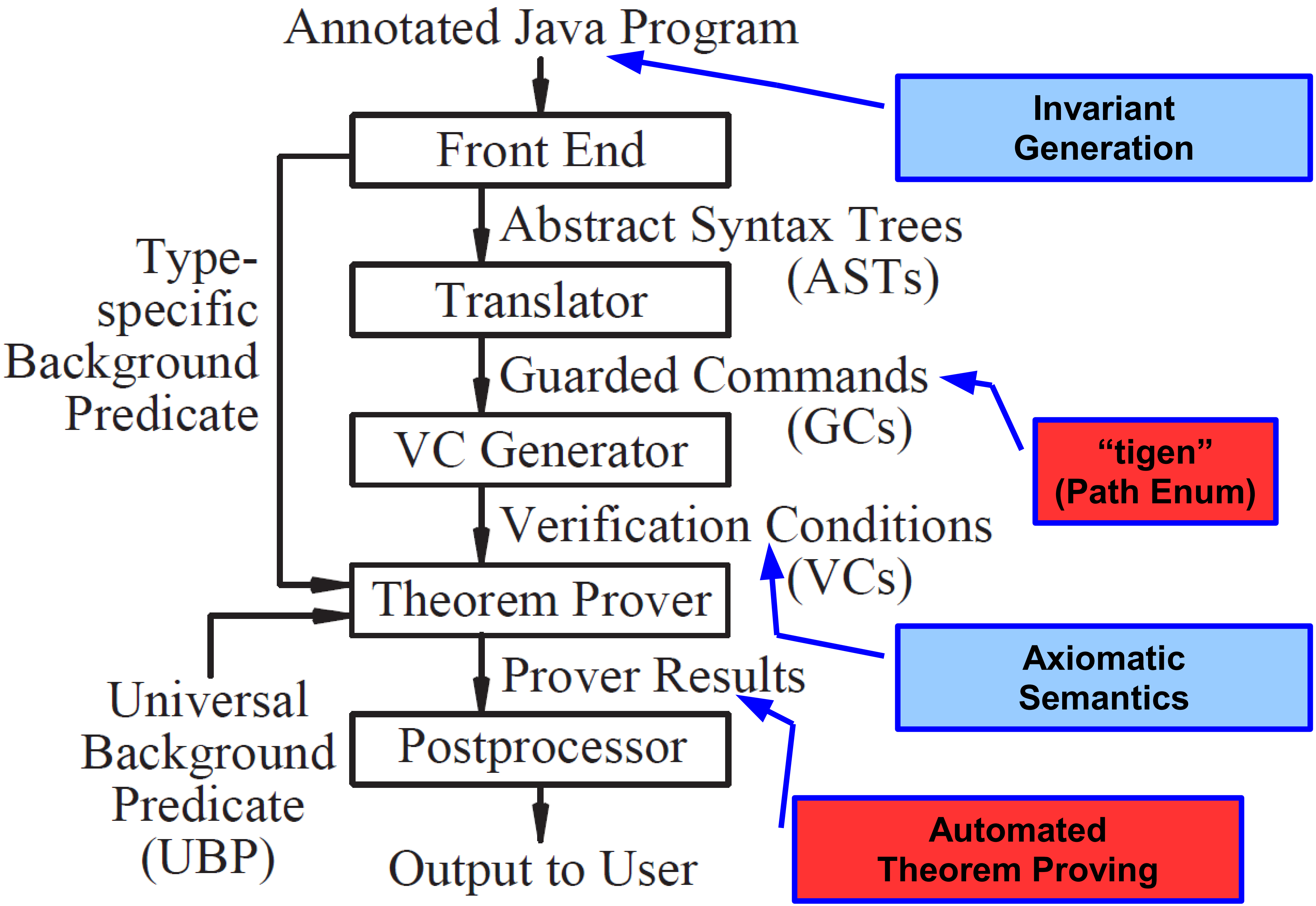
```
P -> False
---------------
!P
```

$$\textsc{Res}\ \frac{\Gamma \vdash \{\phi\} \cup \Theta \quad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

```
P -> X, !P -> Y
---------------------
X || Y
```

Figure 3. Proof system.

- 2012 (PLDI 2002): **Extended Static Checking for Java**, Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata
- ... marks a turning point in the field of static checking, describing pragmatic design decisions that promote practicality over completeness. Pioneered in ESC/Modula-3, techniques from ESC/Java are now widely used in various forms in Microsoft's development tools, notably as part of Code Contracts which ships with VisualStudio. Recent innovations strongly influenced by ESC/Java include refinement types for Haskell, and verification of Eiffel programs.


Annotated Java Program
Invariant Generation
Front End
Abstract Syntax Trees (ASTs)
Type-specific Background Predicate
Translator
Guarded Commands (GCs)
"tigen" (Path Enum)
VC Generator
Verification Conditions (VCs)
Theorem Prover
Axiomatic Semantics
Prover Results
Universal Background Predicate (UBP)
Postprocessor
Automated Theorem Proving
Output to User

- 2017 (POPL 2007): **JavaScript Instrumentation for Browser Security**, Dachuan Yu, Ajay Chander, Nayeem Islam, Igor Serikov.
- ... presents one of the earliest models of the interaction between the browser and JavaScript. It uses this model to work out the formalization and dynamic enforcement of rich security policies. Since then, people have routinely discovered additional, pernicious security problems based on this model. Eliminating these problems remains an important challenge to this day.
- Looking back, the selected paper made a prescient, and influential, contribution to understanding these JavaScript-based security problems. The authors chose a formal, semantic approach to model these problems and potential solutions, while remaining true to the complicated characteristics that make both JavaScript and the browser real-world artifacts.

Secretly IMP?

$$
\begin{array}{rrl}
(Script) & P & ::= \texttt{skip} \mid x = E \mid P; P \\
& & \mid \texttt{if } E \texttt{ then } P \texttt{ else } P \\
& & \mid \texttt{while } E \texttt{ do } P \mid f(\vec{E}) \mid \texttt{act}(A) \\
& & \mid \texttt{write}(E) \\
(Expression) & E & ::= x \mid D \mid op(\vec{E}) \\
(Action) & A & ::= \epsilon \mid \texttt{newWin}(x, E) \mid \texttt{closeWin}(E) \\
& & \mid \texttt{loadURL}(E) \mid \texttt{readCki}(x) \\
& & \mid \texttt{writeCki}(E) \mid \texttt{secOp}(\vec{E}) \\
(Value\ Action) & A^v & ::= \epsilon \mid \texttt{newWin}(\_, D) \mid \texttt{closeWin}(D) \\
& & \mid \texttt{loadURL}(D) \mid \texttt{readCki}(\_) \\
& & \mid \texttt{writeCki}(D) \mid \texttt{secOp}(\vec{D})
\end{array}
$$

Figure 4. CoreScript syntax

- Read and explain their “if” and “while” rules to me …

If $D =$	then $focus(D) =$	and $stepDoc(D, \chi) =$
$js\ P$ where $P \in \{\texttt{skip}, x = E, \texttt{act}(A)\}$	P	ε (empty string)
$js\ \texttt{write}(E)$	$\texttt{write}(E)$	D where $\chi \vdash E \Downarrow D$
$js\ P_1; P_2$	$focus(js\ P_1)$	$\texttt{jux}\ D\ (js\ P_2)$ where $D = stepDoc(js\ P_1, \chi)$
$js\ \texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2$	$\texttt{if}\ E\ \texttt{then}\ P_1\ \texttt{else}\ P_2$	$js\ P_1$ if $\chi \vdash E \Downarrow \texttt{true}$ $js\ P_2$ if $\chi \vdash E \Downarrow \texttt{false}$
$js\ \texttt{while}\ E\ \texttt{do}\ P$	$\texttt{while}\ E\ \texttt{do}\ P$	$js\ \texttt{if}\ E\ \texttt{then}\ (P; \texttt{while}\ E\ \texttt{do}\ P)\ \texttt{else}\ \texttt{skip}$
$js\ f(\vec{E})$	$f(\vec{E})$	$js\ P[\vec{D}/\vec{x}]$ where $\chi \vdash \vec{E} \Downarrow \vec{D}$ and $\chi(f) = (\vec{x})P$
$F\ \vec{D^v}\, D'\, \vec{D}$ where D' is not a value document	$focus(D')$	$F\ \vec{D^v}\, D''\, \vec{D}$ where $D'' = stepDoc(D', \chi)$

What is "type preservation" again?

Lemma 2 (Orthodoxy Preservation) If W is orthodox and $\vdash_\delta (W, q) \rightsquigarrow (W', q') : A^v$, then W' is orthodox.

Proof sketch: By definition of the step relation ($\rightsquigarrow$), with induction on the structure of documents. The case of executing $\texttt{write}(E)$ is no possible because W is orthodox. In the case of executing $\texttt{instr}(E)$, the operational semantics produces an instrumented document to replace the focus node. Orthodoxy thus follows from Lemma 1. In all other cases, the operational semantics may obtain document pieces from other program components, which are orthodox by assumption. $\square$

How did they prove it?

Your Questions

Model Checking, Abstraction Refinement, SLAM, Large-Step Opsem, Contextual Opsem, Structural Induction, Theorem Proving, Simplex, Proof Checking, Axiomatic Semantics, VCGen, Symbolic Execution, Invariant Detection, Abstract Interpretation, Lambda Calculus, Monomorphic and Polymorphic Type Systems, Recursive and Dependent Types, Pi Calculus, Machine Learning, Fault Localization, Program Repair, Instructor.

Grad PL Conclusions

- You are now equipped to read the most influential papers in PL.
- You can also recognize PL concepts and will know what to do when they come up in your research.